Please do not upload this copyright pdf document to any other website. Breach of copyright may result in a criminal conviction.

This Acrobat document was generated by me, Colin Hinson, from a document held by me. I requested permission to publish this from Texas Instruments (twice) but received no reply. It is presented here (for free) and this pdf version of the document is my copyright in much the same way as a photograph would be. If you believe the document to be under other copyright, please contact me.

The document should have been downloaded from my website https://blunham.com/Radar, or any mirror site named on that site. If you downloaded it from elsewhere, please let me know (particularly if you were charged for it). You can contact me via my Genuki email page: https://www.genuki.org.uk/big/eng/YKS/various?recipient=colin

You may not copy the file for onward transmission of the data nor attempt to make monetary gain by the use of these files. If you want someone else to have a copy of the file, point them at the website. (https://blunham.com/Radar). **Please do not point them at the file itself as it may move or the site may be updated.**

It should be noted that most of the pages are identifiable as having been processed by me.

I put a lot of time into producing these files which is why you are met with this page when you open the file.

In order to generate this file, I need to scan the pages, split the double pages and remove any edge marks such as punch holes, clean up the pages, set the relevant pages to be all the same size and alignment. I then run Omnipage (OCR) to generate the searchable text and then generate the pdf file.

Hopefully after all that, I end up with a presentable file. If you find missing pages, pages in the wrong order, anything else wrong with the file or simply want to make a comment, please drop me a line (see above).

It is my hope that you find the file of use to you personally – I know that I would have liked to have found some of these files years ago – they would have saved me a lot of time !

Colin Hinson
In the village of Blunham, Bedfordshire.

TEXAS INSTRUMENTS

Improving Man's Effectiveness Through Electronics

Graphics Programming Language

Programmer's Guide

ORIGINAL ISSUE 1 MAY 1979
REVISED 1 JUNE 1979

Personal Computer Division

TEXAS INSTRUMENTS

GRAPHICS PROGRAMMING LANGUAGE

USER'S GUIDE

Personal Computer Division

June 1, 1979

April 30, 1979

Dear User:

The contents of this manual have been reviewed by members of the Personal Computer Division for clarity, correctness and completeness. We feel that the Graphics Programming Language User's Guide will give a true representation of the graphics language capabilities. If you find contradictions to any claims made in this manual, or have any suggestions or corrections, please let us know on the User's Response Sheet found in the back of this manual. Send your response to:

Texas Instruments Incorporated
Personal Computer Software Manager
RE: GPL Manual
P.O. Box 10508, M/S 5890
Lubbock, Texas 79408

Thank you.

TABLE OF CONTENTS

GRAPHICS PROGRAMMING LANGUAGE

TABLE OF CONTENTS
Page 2

TABLE OF CONTENTS

LIST OF FIGURES AND TABLES

LIST OF FIGURES AND TABLES

1.0 GRAPHICS PROGRAMMING LANGUAGE

The system software resident in the product consists of a monitor and a GPL (Graphics Programming Language) processor. It is the function of the monitor to insure that every time the system is turned on, a new cartridge is inserted, or an existing program terminates, that all memory and peripheral devices are initialized. The GPL processor is an interpreter optimized to execute GPL programs directly out of GROM. The GPL processor software is coded in TMS 9900 assembly language.

1.1 OVERVIEW

GPL is a programming language specially developed by Texas Instruments to provide the best possible tradeoff of code compaction, execution speed, and ease of program development for the target computer system. The GPL instruction set facilitates development of programs which make use of the unique features of the system chip set. It is byte oriented, and instructions typically have one or two operands. The addressing scheme is such that most instructions can access either standard microprocessor RAM, GROM, or the video scratchpad RAM address space easily.

byte values. The addressing modes are: immediate, direct, indirect, indexed, indexed indirect (with pre-indexing), and 'top of stack'. Source operands and destination addresses can be in the CPU, video RAM, or in GROM. Support for two stacks is available; a data stack and a subroutine return address stack (allowing arbitrary nesting of subroutines).

1.2 GPL INSTRUCTION SYNOPSIS

GPL has the following types of instructions:

*DATA TRANSFER	-single or double byte transfers; -block to block transfers -formatted block transfers
*ARITHMETIC	-add, subtract, multiply, divide, negate, absolute value
*LOGICAL	-and, or ,exclusive or, shifting
*CONDITION TESTS	-arithmetic and logical tests
*BRANCHING	-unconditional and conditional
*BIT MANIPULATION	-set, reset, and test
*SUBROUTINING	-call, return, parameter fetching
*STACK OPERATIONS	-push and pop
*MISCELLANEOUS	-random number generation, keyboard scan, coincidence detection pattern movement, sound control, TMS 9900 subroutine linking, I/O

1.3 GPL TIMING

The GPL interpreter contains an interrupt driven service routine which is tied to the video scan. Video symbols may be moved about the screen automatically; also sounds may be generated from a sequence table.

These are of the "set it and forget it" type of instructions which free up the control program to do concurrent decision and computational operations. The interrupt also controls a software real time clock.

Each system will have a clock byte reserved in the console ROM at location >000C to indicate the clock rate for that system. Peripherals may read this byte to adjust their timing interface to the CPU's clock combinations in different consoles. The high nybble contains the integer frequency in megahertz and the low nybble, the fractional frequency.

1.4 GPL ASSEMBLER

The assembler for GPL (GPLASM) is written in a mixture of FORTRAN and assembly language and is currently available for installation on 990/10 DS minicomputers. The assembler provides standard features such as creation of a list file, cross reference tables, and error flagging. A set of macros is included to help structure GPL programs; these include statements such as: REPEAT ... UNTIL and IF ... THEN ... ELSE. The output of the assembler is a 990 object module.

1.5 SOFTWARE MONITOR RECONFIGURATION

The monitor code is executed whenever a system restart is required. The system parameters and control values are initialized to default values. A default character set is loaded into the video pattern generator, making it immediately available to GPL programs. This pattern set consists of 64 ASCII characters, including the upper-case alphabet, digits, arithmetic symbols, and punctuation symbols.

The monitor is also responsible for determining the existing system configuration. The power-up monitor must poll

add-on I/O peripherals and the 'SOLID STATE SOFTWARE CARTRIDGE' to determine which program to execute.

The Home Computer system has been designed to be flexible and expandable. Each plug-in ROM or GROM may contain power-up procedures. These power-up procedures will all be executed allowing for expansion of the power-up routines. A power-up routine may also be replaced by another.

1.6 APPLICABLE DOCUMENTS

- System Monitor Specification
- TMS 9918 Video Display Processor Specification
- TMS 9919 Sound Generation Controller
- TMS 9900 Microprocessor Specification
- File Management Specification
- Home Computer System Memory, CRU, and Interrupt Mapping Specification

2.0 SUMMARY OF SYSTEM ORGANIZATION

The system, as supported by the interpreter, consists of a 9900 microprocessor with the following peripheral devices tied to it:

- a Sound Generation Controller Chip
- a Video Display Processor Chip
- one or more GROM devices
- at least one type of keypad entry device

The Sound chip interface is discussed in Appendix C. The GROM is described in System Memory Organization below. The following is a quick summary of the VDP organization. For more detailed information on any of the system hardware, refer to the appropriate document.

2.1 VDP ORGANIZATION

The VDP RAM contents determine what will appear on the screen. They contain several sub-blocks, each of which is described below. The base address of each sub-block is determined by the contents of the VDP control registers. Table 3.4 (page 3-14) shows the interpretation of the VDP control registers. Also shown are the most commonly used values for these registers. These values keep all the sub-blocks within the first 4K bytes of VDP RAM, and insure that none of the sub-blocks overlap each other.

2.1.1 PATTERNS

The active area of the screen is divided into a grid of 192 (vertical) by 256 (horizontal) pixels. These are clustered into

8 x 8 pixel groups called Patterns. Thus there are 24 x 32 pattern positions on the screen in the normal mode.

There are three sub-blocks of VDP RAM associated with displaying patterns on the screen:

- Pattern Name Table (768 bytes)- Each byte corresponds to a pattern position on the screen, and its value is the pattern number (0 thru 255) displayed at that location.

- Pattern Generator Sets (8 * 256 = 2048 bytes)- Each block of 8 bytes in the Pattern Generator Set defines a pattern (8 x 8 pixels); the first 8 bytes correspond to pattern number 0 (as called out in the Pattern Name Table), the last 8 to pattern number 255. Note that a pattern is not displayed on the screen until an entry in the Pattern Name Table calls for it. Also, a pattern can be displayed in multiple positions on the screen by setting several entries in the Pattern Name Table to the same pattern number.

- Pattern Color Table (32 bytes)- Each byte of the Color Table contains in its left nybble a foreground color (1's in the pattern) and in its right nybble a background color (0's in the pattern). The first byte describes colors for pattern numbers 0 thru 7, the next for numbers 8 thru 15, etc. See Table 2.1 (page 2-4) for color nybble assignments. Table 2.1.A(page 2-5) contains some of the best foreground/background combinations.

2.1.2. SPRITES

Sprites are objects that exist essentially in planes in front of the pattern plane. These objects can be moved on a pixel-by-pixel basis, providing for excellent animation capability. Up to 32 Sprites may be on the screen at any time; however, no more than 4 on a given horizontal pixel line are allowed (subsequent sprites on that line will not be displayed). Three sub-blocks of VDP RAM define the Sprites:

o Sprite Attribute Block (SAB) (4 * 32 = 128 bytes)- Each 4-byte entry in this block describes the position and color of each Sprite:

byte 1- y-position of Sprite (0 is top of screen-- >FF in SAL--must subtract 1 from desired starting position of sprite);

byte 2- x-position of Sprite (0 is left edge of screen);

byte 3- pointer to Sprite Descriptor Block entry;

byte 4- early clock and color nybble.

The pointers to Sprite Descriptor Block entries, when the recommended base addresses are chosen, range from >80 to >FF if no Sprite motion is used and from >80 to >EF if Sprite motion is used (each pointer points to a succeeding 8-byte block in the Sprite Descriptor Block). When size 1 sprites (32-byte) are chosen, the pointer value must be an even multiple of 4 (i.e. >80, >84, >88, etc.) and point to a 32-byte block in the Sprite Descriptor Block.

TABLE 2.1

COLOR NYBBLE ASSIGNMENTS

NYBBLE VALUE (>)	COLOR
0	Transparent
1	Black
2	Green 2
3	Green 1
4	Blue 2
5	Blue 1
6	Red 3
7	Cyan
8	Red 2
9	Red 1
A	Yellow 2
B	Yellow 1
C	Green 3
D	Magenta
E	Gray
F	White

When there is more than one shade of the same color, the lowest numbered color is the lightest and the highest numbered color is the darkest (e.g., Green 1 is the lightest, Green 2 is medium, and Green 3 is darkest.)

TABLE 2.1.A

BEST COLOR COMBINATIONS

BEST

Black on Light Green
Black on Light Blue
Black on Dark Red
Black on Cyan
Black on Light Red
Light Green on Medium Red
Light Green on Dark Red
Light Blue on White
Light Yellow on Magenta
White on Cyan
White on Light Red
White on Gray

SECOND BEST

Medium Red on Dark Red
Light Red on Medium Red
Dark Yellow on Medium Green
Light Yellow on Light Green
Light Yellow on Dark Blue
Light Yellow on Medium Red
Gray on Medium Green
Gray on Medium Red
Gray on Magenta
White on Black
White on Medium Red
White on Magenta
Dark Red on White

THIRD BEST

Light Green on Light Blue
Dark Blue on Black
Light Red on Black
Light Red on Dark Green
Dark Yellow on Black
Light Green on Black
Light Yellow on Black
Gray on Light Blue
White on Light Green
White on Light Blue
White on Dark Red
White on Dark Green

FOURTH BEST

Medium Green on Black
Medium Green on Dark Green
Light Green on Dark Blue
Light Green on Dark Green
Light Blue on Black
Light Blue on Dark Blue
Dark Red on Black
Cyan on Black
Cyan on Dark Blue
Cyan on Dark Green
Medium Red on Black
Light Red on Dark Red
Light Red on Magenta
Dark Yellow on Dark Green
Light Yellow on Dark Green
Light Yellow on Dark Red
Light Yellow on Medium Green
Dark Green on Black
Magenta on Black
Magenta on Dark Blue
Gray on Black
Gray on Dark Blue
Gray on Dark Red
Gray on Dark Green
White on Medium Green
White on Dark Blue

The MSB of byte 4 is set if you want the sprite to come in or go off smoothly on the left side of the screen. If this bit is not set, the sprite will come in or go off smoothly on the right side of the screen. The right nybble of this byte is the color nybble. A >D0 in the first byte of a 4-byte block in the Sprite Attribute Block will tell the system to disregard all following data in the Sprite Attribute Block. The >D0 indicates to the system that the preceding 4-byte block is the last sprite to be displayed on the screen.

• Sprite Descriptor Block (SDB) (32*32 = 1024 bytes if no sprite motion is used; 32*28 = 896 bytes if sprite motion is used since the Sprite Velocity Block begins at >780). The SDB is similar to the Pattern Generator Set area, each block of 8 bytes describes an 8 x 8 pixeled Sprite; alternately, each block of 32 bytes may describe a 16 x 16 pixel Sprite (when the size bit is set to a 1 in the VDP Command Register 1). When the size bit is set and 4 characters (32 bytes) are used to make the sprite, the first 8 bytes are the upper left character, the next 8 bytes are the lower left character, the next 8 bytes are the upper right character and the last 8 bytes are the lower right character. For example, if the bytes in a 32-byte Sprite Descriptor Block area are numbered 0 through

31, this is how the characters would be displayed in a sprite:

bytes 0-7	bytes 16-23
bytes 8-15	bytes 24-31

When the magnification bit in the VDP Command Register (1) is set, all sprites double their size, but keep the same pixel dimensions (8x8 or 16x16). Each pixel doubles its size. This expansion of size is to the right and down. Therefore, an unmagnified sprite on the screen will keep the same upper left corner position when the magnification bit is set.

- Sprite Velocity Block (SVB) (4*32 = 128 bytes)-Each 4-byte entry in this block assigns motion to the corresponding 4-byte entry in the Sprite Attribute Block:

  byte 1- y-velocity of Sprite (positive number means down, negative number means up)

  byte 2- x-velocity of Sprite (positive number means right, negative number means left)

bytes 3 and 4- reserved for system use (must be initialized to zero).

A velocity can range from 0 to >7F in the positive direction and from >FF to >80 in the negative direction. See Appendix B for more information on Automatic Sprite Motion.

2.1.3 VDP TEXT MODE AND MULTICOLOR MODE

The VDP Text Mode and Multicolor Mode as described in the VDP Specification are supported to the extent described in Appendix G. The programmer may use Text, Multicolor and normal mode in the same program if he chooses. The programmer should be aware, however, that a new mapping of VDP RAM into a screen image is created for each mode.

2.2 SYSTEM MEMORY ORGANIZATION

There are three segments of memory associated with the basic system:

- CPU RAM: 256 bytes of high speeed Read/Write random access memory (Figure 2.2, page 2-10, 2-11). To access CPU RAM in Asssembly Language, a bias of >8300 is added to the address.
- VDP RAM: 4K, 8K or 16K of Read/Write random access memory (Figure 2.3, page 2-12); as discussed earlier, this memory is segmented into subblocks whose data map into a screen image; whatever memory is left over is available for GPL programming use.
- GROM: Increments of 6K bytes located at 8K-byte boundaries; this is special, medium speed, ROM; it typically contains GPL programs and data.

Certain areas of the three segments are dedicated for special use by the VDP hardware or the interpreter software. See Figure 2.2 (page 2-10, 2-11) for CPU RAM segments dedicated to the interpreter. See Figure 2.3 (page 2-12) for VDP RAM dedicated for use by the VDP chip (note the base addresses of the sub-blocks assume that the recommended values are loaded in the VDP Registers). Also shown in Figure 2.3 (page 2-12) is a sub-block that is used by the Interpreter software for auto-motion of sprites. If auto-motion is not to be used in a GPL program, this memory space is freed up for other use. See Appendix B for details on Auto-Sprite motion. GROMs have a format protocol which they must adhere to in order to maintain system compatibility. See the System Monitor Specification for details.

FIGURE 2.2

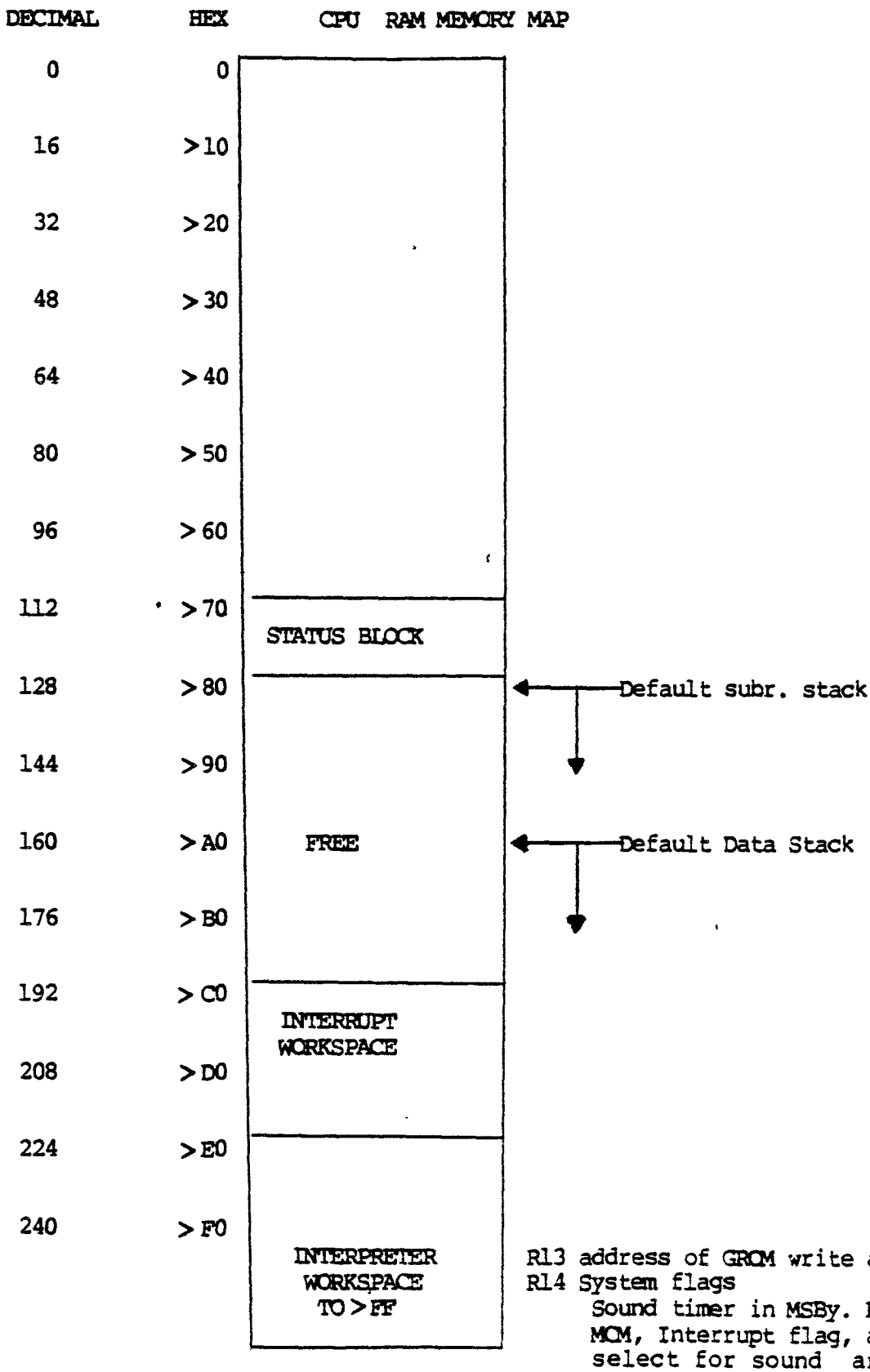

DECIMAL
HEX
CPU RAM MEMORY MAP
0
0
16
>10
32
>20
48
>30
64
>40
80
>50
96
>60
112
>70
STATUS BLOCK
128
>80
Default subr. stack
144
>90
160
>A0
FREE
Default Data Stack
176
>B0
192
>C0
INTERRUPT WORKSPACE
208
>D0
224
>E0
240
>F0
INTERPRETER WORKSPACE TO >FF
R13 address of GROM write address
R14 System flags
Sound timer in MSBy. Flags for MCM, Interrupt flag, and GROM/VDP select for sound are in LSBy.
R15 address of VDP write address

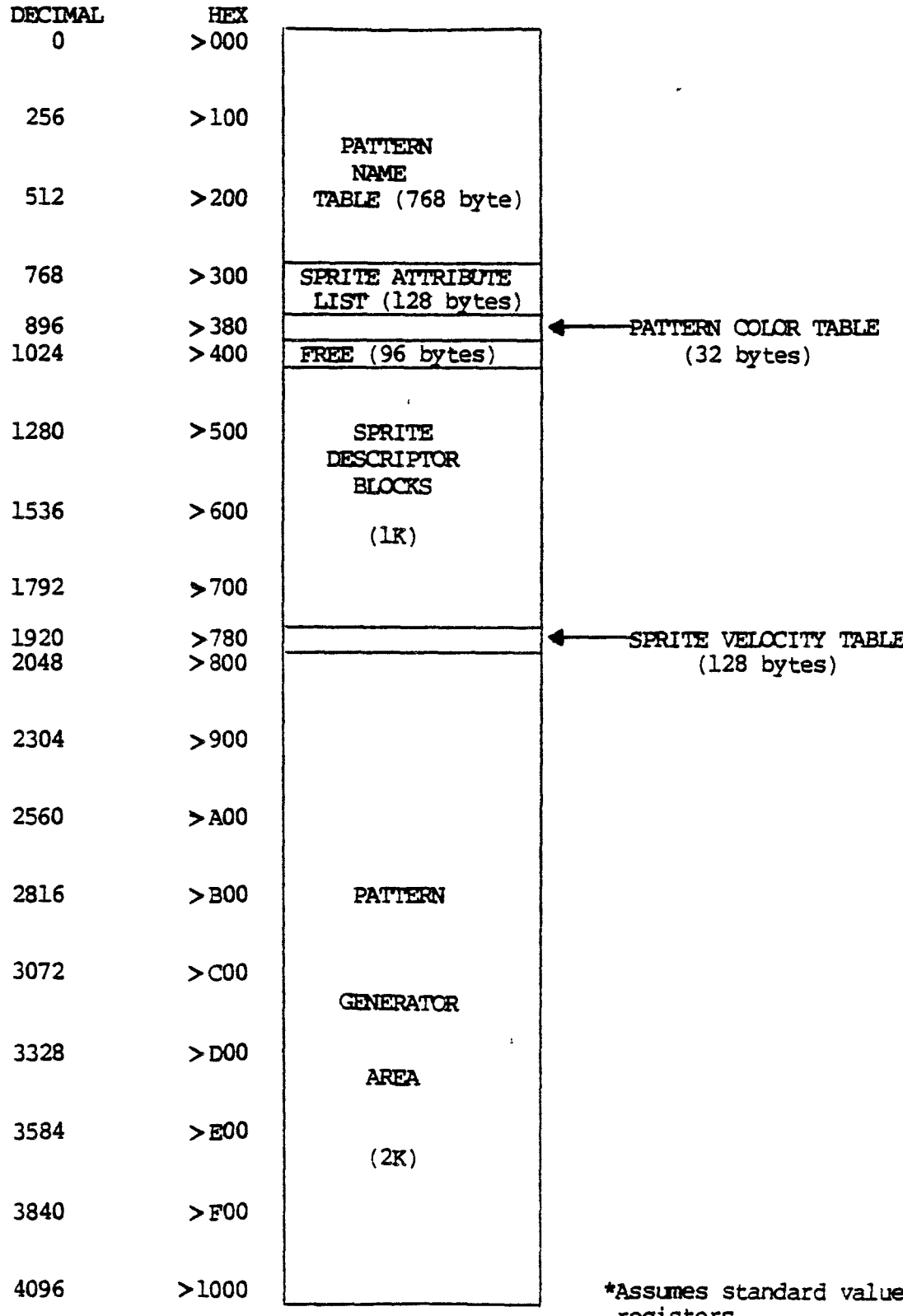
FIGURE 2.2 (Cont.)

INTERRUPT WORKSPACE

>C0: Random Seed
>C2- C9: Remote handset debounce
>CA: Console Keyboard debounce
>CC: Sound list pointer
>CE: Number of sound bytes
>D0: Search pointers for
>D2: GROM and ROM searches
>D4: One byte - stores last VDP (1)
>D6: Screen timeout counter
>D8: Save return address for scan routine
>DA: Save player number in scan routine

R13..R15: Return linkage for interrupts

FIGURE 2.3
VDP RAM MEMORY MAP *



*Assumes standard values in VDP registers.

3.0 GPL INSTRUCTIONS

The Graphics Programming Language is similar to an Assembly Language in many respects. Commands are followed by operands which specify addresses and immediate values. The completed program is run through an assembler which generates, for each instruction, the opcode followed by an encoding of the operands. Many instructions can operate on single or double byte values. In the instruction descriptions of Section 4, this is indicated by a "D" prefix on the mnemonic; for example, the single-byte to single-byte "add" instruction is an ADD, while the double-to-double-byte add is a DADD.

The extent of graphics support is through the following:

- Almost all instructions can modify locations in VDP RAM easily; this can cause a change in the screen image;
- Locations in the Pattern Name Table can be addressed by specifying an X pointer and a Y pointer;
- Special instructions allow the reading and writing of large blocks of VDP RAM quickly;
- Automatic motion of Sprites can be initiated; after enabling auto-motion with a GPL instruction, motion of sprites is automatically controlled until stopped by another GPL command.

GPL instructions fall into several classes:

- Data Transfer
- Arithmetic
- Logical
- Condition Tests

- Branching
- Bit Manipulation
- Subroutining
- Stack Operations
- Miscellaneous

3.1 ADDRESSING MEMORY

The addressing modes of most instructions allow operands to reside anywhere in VDP RAM or CPU RAM. This is called "Global Addressing". Each address above CPU location >7F requires two bytes to specify its address.

The next section is a description of all GPL instructions. The mnemonics used for specifying the operand types required for a given instruction are always of the following types:
GS (Global Source), GD (Global Destination), IMM (immediate value), LABEL (GPL label). These are each described more fully below.

• 3.1.1.IMM

An immediate field can be a numeric constant in decimal, hexadecimal or binary format. Depending upon the context, values can be single or double byte values. In DATA statements double-byte values must be preceded by a pound sign (#). The # sign is optional for double-byte values in branches, move statements, and double instructions.

A symbol can be used in an IMM field if it is equated to an immediate value using the assembler EQU directive

(commonly used locations in CPU RAM and VDP RAM are often assigned symbolic equates to improve program clarity). If it is a label in the GPL program, it is a double-byte value unless used in a single byte operation. In this case the least significant byte is used.

To illustrate the possibilities:

```
FIVE EQU 5      (now the symbol FIVE can be used
                 wherever IMM is called for;)
51              ..decimal 51;
>33 or 033      ..hexadecimal 33;
&110011         ..binary 110011;
#LOOP           ..(if LOOP is a label in the GPL program)
```

The ASCII equivalent of characters can also be used for IMM fields. The character(s) should be enclosed between colons; e.g.

```
:A:     is equivalent to >41
:2A:    is equivalent to >3241
```

The FMT instruction, to be discussed later, as well as the assembler directive DATA (in Appendix A) can use IMM fields of arbitrary length (e.g., :ABCD1234:). Instructions that require double-byte IMM operands begin with a "D" (e.g., DADD = Double Add) as opposed to instructions that do not (e.g., ADD = Add). The instructions D or DIV, DEC, and DECT require single-byte IMM operands; while DD or DDIV, DDEC, and DDECT require double-byte IMM operands.

• 3.1.2 GS (GLOBAL SOURCE)

Unless otherwise specified for a given instruction, a Global Source operand can be an immediate value (i.e. anything that fulfills requirements for IMM), or an address with any combination of the following features in effect:

1) Select CPU RAM or VDP RAM (select ROM in a MOVE statement only);
2) Select direct or indirect addressing;
3) Select indexing or not.

There are two special mnemonics that can be used wherever GS is called for: POP and TOP. POP pops the top value off the data stack and uses this data as an operand. TOP uses the data pointed to by the data stack pointer, but it does not actually pop the data off the stack. POP and TOP should not be used in double-byte instructions. An example of the use of POP and TOP is:

```
ADD POP,TOP
```

This instruction is equivalent to the sequence:

```
ST  *DATSTK,@TEMP
DEC @DATSTK
ADD @TEMP,*DATSTK
```

The next section discusses the Data Stack more fully.

• 3.1.3 GD (GLOBAL DESTINATION)

Global Destination is exactly the same as Global Source except that immediate values are not allowed.

• 3.1.4 LABEL

A LABEL field refers to a symbol which has been used in front of a GPL instruction, or a symbol that has been equated (using EQU) to an IMM. A LABEL always generates a 2-byte immediate value (16 bits). LABEL fields are called for in Branch instructions and Subroutine call instructions. A long branch (B) insturction, a CALL subroutine instruction, and a GS or GD of ROM (#LABEL) in a MOVE statement may use labels contained anywhere in the program, but short branch instructions (BR, BS, or $IF-GOTO) must use labels contained in the same 6K GROM segment as the instruction. A special LABEL, "$", is used to represent the current location; (e.g. "B $" will cause the GPL program to loop forever).

LABELs can have, in addition to the symbol, an expression of the form (symbol)+IMM or (symbol)-IMM; for example,

```
        BR    #LAB1+3
or:     BR     LAB3-1
```

• 3.1.5 ADDRESSING MODES

Table 3.1 (page 3-7) shows the formats for the various mode combinations with an example. IMM specifies a numeric constant. If an "at sign" (@) precedes an IMM value, it specifies the contents stored at location IMM in CPU RAM.

If a star (*) precedes an IMM value, it specifies indirect addressing through location IMM in CPU RAM. For example,

```
A    EQU   >02
B    EQU   >04
     ST    >60, @A
     ST    *A, @B
```

will take the data stored in CPU location >60 and store it in CPU location >04.

A double byte value in CPU RAM can be used as an index to a specified location. For example,

```
A       EQU   >02
B       EQU   >04
INDEX   EQU   >06
        DST   >000A, @INDEX
        ST    @A(INDEX), @B
```

will store the contents of CPU location >0C in CPU >04.

You would obtain the same results with:

```
        ST    @A(>06), @B
```

Notice that indexing takes the IMM value in the parentheses and adds the double-byte value stored in location IMM in CPU RAM to the location. You do not use the @ sign when indexing. In the case of VDP RAM indexing, the inner parentheses contain the index value.

TABLE 3.1
SYNTAX FOR GS, GD

	SYNTAX	MEANING	EXAMPLE
C P U R A M	IMM	IMMEDIATE[1]	#LAB1, 21
	@IMM	CONTENTS OF CPU RAM	@LOC, @30
	*IMM	INDIRECT TO & THRU CPU RAM (IMM MAY BE 0-6K BUT INDIRECTS TO 0-255)	*LOC
	@IMM(IMM)	INDEXED BY A DOUBLE-BYTE VALUE (BOTH IMM'S IN (0-255))	@LOC1(LOC2)
	*IMM(IMM)	INDEXED INDIRECT[2] (INDEX MUST BE 0-255)	*L(LOC)
V D P R A M	RAM(IMM)	VDP RAM DIRECT	RAM(LOC1)
	RAM(@IMM)	VDP RAM INDIRECT[4] (THRU THE CPU RAM; IMM IN (0-255))	RAM(@LOC)
	RAM(IMM_1(IMM_2))	VDP RAM INDEXED BY CPU RAM (IMM_2 IN (0-255))	RAM(LOC(LOC2))
	RAM(@IMM(IMM_2))	VDP RAM INDIRECT INDEXED[2,4] (IMM_2 IN (0-255))	RAM(@LOC(LOC2))
	DISPLAY(X=IMM,Y=IMM)	VDP RAM'S PATTERN NAME TABLE[3]	
	CHAR(IMM)	VDP RAM'S PATTERN GENERATOR[3]	
	TABLE(IMM)	VDP RAM'S PATTERN COLOR TABLE[3]	
	SPRITE(IMM)	VDP RAM'S SPRITE ATTRIBUTE LIST[3]	
	FIGURE(IMM)	VDP RAM'S SPRITE DESCRIPTOR BLOCK[3]	
	VEL(IMM)	VDP RAM'S SPRITE VELOCITY TABLE[3]	
	VDP(IMM)	VDP REGISTER	

1 GS ONLY
2 PRE-INDEXING
3 ASSUMES RECOMMENDED BASE ADDRESS VALUES ARE USED, SEE FIGURE 3.4 AND APPENDIX A, "BASE" DIRECTIVE
4 THE INDIRECT ADDRESS IS IN CPU

3.2 FORMAT TYPES

In the next section you will see that instructions get assembled into several different variations of formats. Each instruction has a "format type" number. Table 3.2 (page 3-9) shows all the possible formats, listed by format type. Also shown is the op-code range for each of the format types. The X's in the formats represent bits that may be turned on or off according to the opcode for the instruction. Each letter in the format other than X is described on page 3-10 along with the five forms of GS and GD.

TABLE 3.2

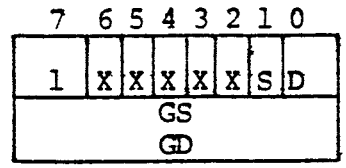
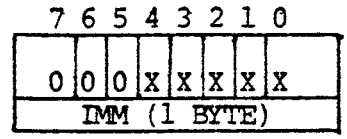
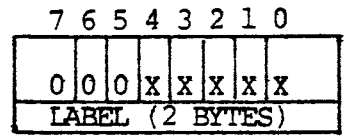
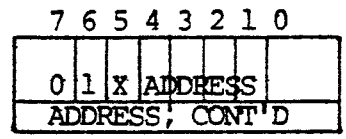
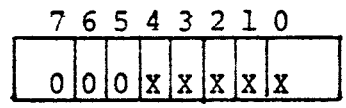
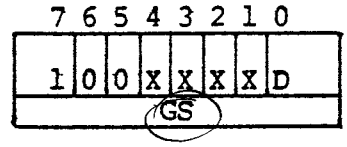
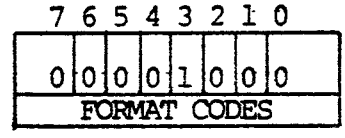
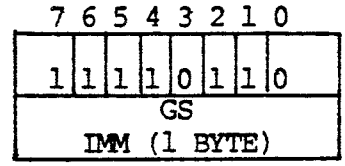
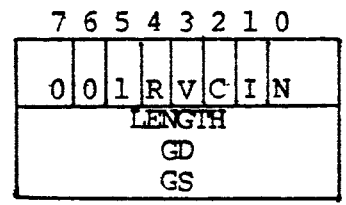
FORMAT TYPE	FORMAT	OP CODES
1		OAX,OBX,OCX ODX,OEX
2		OX, 1X
3		OX,1X
4		4X,5X,6X,7X
5		OX, 1X
6		8X,9X
7		08
8		F6
9		2X, 3X

TABLE 3.2
(Cont.)

D = ⇨ 0 = SINGLE BYTE OPERATION
1 = DOUBLE BYTE OPERATION

S = ⇨ 0 = GS IS NOT IMMEDIATE
1 = GS IS IMMEDIATE (1 OR 2 BYTES DEPENDING ON D)

0 0 1 R V C I N

R = ⇨ 0 = GD is ROM
1 = GD is not ROM

V = ⇨ 0 = GD is not a VDP register
1 = GD is a VDP register

C = ⇨ 0 = GS is not RAM
1 = GS is RAM

I = ⇨ 0 = GS is not ROM addressed by CPU
1 = GS is ROM indexed or addressed by a variable in CPU RAM

N = ⇨ 0 = Number of bytes moved is not immediate value
1 = Number of bytes moved is immediate value

GS, GD HAVE 5 FORMS:

I | 0 ADDRESSES | = DIRECT ADDRESSING TO FIRST 128 BYTES OF CPU RAM;

II | 1 0 V I ADDRESS
ADDRESS (CONT'D) | ; V=1 SELECTS VDP RAM; 0=CPU RAM
I=1 SELECTS INDIRECT;0=DIRECT

III | 1 1 V I ADDRESS
ADDRESS (CONT'D) | LIKE ABOVE, EXCEPT AN INDEX VALUE IS ADDED TO THE ADDRESS IN CPU RAM

IV | 10 VI 1111
ADDRESS
ADDRESS | LIKE II WITH EXTENDED RANGE 0-65K

V | 11 VI 1111
ADDRESS
ADDRESS
INDEX | LIKE ABOVE III EXTENDED ADDRESS AND INDEXED

3.3 RUNNING GPL PROGRAMS

The system Monitor performs the startup of a GPL program. See the Monitor Specification for details on power-up and restart sequences. It will suffice here to know the state of all RAM and Register locations upon beginning program execution.

- An >60 is written to the VDP Command Register, which makes the Start bit a 0; this turns the TV screen to the background color.
- A default character set has been loaded into Pattern Generator Sets 4 thru 11, corresponding to ASCII symbols >20 thru >5F; see Table 3.3. The Pattern Color Table is initialized and all other locations are zero.
- Several locations in the "Status Block" in the CPU RAM have been initialized to pre-defined values; these locations are explained in Appendix H under System Initialization.

The programmer has the responsibility of initializing the values of the VDP Registers if default values are not to be used. The values in Table 3.4 are the default values. For a system without RAM expansion, these VDP block bases are suggested.

VDP(0) is a read-only register which is read in the VDP status byte in CPU RAM >7B. The MSB is a frame interrupt bit. Bit 6 is the fifth sprite bit and is set any time there are five sprites on a line. Bit 5 is a sprite coincidence flag and is set any time there is sprite coincidence. The last five bits are used for the number of the fifth sprite on a line.

TABLE 3.3

DEFAULT CHARACTER SET

PATTERN #							
>20	BLANK	>30	0	>40	@	>50	P
	!		1		A		G
	"		2		B		R
	#		3		C		S
>24	$	>34	4	>44	D	>54	T
	%		5		E		U
	&		6		F		V
	'		7		G		W
>28	(	>38	8	>48	H	>58	X
	)		9		I		Y
	*		:		J		Z
	+		;		K		[
>2C	,	>3C	<	>4C	L	>5C	\
	-		=		M		]
	.		>		N		^
	/		?		O		_

VDP(1) is the command register. Bit 7 is set if there is a 16K chip in the system. This bit should always be reset by the programmer. The interpreter will set the bit if there is 16K. Bit 6 turns the screen on when set. Bit 5 is the interrupt enable bit. The Bit 4 tells the VDP to use text mode when it is set and Bit 3 tells the VDP multicolor mode when it is set. Bits 3 and 4 may not be set at the same time. Bits 1 and 0 tell the system double-size and magnified sprites, respectively, when set. Bit 2 must always be reset.

The value in VDP(2) can range from 0 to 15. The Pattern Name Table will begin at location VDP(2) * 256.

The value is VDP(3) can range from 0 to>FF. The Pattern Color Table will begin at location VDP(3) * 64.

The value in VDP(4) can range from 0 - 7. The Pattern Generator Table will begin at location VDP(4) * >800.

The value in VDP(5) can range from 0 - >7F. The Sprite Attribute List will begin at location VDP(5) * 128.

The value of VDP(6) can range from 0 - 7. The Sprite Descriptor Block will begin at location (VDP(6) * >800) + >400.

The value of VDP(7) contains the only way of giving foreground and background colors to Text mode. The most significant nybble is the foreground color, and the least significant nybble is the background color and also the border color in any mode.

The value of VDP(1) will probably be the register most often changed in a program. Table 3.4.A (page 3-15) lists some of the most common values used in this command register and what they represent.

TABLE 3.4

VDP REGISTERS

EGISTER NUMBER	FORMAT								MEANING	RECOMMENDED VALUE
	7	6	5	4	3	2	1	0		
0	FR INT	5TH SP	SC		FIFTH SPRITE #				Frame interrupt, Five sprites on a line, Sprite Coincidence, Number of the fifth sprite.	Read only register (CPU RAM >7B)
1	4K 16K	ST INT ENB	TXT	MCM	X	X	SIZE	MAG	Command Reg. bits turn on/off 4K or 16K, screen on/off, interrupt text mode, MC mode Sprite size & mag	>60-sets VDP for 4K memory, turns screen on, enable VDP interrupt puts VDP in normal pattern mode with size 0, mag 0 sprites
2	X	X	X	X	PNT BASE				Multiple of 256 to start PNT at	0-puts Pattern Name Table at 0.
3	PATTERN COLOR TABLE BASE								Multiple of 64 to to start col. table	>OE-puts color table at >380
4	X	X	X	X	PAT GEN BASE				Multiple of >800 to start pat. gen.	>1-puts pattern generator area >at 800
5	X	SPRITE ATTRIB. LIST BASE							Multiple of 128 to start SAL,	>06-puts Sprite Attribute List at >300
6	X	X	X	X	SDB BASE				Multiple of >800 then add >400 to start SDB	>0-puts Sprite Descriptor Blocks at >400
7	TEXT COLOR				BACKDROP COLOR				Colors for text mode, backdrop color	>F7-make text color white, backdrop cyan

TABLE 3.4.A

COMMAND REGISTER VALUES

VALUE	MEANING
>61	4K memory, screen on, interrupt on, single-sized magnified sprites.
>62	4K memory, screen on, interrupt on, double-sized unmagnified sprites
>63	4K memory, screen on, interrupt on, double-sized magnified sprites.
>20	4K memory screen off (viewer sees a blank screen the color of the border).
>70	4K memory, screen on, interrupt on, text mode (40 x 24 character screen)
>68	4K memory, screen on, interrupt on, multicolor mode, single sized unmagnified sprites.
>69	4K memory, screen on, interrupt on, multicolor mode, single-sized magnified sprites.
>6A	4K memory, screen on, interrupt on, multicolor mode, double-sized unmagnified sprites.
>6B	4K memory, screen on, interrupt on, multicolor mode, double-sized magnified sprites.

The actual mechanics of writing and running a GPL program are described in Appendix A. This describes the format of instructions that the GPL assembler will accept.

The interpreter and the GPL program communicate with each other through a dedicated location in CPU RAM, called the Status Block. Table 3.5 (page 3-20) shows the fixed locations of each Status Block variable.

3.3.1 THE STATUS BLOCK

If any of the bytes in the Status Block are to be accessed from a GPL program, it is recommended that the symbols in Table 3.5 (page 3-20) be equated to the proper values as shown at the beginning of the GPL program. The symbol can then be used as an instruction operand.

The following is a discussion of each of the Status Block bytes:

- MAXMEM - Highest available VDP memory address. For a 4K system this would be >0FFF.

- DATSTK- Stack pointer for data; initialized to >9F by the Monitor, the pointer always points to the last value pushed on the data stack. The data stack is a pre-incremented, byte-oriented stack, and grows to increasing values in CPU RAM. If the user wishes, he can change the location of the stack by doing an ST into DATSTK (e.g. ST >92,@DATSTK). PUSH and POP affect the pointer value, as well as the operand POP.

- SUBSTK- Stack pointer for subroutine return addresses; initialized to >7E by the Monitor, the pointer always points to the last address pushed onto the stack. Addresses are automatically pushed onto the stack by the CALL instruction, and popped off by the RTN and RTNC instructions. As with DATSTK, the user can change the default address of the stack. The user should be careful when changing this stack pointer. SUBSTK should only be initialized with even numbers if it is changed. The MOVE and SCAN instructions use one level of subroutine stack.

- KEYBOARD, KEY,JOYY,JOYX- These locations are used for handset, joystick and keyboard interfaces. KEYBRD is the keyboard number, KEY is the returned keycode, JOYY and JOYX are the returned joystick parameters. See the SCAN instruction description for more details. Also see Appendix D. These values are initialized to 0 by the Monitor.

- RANDOM- This location is loaded with a random number when the RAND instruction is executed. It is initialized to a random number generated by the Monitor.

- TIMER- When the VDP Frame interrupt is enabled , this byte gets incremented by one every 1/60 second. By clearing it with a CLR and then using the loop

```
LOOP CEQ  (delay),@TIMER
BR LOOP
```

a fixed delay in the GPL program can be implemented.

- MOTION- This location, when set to a non-zero value by the programmer, represents the number of Sprites that are included in auto-motion. For example, if it contains a two, Sprites 0 and 1 will be put into auto-motion. See Appendix B for details on Sprite auto-motion.

- VDPSTT- This location is a copy of the VDP Status register. It is updated every frame interrupt (when frame interrupts are disabled, VDPSTT is not updated).

- STATUS- This byte automatically gets loaded with bits as a result of many instructions. It contains bits representing equality, arithmetic greater than, logical greater than, carry and overflow. See Section 3.4 for details.

- CB, YPT, XPT- These bytes, in conjunction with one another, provide a method for writing information out to the VDP Pattern Name Table. When the CB location is used as a source operand in an instruction, it is first loaded with the value of the Pattern Name Table specified by XPT and YPT. This assumes that the Pattern Name Table base address is 0 and the absolute VDP RAM address is calculated by 32*YPT+XPT. This provides a convenient method for reading information off of the screen. If CB is ever found to have been modified by an instruction, the new value of CB is

written to the Pattern Name Table location specified by XPT and YPT.

Some examples:

```
DST #0302,YPT
ST CB,@TEMP  ..causes TEMP to get loaded with the
                byte from location XPT=2,YPT=3;
ST  @CHR1, CB ..causes whatever is in CHR1 to be
                written to the screen at the
                location corresponding to the
                current values of XPT and
                YPT.
```

Multicolor mode uses YPT and XPT to do mapping automatically in range YPT = 0 to 47, XPT = 0 to 63. CB, XPT, and YPT are predefined symbols and can be used with or without @ sign in front of them.

TABLE 3.5

STATUS BLOCK

RECOMMENDED SYMBOL	ADDRESS IN CPU RAM (>)	INITIALIZED TO BY MONITOR
MAXMEM	70, 71	MAXIMUM VDP MEMORY ADDRESS
DATSTK	72	>9F
SUBSTK	73	>7E
KEYBRD	74	0
KEY	75	0
JOYY	76	0
JOYX	77	0
RANDOM	78	0
TIMER	79	0
MOTION	7A	0
VDPSTT	7B	0
STATUS	7C	0
CB	7D	0
YPT	7E	0
XPT	7F	0

3.4 THE STATUS BYTE

The byte in the STATUS BLOCK called STATUS is equivalent to the status register found in many microprocessors. 5 bits in the byte indicate the result of operations. The format of the STATUS byte is:

	H	GT	COND	CARRY	OVF	0	0	0
bit	7	6	5	4	3	2	1	0

The COND bit is most important, since the BR (Branch on Reset) and BS (Branch on Set) instructions use this bit to decide whether to branch or not. Many operations affect all the bits, especially single and double operand arithmetic/logical instructions. Instructions have been provided which transfer one of the other bits into the COND bit; this makes it easy to conditionally branch based on the results of an operation (See instructions H, GT, CARRY, OVF). For example, to branch to the LABEL "BR1" if the CARRY bit or the OVF bit is set, the following sequence can be used:

```
CARRY
BS  #BR1
or $IF .CARRY. GOTO BR1
OVF
BS  #BR1
or  $IF .OVF. GOTO BR2
```

In the instruction descriptions in the following section, the STATUS bits affected for each instruction are shown boxed in. Other STATUS bits are not affected at all. Note that some instructions like the branches always reset the COND bit.

4.0 INSTRUCTION DESCRIPTIONS

The following pages are a description of each Graphics Language instrucion.

All instruction descriptions tell how the status byte is affected and give execution results. The symbol := represents "takes the value of."

4.1.1 TEST LOGICAL HIGH BIT

Syntax definition: H

Example: LAB1 H TEST THE LOGICAL HIGH BIT

Definition: Set/reset condition bit to the logical high status bit value

Status bits affected: / H / GT / cond / carry / OVF /

Execution results: COND := H

Application notes: Use the H instruction to detect whether the logical high status bit was set as a result of the previous instruction as a prelude to a conditional branch (BR or BS)
For example:

```
H
BS LAB1
or $IF .H. GOTO LAB1
```

causes a Branch to LABEL "LAB1" if the logical high bit has been set.

Op Code: > 09

Format Type: 5

4.1.2 TEST ARITHMETIC GREATER THAN BIT GT

Syntax definition: GT

Example: LABl GT TEST THE ARITHMETIC GT BIT.

Definition: Set/reset condition bit to the arithmetic greater than status bit value.

Status bits affected: / H / GT / $\overline{\text{cond}}$ / carry / OVF/

Execution results: COND := GT

Application notes: Use the GT instruction to detect whether the Arithmetic greater than status bit was set as a result of the previous instruction as a prelude to a conditional branch (BR or BS)

Op Code: >0A

Format type: 5

4.1.3 TEST CARRY BIT

Syntax definition: CARRY

Example: LAB1 CARRY TEST THE CARRY BIT

Definition: Set/reset condition bit to the carry status bit value

Status bits affected: / H / GT / cond / carry / OVF /

Execution results: COND := CARRY

Application notes: Use the CARRY instruction to detect whether there was a carry out of the most significant bit of a byte or word as a result of the previous instruction as a prelude to a conditional branch (BR or BS)

Op Code: >0C

Format type 05

4.1.4 TEST OVERFLOW BIT

OVF

Syntax definition: OVF

Example: LAB1 OVF TEST THE OVERFLOW BIT

Definition: Set/reset condition bit to the overflow status bit value.

Status bits affected: / H / GT / cond / carry / OVF /

Execution results: COND := OVF

Application notes: Use the OVF instruction to detect whether an arithmetic overflow (the result is too large or too small to be correctly represented in two's complement representation) has occurred as a prelude to a conditional branch (BR or BS).

Op Code: >0D

Format type: 05

4.1.5 COMPARE EQUAL

CEQ
DCEQ

Syntax definition:
```
CEQ    GS,GD
DCEQ   GS,GD
```

Example:
```
LAB1    CEQ  48,@A      COMPARE (A) TO 48 AND
                        SET CONDITION BIT
                        ON EQUAL
```
OR
```
LAB1 $IF @A .EQ. 48 THEN
```

Definition: Compare the GD to the GS and set the condition bit depending on the result.

Status bits affected: / H / GT / cond / carry / OVF/

Execution results: (GD) ? = (GS)
COND: = set, if true
COND: = reset, if false

Application Notes: Use the CEQ instruction to compare the GD to the GS and set the condition bit if they are equal. This is used as a prelude to a conditional branch (BR or BS). The effect on the status bits is as if GS is subtracted from GD and the result compared to zero.

Op Code: >D4

Format type: 1

4.1.6 COMPARE LOGICAL HIGH

CH
DCH

Syntax definition: CH GS,GD
DCH GS,GD

Example:
```
.LABl    CH     @A,@B        COMPARE (B) TO (A)
           or                 AND IF (B) IS LOGICALLY
                              HIGHER THAN (A) SET THE
                              CONDITION BIT
         LABl $IF @B .H. @A THEN
```

Definition: Compare the GD to the GS and set the condition bit if the GD is logically higher than the GS

Status bits affected: / H / GT / cond / carry / OVF /

Execution results: COND := (GD) H (GS)

Application Notes: Use the CH instruction to do the comparison GD.H.GS and set the condition bit if the relation is true. Use as a prelude to a conditional branch (BR or BS).

Op Code: >C4

Format type: 1

4.1.7 COMPARE LOGICAL HIGH OR EQUAL

Syntax definition:
```
CHE    GS,GD
DCHE   GS,GD
```

Example:
```
LABl    CHE     20,@VALUE     COMPARE (VALUE) TO 20
                              & SET CONDITION BIT IF
                or            (VALUE) IS LOGICALLY
                              HIGHER THAN OR EQUAL TO 20
        $IF @VALUE .HE. 20 THEN
```

Definition: Compare the GD to the GS and set the condition bit if the GD is logically higher than or equal to the GD

Status bit affected: / H / GT / cond / carry / OVF /

Execution Result: COND := (GD) ?HE (GS)

Application Notes: Use the CHE instruction to do the comparison GD.HE.GS and set the condition bit if the relation is true. Use as a prelude to a conditional branch (BR or BS)

Op Code: >C8

Format type: 1

4.1.8 COMPARE GREATER THAN '

CGT
DCGT

Syntax definition: CGT GS,GD
DCGT GS,GD

Example: LABEL CGT @A,NEW COMPARE NEW TO (A) AND SET CONDITION BIT IF NEW IS GREATER THAN (A)

OR

LABEL $IF @NEW .GT. @A THEN

Definition: Compare the GD to the GS and set the condition bit if GD is greater than (arithmetically) the GS.

Status bits affected: / H / GT / cond / carry / OVF /

Execution results: COND := (GD) ? GT (GS)

Application Notes: Use the CGT instruction to do the comparison GD.GT.GS and set the condition bit if the relation is true. Use as a prelude to a conditional branch (BR or BS)

Op Code: >CC

Format type: 1

4.1.9 COMPARE GREATER THAN OR EQUAL

CGE
DCGE

Syntax definition:
```
CGE    GS,GD
DCGE   GS,GD
```

Example:
```
LAB1    CGE     82,@B     COMPARE (B) TO 82 AND SET
                          CONDITION BIT IF (B) IS
                or        GREATER THAN OR EQUAL TO 82

        LAB1 $IF @B .GE. 82 THEN
```

Definition: Compare the GD to the GS and set the condition bit if GD is greater than or equal to the GS

Status bits affected: / H / GT / cond / carry / OVF /

Execution results: COND := (GD) GE (GS)

Application Notes: Use the CGE instruction to do the comparison GD GS and set the condition bit if the relation is true as a prelude to a conditional branch (BR or BS)

Op Code: >D0

Format type: 1

4.1.10 COMPARE LOGICAL

CLOG
DCLOG

Syntax definition:
```
CLOG    GS,GD
DCLOG   GS,GD
```

Example:
```
LABEL  CLOG  >86,@VALUE     SET CONDITION IF RESULT
                           OF >86.AND.(VALUE)
                           IS ZERO
```

Definition: Perform the bit by bit logical AND operation between GS and GD and set the COND bit if the result is 0.

Status bits affected: / H / GT / cond / carry / OVF /

Execution Results: COND := (GS) AND (GD) ?= 0

Application Notes: Use the CLOG instruction to set COND if GD and GS have no 1's in same positions.

Use as a prelude to a conditional branch (BR or BS)

Op Code: >D8

Format type: 1

4.1.11 COMPARE ZERO

CZ
DCZ

Syntax definition: CZ GD
DCZ GD

Example: LAB1 CZ @VALUE SET CONDITION BIT IF (VALUE)IS EQUAL TO ZERO

Definition: Compare the GD to zero and set the condition bit accordingly.

Status bit affected: / H / GT / cond / carry / OVF /

Execution Results: COND =: (GD) ? = 0

Application Notes: Use the CZ instruction to do the comparison GD = 0 and set the condition bit if the relation is true. Use as a prelude to a condtional branch (BR or BS)

Op Code: >8E

Format type: 6

4.2.1 BRANCH ON SET

Syntax definition: BS LABEL

Example: LABEL BS HERE BRANCH TO ADDRESS OF HERE IF CONDITION IS SET

Definition: Branch to address of the LABEL operand if the COND bit is set. After branching the condition bit is reset.

Status bit affected: / H / GT / cond / carry / OVF /

Execution results: IF (COND.EQ.set) THEN (PC):= LABEL

Application Notes: Use the BS instruction to branch to another portion of the program depending on whether the condition bit is set. For example if the previous instruction was a SUB that resulted in a zero result; the instruction,

```
BS   ZERO
```

program execution commencing at the instruction at label "ZERO".
NOTE: The LABEL must reside in the same 6K GROM segment as the BS instruction.

Op Code: > 60

Format type: 4

4.2.2 BRANCH ON RESET

Syntax definition: BR LABEL

Example: LAB1 BR HERE BRANCH TO ADDRESS "HERE" IF CONDITION IS RESET

Definition: Branch to address of the label operand if the condition bit is reset. After execution the condition bit is reset.

Status bits affected: / H / GT / $\overline{\text{cond}}$ / carry / OVF /

Execution results: IF (COND.EQ.0) THEN (PC):= LABEL

Application notes: Use the BR instruction to branch to another portion of the program depending on whether the condition bit is reset. For example if the previous instruction was an ADD that resulted in a non-zero result the instruction,

```
BR   NONZ
```

would result in the program commencing at the instruction at "NONZ" in the program. NOTE: The LABEL must reside in the same 6K GROM segment as the BR instruction.

Op Code: >40

Format type: 4

4.2.3 BRANCH B

Syntax definition: B LABEL

Example: LAB1 B HERE BRANCH TO ADDRESS OF HERE

Definition: Branch absolutely to address of the label operand. This branch is unconditional. The condition bit is reset after execution.

Status bits affected: / H / GT / $\overline{\text{cond}}$ / carry / OVF /

Execution results: (PC):= LABEL

Application Notes: Use the B instruction to unconditionally transfer program control to another portion of the program. If the label HERE is at the address >0B; the instruction,

B HERE

will replace the PC with the value >0B. The condition bit will be reset.
NOTE: The B instruction should be used to transfer control between 6K GROM segments.

Op Code: >05

Format type: 3

4.2.4 CASE

Syntax definition: CASE GD
DCASE GD

Example:
```
LAB1    CASE    @A      GOTO NEXT INSTRUCTION FOR (A)
                        EQUAL TO ZERO, TWO MORE IF
                        (A) EQUAL TO ONE, ETC
```

Definition: Add two times the value of the operand to the current GROM Program Counter. Resets condition bit in status.

Status bits affected: / H / GT / cond / carry / OVF /

Execution results: (PC):= 2 *(GD)+PC

Application Notes: The CASE instruction is typically followed by a series of BR statements. Since the condition bit resets after executing, the BR's are always taken. (The BR is used because it is a two-byte instruction while B is a 3-byte instruction). An example of use of the CASE statement is:

```
CASE @NMBR
BR LAB1
BR LAB2
BR LAB3
```

If the byte at location NMBR is a 0, branch to LAB1, if a 1, branch to LAB2; if a 2, branch to LAB3.
NOTE: All the labels have to reside in the same 6K GROM segment as the (D) CASE instruction.

Op Code: >8A

Format type: 6

4.2.5 CALL SUBROUTINE

Syntax definition: CALL LABEL

Example: LAB1 CALL HERE CALL THE SUBROUTINE STARTING AT THE ADDRESS OF THE LABEL HERE

Definition: Replace the PC with the address of the LABEL. Place the old PC at the top of the call stack (pointer at CPU RAM >73). Reset condition bit.

Status bits affected: / H / GT / cond / carry / OVF /

Execution results: (SBRSTK):=(SBRSTK)+2
((SBRSTK)):=(PC)
(PC):=LABEL

Application Notes: Use the CALL instruction to enter a subroutine.

Op Code: >06

Format type: 3

The following table may be used as a reference for determining when it is more economical to use a subroutine:

Instruction Set Length (m) in bytes	Minimum Number of Times Instruction Set is Used	Bytes Saved n = Times Used
3 or less	-	-
4	6	n - 5
5	4	2n - 6
6	3	3n - 7
7	3	4n - 8
8+	2	(m-3)n - (m+1)

Syntax: FETCH GD

Example: LAB1 FETCH @VAL1 FETCH 1ST PARAMETER

Definition: Retrieves a byte of data pointed to by the return address on the subroutine stack and increments this return address by 1.

Status bits affected: / H / GT / cond / carry / OVF /

Execution results: (GD) := (((SBRSTK)))
((SBRSTK)):=((SBRSTK))+1

Applications Note: Use the FETCH instruction to pass parameters in line to a subroutine . For example in this sequence,

```
      CALL  SUB
      DATA 1,24
           .
           .
           .
SUB   FETCH  @ARG1
      FETCH  @ARG2
```

The FETCH statement at SUB will place a 1 in location ARG1. The next instruction will place a 24 in location ARG2. Upon returning from the subroutine, execution commencences at instruction after the 24. The FETCH instruction uses two bytes of the subroutine stack. The FETCH instruction can only use CPU RAM as GD.

Op Code: >88

Format type: 6

4.2.7 RETURN FROM SUBROUTINE

Syntax definition: RTN

Example: LAB1 RTN RETURN WITH 0 TO CONDITION

Definition: Replaces PC with the value at the top of the subroutine stack (pointer at >73 in CPU RAM). Resets the condition bit.

Status bits affected: / H / GT / cond / carry / OVF /

Execution results: (PC):= ((SBRSTK))
(SBRSTK):=(SBRSTK)-2
COND:=reset

Applications Notes: RTN is used to return from a subroutine call when you don't care about saving the condition bit value. By changing the value of the top of the subroutine call stack (pointed to by CPU RAM location >73), the return address may be modified.

Op Code: >00

Format type: 5

4.2.8 RETURN FROM SUBROUTINE (SAVE CONDITION)

Syntax definition: RTNC

Example: LAB1 RTNC RETURN WITH NO EFFECT ON STATUS

Definition: Replaces PC with the value at the top of the subroutine stack (pointer at >73 in CPU RAM). Does not affect status.

Status bits affected: / H / GT / cond / carry / OVF /

Execution results: (PC):= ((SBRSTK))
(SBRSTK):=(SBRSTK)-2

Applications Notes: see RTN

Op Code: >01

Format type: 5

4.3 BIT MANIPULATION INSTRUCTIONS

Individual bits of memory may be set, reset, or tested using bit operations. The memory bits are numbered 76543210, with 0 being the least significant and 7 the most significant bit. The immediate operand that specifies bit number is truncated to 3 bits. The status byte is modified by these insructions.

These instructions are macro-instructions which the assembler converts into equivalent GPL instructions.

4.3.1 RB GD,IMM := AND IMM_1,GD

Reset the bit in memory identified by the two operands. The COND bit is set if the resulting destination byte is zero and reset otherwise. Note that an AND instruction is generated by the assembler.

4.3.2 SB GD,IMM := OR IMM_1,GD

Set the bit in memory identified by the two operands. The COND bit is always reset. This instruction is assembled as an OR instruction.

4.3.3 TBR GD,IMM := CLOG IMM_1,GD := $IF BIT(IMM) GD .EQ. Ø THEN

Test the bit in memory identified by the two operands and set the COND bit if the tested bit is a zero. Otherwise reset the COND bit. This instruction is assembled into a CLOG statement.

4.4 ARITHMETIC & LOGICAL INSTRUCTIONS

Arithmetic operations work on operands in two's complement form and affect the status byte. The result of an add, subtract, increment, or decrement instruction sets the COND bit if the result is zero, the H bit if logical high, the GT bit if arithmetic greater than, the OVF bit on overflow, and the CARRY bit if a carry occurs from the most significant digit. The divide instruction sets the OVF bit if the divisor is less than or equal to the first byte of the dividend. The compare instructions compare the destination operand to the source operand. For example, a CGT instruction sets the COND bit if the destination is greater than the source.

The address fields of these instructions contain one or two operands. In general the first is the source operand and the second the destination. For example, in an add operation the first operand is added to the second and in a subtract operation the first is subtracted from the second.

4.4.1 ADD

Syntax definition: ADD GS,GD
DADD GS,GD

Example: LAB1 ADD 48,@X(ONE) ADD 48 TO (X) INDEXED BY (ONE)

Definition: Replace GD with the sum of the GS and GD. Compare the result to zero and set/reset status bits to indicate this result

Status bits affected: / H / GT / cond / carry / OVF /

Execution resluts: (GD) := (GS)+(GD)

Application notes: ADD is used to add Twos complement integer. For example, if the address labeled TABLE contains >FE and the address labeled NO contains a >01; the instruction

ADD @TABLE,@NO

would result in NO containing a >FF and TABLE remaining unchanged. The logical high bit would be set and the other bits reset.

Op Code: >A0

Format type: 1

4.4.2 SUBTRACT

SUB OR S
DSUB OR DS

Syntax definition:
```
SUB    GS,GD
DSUB   GS,GD
```

Example:
```
LAB1    SUB   @A,@B    SUBTRACT (A) FROM (B)
```

Definition: Replace GD with the GD less the GS. Compare the result to zero and set/reset status bits to indicate this result.

Status bits affected:

H	GT	cond	carry	OVF

Execution results: (GD) := (GD) - (GS)

Application notes: Use the SUB instruction to subtract signed integer values. For example, if the location NEW contains a value of >6F and memory location OLD contains a value of -1; the instruction,

```
SUB   @OLD,@NEW
```

results in the contents of NEW changing to >70. The logical high, greater than status bits set, the others reset.

Op Code: >A4

Format type: 1

4.4.3 MULTIPLY

Syntax definition: MUL GS,GD
DMUL GS,GD

Example: LAB1 MUL >4,@A MULTIPLY >4 TIMES (A)

Definition: Multiply the GD by the GS. In the single byte MUL, both operands are single byte values but the result is stored in a double byte location at GD. The 8 most significant bits are stored in the GD. In the double byte DMUL, both operands are double byte values and the result is a four byte value at GD. No status bits are affected. The multiply is an unsigned type.

Status bits affected: / H / GT / cond / carry / OVF /

Execution results: MUL : (GD,GD+1) :=(GS)*(GD)
DMUL : (GD,GD+1,GD+2,GD+3):= (GS, GS+1) *
(GD, GD+1)

Application notes: In the single MUL the GS & GD are 8-bit values. The result is a 16-bit value. In the double DMUL, the GS and GD are 16-bit values. The result is a 32-bit value. For example, if location A contains a >F3 and location B contains a >82, the instruction

MUL @A,@B

would result in location A being unchanged & location B containing >7B, and location (B+1) containing >66. Status bits are unchanged.

Op Code: >A8

Format type: 1

4.4.4 DIVIDE

Syntax definition: DIV GS,GD
DDIV GS,GD

Example: LAB1 DIV >08,@VALUE , DIVIDE TWO BYTES STARTING AT VALUE BY >08

Definition: Replace the GD with the quotient and remainder of GD divided by GS. Compare the result to zero and set/reset status bits to indicate the result.
The divide is of the signed type.

Status bits affected: / H / GT / cond / carry / OVF /

Execution results:
DIV: (GD):= (GD,GD+1)/(GS) ;(GD+1):= remainder;
DDIV: (GD,GD+1):= (GD,GD+1,GD+2,GD+3)/(GS,GS+1);
(GD+2,GD+3):= remainder.

Application Note: If the DIV instruction is a single byte instruction, the single byte GS is divided into the double byte GD and the quotient is put in the GD. If the DDIV instruction is used, the two byte GS is divided into the four byte GD and the quotient is put into the two bytes at GD; the remainder is placed in two bytes at GD+2.

Op Code: >AC

Format type: 1

4.4.5 INCREMENT BY ONE

INC
DINC

Syntax definition:

```
INC     GD
DINC    GD
```

Example:

```
LAB1    INC  @A     INCREMENT (A) BY 1
```

Definition: Replace the GD with the GD plus one. The result is compared with zero and the status bits are set/reset to indicate the result of this comparison.

Status bits affected:

/ H /	GT /	cond /	carry /	OVF /
'	'	'	'	'

Execution results: (GD) := (GD)+1

Application notes: Use the INC instruction to count and index byte arrays, add a value of one to an addressable memory location, or set flags. For example, if COUNT contains a zero, the instruction

```
INC @COUNT
```

places a >0001 in COUNT and sets the logical high, and arithmetic greater than status bits, while the condition, carry, and overflow status bits are reset.

Op Code: >90

Format type: 6

4.4.6 INCREMENT BY TWO

INCT
DINCT

Syntax definition:

```
INCT     GD
DINCT    GD
```

Example:

```
LAB1     INCT    @A     INCREMENT (A) BY 2
```

Definition: Replace the GD with the GD plus two. The result is compared with zero and the status bits are set/reset to indicate the result of this comparison.

Status bits affected: / H / GT / cond / carry / OVF /

Execution Results: (GD) := (GD)+2

Application notes: Use the INCT instruction to count and index double byte arrays and add a value of two to an addressable memory location. For example, if TEMP contains the address >00 (i.e. points to the first temporary two-byte location in CPU RAM; the instruction,

```
INCT @TEMP
```

places a 0002 in TEMP (so that it now points to the next two bytes of temporary byte CPU RAM).

Op Code: >94

Format type: 6

4.4.7 DECREMENT BY ONE

DEC
DDEC

Syntax definition: DEC GD
DDEC GD

Example: LAB1 DEC @A DECREMENT (A) BY 1

Definition: Replace the GD with the GD minus one. The result is compared with zero & the status bits are set/reset to indicate the result of this comparison.

Status bits affected: / H / GT / cond / carry / OVF /

Execution results: (GD) := (GD)-1

Application notes: Use the DEC instruction to subtract a value of one from any addressable operand. The DEC instruction is also useful in counting and indexing byte arrays. For example, if COUNT contains a value of 1, then

DEC @COUNT

results in a value of zero in location COUNT & sets the condition and carry status bits while resetting the logical high, arithmetic greater than, and overflow status bits.

Op Code: >92

Format type: 6

4.4.8 DECREMENT BY TWO

DECT
DDECT

Syntax definition: DECT GD
DDECT GD

Example: LAB1 DECT @A DECREMENT (A) BY 2

Definition: Replace the GD with the GD minus two. The result is compared with zero and the status bits are set/reset to indicate the result of this comparison.

Status bits affected:

H	GT	cond	carry	OVF
'	'	'	'	'

Execution results: (GD) := (GD)-2

Application notes: The DECT instruction is useful in counting & indexing two byte arrays. Also, use the DECT instruction to subtract a value of two from any addressable operand. For example, if COLOR contains the value >0A the instruction

```
DECT @COLOR
```

would result in the value >08 being stored in COLOR .

Op Code: >96

Format type: 6

4.4.9 ABSOLUTE VALUE

Syntax definition: ABS GD

Example: LAB1 ABS @DX(INDEX) ABSOLUTE VALUE OF (DX) INDEXED BY (INDEX)

Definition: Replace the GD with the absolute value of the GD. Does not affect status bits.

Status bits affected: / GT / H / cond / carry / OVF /

Execution results: (GD) := ABS(GD)

Application notes: Use the ABS instruction to take the absolute value of an operand. For example if the location >76 (joystick Y) contains -4 then

```
ABS @ 76
```

will result in a +4 at >76.

Op Code: >80

Format type: 6

4.4.10 NEGATE

NEG
DNEG

Syntax definition: NEG GD
DNEG GD

Example: LAB1 DNEG @B NEGATE TWO BYTES AT B

Definition: Replace the GD with its two's complement value. Does not affect status bits .

Status bits affected: / H / GT / cond / carry / OVF /

Execution results: (GD) := -(GD)

Applications notes: Use the NEG instruction to make the contents of an addressable memory location its additive inverse. For example if TEMP contains the value of 1, the instruction,

NEG @TEMP

would result in the value > FF being stored in TEMP.

Op Code: >82

Format type: 6

4.4.11 INVERT

INV
DINV

Syntax definition: INV GD
DINV GD

Example: LAB1 INV @A INVERT (A)

Definition: Replace the GD with its one's complement value. Does not affect status bits.

Status bits affected: / GT / H / cond / carry / OVF /

Execution results: (GD) := LOGICAL INVERSION (GD)

Application notes: Use the INV instruction to complement an operand. For example if location COUNT contained a zero; the instruction

INV @COUNT

will result in a >FF being stored in COUNT.

Op Code: >84

Format type: 6

4.4.12 LOGICAL AND

AND
DAND

Syntax definition:
```
AND    GS,GD
DAND   GS,GD
```

Example: `LAB1 AND >F0,@Y` SET 4 LSB OF (Y) TO ZERO

Definition: Perform a bit-by-bit AND operation of the 8 (16) bits in GS with the GD and store the result in the GD. The result is compared to zero and the status bits are set/reset to indicate the result.

Status bits affected:

/ H /	GT /	cond /	carry /	OVF /
↑	↑	↑	↑	↑

Execution results: (GD) := (GS) AND (GD)

Application notes: Use the AND instruction to perform a logical AND operation between a GS and GD. The AND operation is useful in masking out bits before a comparison. If location X contains a >66 and location Y contains a >0F; the instruction

```
AND @Y,@X
```

would result in X containing a >06. The GT and H status bits will be set and all other status bits reset.

Op Code: >B0

Format type: 1

4.4.13 LOGICAL OR

OR
DOR

Syntax definition:
```
OR    GS,GD
DOR   GS,GD
```

Example:
```
LAB1    DOR >FFFE,@VALUE    "OR" THE DOUBLE BYTE
                             IMMEDIATE VALUE FFFE WITH
                             (VALUE)
```

Definition: Replace the GD with the GD OR'd with the GS. Compare the result to zero & set/reset the status bits to indicate this result.

Status Bits Affected: / H / GT / cond / carry / OVF /

Execution results: (GD) := (GS) OR (GD)

Application notes: Use the OR instruction to perform a logical OR between the GS and GD. If location A contains >F6 and location B contains a >68 the instruction

```
OR @A,@B
```

would result in location B changing to >FE. The logical high status bit will be set, the rest will be reset.

Op Code: >B4

Format type: 1

4.4.14 EXCLUSIVE OR

XOR
DXOR

Syntax definition: XOR GS,GD
DXOR GS,GD

Example: LABl XOR >F8,@A "EXCLUSIVE OR" >F8 WITH (A)
A

Definition: Exclusively OR the GS and GD and replace the GD with the result. The result is compared to zero and the status bits are set/reset to indicate the result.

Status bits affected: / H / GT / cond / carry / OVF /

Execution results: (GD) := (GS) XOR (GD)

Application notes: The exclusive OR is accomplished by comparing the GD and GS on a bit-by-bit basis. If the bits are both 0 or both 1, the GD is reset; otherwise it is set. If location A contains >88 and location B contains >87, the instruction

```
XOR  @A,@B
```

would result in location B changing to >OF. The logical high and greater than status bits will be set, the rest will be reset.

To reverse bits in a byte, do an XOR with a number which has all bits set you want to reverse.

Op Code: >B8

Format type: 1

4.4.15 CLEAR LOCATION

CLR
DCLR

Syntax definition: CLR GD
DCLR GD

Example: LAB1 CLR @A STORE ZERO IN (A)

Definition: Replace the GD with a zero. Does not affect status bits.

Status bits affected: / H / GT / cond / carry / OVF /

Execution results: (GD) := 0

Applications notes: Use the CLR instruction to replace any addressable memory location with a zero.

Op Code: >86

Format type: 6

4.4.16 STORE

ST
DST

Syntax definition: ST GS,GD
DST GS,GD

Example: LABl ST @X,@TEMP STORE CONTENTS OF LOCATION X IN LOCATION TEMP

Definition: Replace the GD with the GS. Status bits are not affected.

Status bits affected: / H / GT / cond / carry / OVF /

Execution results: (GD) := (GS)

Application notes: Use the ST instruction to copy the contents of any addressable memory location or an immediate value into any addressable memory location. For example, if location X contains a>88; the instruction

ST @X,@TEMP

will result in both location X and TEMP containing a >88.

Op Code: >BC

Format type: 1

4.4.17 EXCHANGE

EX
DEX

Syntax definition:

```
EX      GD,GD
DEX     GD,GD
```

Example:

```
LAB1    EX      @X,@Y       EXCHANGE THE CONTENTS OF
                            LOCATIONS X & Y.
```

Definition: The contents of the first operand is exchanged with the contents of the second operand. No status bits are affected.

Status bits affected: / H / GT / cond / carry / OVF /

Execution results: (GD) (EXCHANGE WITH) (GD)

Application notes: Use the EX instruction to exchange the contents of two locations in memory. For example if location >380 in VDP RAM contains a 03 and location >381 in VDP RAM contains a 05 the instruction;

```
EX  RAM(>380),RAM(>381)
```

would result in location>380 in VDP RAM containing a >05 & location >381 in VDP RAM containing a >03; thus swapping the colors of pattern set #0 with pattern set #1.

Op Code: >C0

Format type: 1

4.4.18 PUSH ONTO DATA STACK

Syntax definition: PUSH GD

Example:
```
LAB1    PUSH    @NEWEST     PUSH VALUE AT LOCATION
                            NEWEST ONTO DATA STACK
```

Definition: Increment the data stack pointer & push the one byte operand onto it. (Opposite of instruction POP). No status bits are affected.

Status bits affected: / H / GT / cond / carry / OVF /

Execution results:
```
(DATSTK)    (DATSTK) + 1
((DATSTK)):=(GD)
```

Application notes: Use PUSH instruction to add to data stack. Opposite of POP.

Op Code: >8C

Format type: 6

4.4.19 POP OFF OF DATA STACK

Syntax definition: POP GD

Example:

```
LAB1  POP  @DAT     POP top value off data
                    stack and into location DAT
```

Definition: Pop a byte off the data stack and load it into GD, then decrement the value of the data stack pointer

Status bits affected: / H / GT / cond / carry / OVF /

Execution results:

```
(GD):=((DATSTK))
(DATSTK):= (DATSTK) - 1
```

Application notes: This is a macro instruction which the assembler accepts. The pop instruction is the opposite of the PUSH instruction. It assembles into:

```
ST    *STATUS,GD
```

The interpreter traps this out to POP a byte of data off the data stack and places it into the GD, then decrements the data stack pointer.

4.4.20 BLOCK MOVE

Syntax definition: MOVE GS_1 FROM GS_2 TO GD

Example: MOVE 21 FROM ROM(#TABL) TO RAM(800)

Definition: Move the specified number of bytes from the source to the destination.

Status bits affected:

/ H / GT / cond / carry / OVF /
L1 (GD):=(GS2)
GD:=GD+1
GS_2:=GS_2+1
GS_1:=GS_1-1
$IF GS_1 .GT. 0 GOTO L1;

Execution results: The requested number of bytes are transferred from the Source to the destination.

Application notes: The MOVE instruction is useful wherever a block of data must be moved from one section of memory to another. Note that the byte count is IMM or CPU and a double-byte value.The Source and Destination operands can represent blocks in CPU RAM, VDP RAM, or ROM. In addition, the following mnemonics can be used:

Destination only: VDP(IMM) ..block of VDP Registers

Instead of using a LABEL for GROM, an IMM field 0-48K, or a GS pointing into CPU RAM can be used (e.g. ROM(22), ROM(@CPULOC)). Furthermore, an index can be used, as in the normal addressing mode (e.g. ROM(#AB(A)). The VDP registers cannot be used as Source, since they are write-only registers. The MOVE instruction uses two bytes of the sub-routine stack.

Op code: >20

Format type: 9

More examples:

```
MOVE 7 FROM ROM(2000) TO VDP(1)    ..loads up registers 1 thru 7.
MOVE @COUNT FROM @0 to @100          ..copy a block from CPU to CPU
```

4.4.21 SHIFT LEFT LOGICAL

Syntax Definition:
```
SLL     GD, GS
DSLL    GD, GS
```

Example:
```
LAB1    SLL    @ VALUE,5        SHIFT (VALUE) LEFT
                                LOGICAL 5 BITS
```

Definition: Shift the (GD) left for the (GS) number of bits. Fill in the vacated bits with logical zeros. Status is not affected.

Status bits affected: / H / GT / cond / carry / OVF /

Execution results: Shift the (GD) left for the (GS) number of bits and fill in the vacated bit with zeros.

Application notes: Use the shift left logical to shift the GD. For example, if VAL has >21 in it, the instruction

```
SLL @VAL, 2
```

results in the contents of VAL becoming >84. DSLL requires a 2-byte shift count.

Op Code: >EO

Format type: 1

4.4.22 SHIFT RIGHT ARITHMETIC

SRA
DSRA

Syntax definition: SRA GD, GS
DSRA GD, GS

Example: LAB1 SRA @A,@B SHIFT (A) RIGHT ARITHMETIC BY THE NUMBER OF BITS SPECIFIED IN LOCATION B

Definition: Shift the (GD) right for the (GS) number of bits. Fill in the vacated bits with the MSB of (GD). Status is not affacted.

Status bits affected: / H / GT / cond / carry / OVF/

Execution results: See definition

Application Notes: An example of an arithmetic right shift is: if location contains a >86; the instruction

SR @A,2

will result in changing location A to be a >E1. DSRA requires a 2-byte shift count.

Op Code: >DC

Format type: 1

4.4.23 SHIFT RIGHT LOGICAL

SRL
DSRL

Syntax definition: SRL GD, GS
DSRL GD, GS

Example: LAB1 SRL @VALUE,7 SHIFT (VALUE) RIGHT 7 BIT POSITIONS

Definition: Shift the contents of the GD to the right for the (GS) number of bits while filling in the vacated bit positions with zeros. Status is not affected.

Status bits affected: / H / GT / cond / carry / OVF /

Execution results: See definition

Application notes: An example of a logical right shift is: If the double byte location A contains the value >FFEF, then the instruction,

```
DSRL @A,3
```

changes the contents of location A to >1FFD. DSRL requires a 2-byte shift count.

Op Code: >E4

Format type: 1

4.4.24 SHIFT RIGHT CIRCULAR

SRC
DSRC

Syntax definition:

SRC	GD,	GS
DSRC	GD,	GS

Example:

LAB[1]	SRC	@A,@B	SHIFT (A) RIGHT CIRCULAR BY THE NUMBER OF BITS SPECIFIED IN LOCATION B.

Definition: Shift the (GD) to the right for number of bits specified in the GS while filling vacated bit positions with the bit shifted out (LSB). Status bits are not affected.

Status bits affected: / H / GT / cond / carry / OVF /

Execution results: See definition

Application notes: An example of a right circular shift is, if location VALUE contains a >A5, the instruction

```
SRC @VALUE,1
```

will result in location VALUE containing a >D[2]. DSRC requires a 2-byte shift count

Op Code: >E[8]

Format type: 1

4.5.1 COINCIDENCE

Syntax definition: LAB 1 COINC GS, GD

Example: COINC RAM (>300),RAM(>304)

Definition: The Source operand must indicate a Y,X byte pair for object 1; likewise, the Destination operand indicates the Y,X byte pair for object 2; COINC sets the status equal bit if the objects are in coincidence; otherwise it resets the status equal bit.

Status bits affected: H / GT / cond / carry / OVE /

Execution results: COND = (objects in coincidence?)

Application notes: See Appendix E for details on operation of the COINC INSTRUCTION.

Op Code: >ED

Format type: 1

4.5.2 LOAD BACKDROP COLOR

Syntax definition: BACK IMM

Example: LAB1 BACK 3 LOAD BORDER WITH COLOR 3

Definition: Load the border area of the display with the immediate color specified. Does not affect status bits. Loads VDP register 7.

Status bits affected: / H / GT / cond / carry / OVF /

Execution results: (VDP reg.7) := IMM

Application notes: Use the BACK instruction to change the VDP register 7 to change the border color of the display to the desired color.

Op Code: >04

Format type: 2

Syntax definition: ALL IMM

Example: LAB1 ALL 0 LOAD EVERY BLOCK ON SCREEN WITH PATTERN #0 (RESIDES AT >800 - >807 IN VDP RAM)

Definition: Replace every byte in the pattern name table (768 bytes) with the immediate operand. No status bits are affected.

Status bits affected: / H / GT / cond / carry / OVF /

Execution results: ST IMM,RAM(0)67
MOVE 767 from RAM(0) to RAM(1)

Application notes: Use the ALL instruction to display a repetitive pattern on the entire screen. This can be used to clear the screen. Assuming >900 to >907 (Pattern number >20) contains 00's (which they will at power up from the ASCII default character set); the instruction

ALL >20

will result in the the sceen getting filled up with ASCII blanks. No status bits are affected.

Op Code: >07

Format type: 2

4.5.4 FORMATTED BLOCK MOVE

Syntax definition: FMT OPERAND1,OPERAND2,OPERNAND3,...

Example: FMT BIAS=>20,4('0,2,4,6')

Definition: Output immediate and variable data to the Pattern Name Table in a controlled, formatted fashion.

Status bits affected: / H / GT / cond / carry / OVF /

Execution results: The Pattern Name Table is modified; see below.

Application notes:

The operands of the FMT instruction are encoded by the assembler and placed inline after the FMT op code. The FMT processor in the interpreter is essentially a sub-interpreter in that its "language" is different from the rest of GPL. The FMT instruction places data into the VDP Pattern Name Table in such a way that the resulting screen image is formatted in the way the programmer desires.

The locations XPT and YPT in the CPU RAM are used heavily by FMT to determine where to put the next bytes of data. These locations can be set within the FMT statement; they are updated by the FMT statement also.

Some of the FMT capabilities are:

- Place a sequence of immediate data across the screen from a defined starting point;
- Place a sequence of immediate data down the screen from a defined starting point;
- Repeat the same immediate data byte or sequence of bytes across or down the screen;
- Nest the above features in order to put data up in a rectangular fashion;

Each of the OPERANDs can be of one of the following formats:

'IMM,IMM,IMM,.....'
(places the data across the screen from starting point specified by XPT,YPT);

"IMM,IMM,IMM,...."
(places the data down the screen from starting point specified by XPT,YPT);

M('@IMM')
(repeats the data from location IMM in CPU RAM across the screen M times, where M is from 1 to 32, or left off for 1, uses the data stack);

M'IMM'
(repeats the same value across the screen M times; M is from 1 to 32, or left off for 1);

N"IMM"
(repeats the same value down the screen N times; N is from 1 to 32);

':character string:' or ":character string:"
(outputs the ASCII equivalents of the character string to the Pattern Name Table; note also that this colon-delimited string can be used wherever IMM is called for in the above formats);

N
(adds the value of N to YPT, N from 1 to 32);

M
(adds the value of M to XPT, M from 1 to 32);

XPT=IMM
(sets XPT to a specified value);

YPT=IMM
(sets YPT to a specified value);

BIAS=GS
(sets the BIAS to a specified value, see below);

Upon entering the FMT statement, the BIAS is 0. Everything that gets output to the Pattern Name Table gets the value of BIAS added to it. Setting the BIAS to a non-zero value allows using the same FMT statement to output alternate character sets, the same character set of different colors, etc.

Any sequence of operands to the FMT instruction may be enclosed in parentheses, and a "loop" count constant used in front of it. The operands inside the parentheses are then effectively repeated the number of times specified by the loop count. Examples of this will be seen below. Furthermore, these loop structures may be nested inside one another.

If a horizontal boundary is reached while outputting data to the VDP, XPT is reset to 0 and YPT is incremented. Thus further data is then output starting at the beginning of the next line. If the vertical boundary is reached, YPT is reset to 0 ; XPT is kept the same (this means vertical wrap-around will be to the same column).

Examples:

```
FMT 3":HELLO:"          (Repeats HELLO 3 times
                         down the screen)

HOME
FMT 32'>E0',22('>E0',30<,'>E0'),32,'>E0'
                        (Puts a border around the
                         screen one character wide
                         of character >E0)

FMT BIAS=>A0,XPT=13,YPT=23,':TENNIS:'
                        (Adds >A0 to the hex value of
                         the ASCII characters and puts
                         those characters on the screen)

PADL1 EQU >A5

FMT XPT=15,YPT=1, 22"PADL1"  (Puts 22 of characer >A5
                              down the screen)

FMT  BIAS=>20,2('@NUM')  (Adds >20 to the value stored
                         in NUM and puts 2 of those
                         characters on the screen.)

FMT 3(1^,29<,3(':AAA:'))  (Moves pointer one line down,
                         29 spaces right, and puts 9
                         A's on the screen--repeats
                         this two more times)

FMT  BIAS=RAM(0),1,2,3,4,5,6,7,8,9,0
                        (Adds the value at RAM(0) to
                         each number and puts that
                         character across the screen)

FMT XPT=0,YPT=0,":MY:",1^,2<,":ARM:",1^,3<,
         ":THAT:",1^,4<,":THROWS:"
                        (Starting at the top corner,
                         puts MY down the screen, moves
                         down 1 line, right 2 spaces,
                         puts ARM down screen, moves
                         down 1 line, right 3 spaces,
                         puts THAT down the screen,
                         moves down one line, right 4
                         spaces, puts THROWS down the
                         screen.)
```

Op Code >08

Format type: 7

Syntax definition: RAND IMM

Example: LABEL RAND 25 GENERATE A RANDOM NUMBER FROM 0 TO 25 INCLUSIVE

Definition: Generate a random number from zero to the immediate operand and store this number in location >78 of CPU RAM. Does not affect status bits. If no immediate value is specified, the default is 255.

Status bits affected: / H / GT / cond / carry / OVF /

Execution results: (RAND) := Random number in (0,IMM)

Application notes: Use RAND to generate random numbers. For example, for RAND sprite motion the instruction,

```
RAND 3
```

would generate a random number between 0 and 3 inclusive.
There is a useful way to generate the initial seed for the random number:

```
LOOP1 RAND
      SCAN
      BR LOOP1
```

This method generates a "random" number of calls to RAND, depending on how long it takes for a key to be pressed. All subsequent calls to RAND will thus generate unique random numbers every time the program is run. It is good to use this loop everywhere you do a scan if you need really random numbers.

Op Code: >02

Format type: 2

4.5.6 SCAN KEYBOARD

Syntax definition: SCAN

Example: LAB1 SCAN SCAN KEYBOARD

Definition: Scans keyboard specified in >74 in CPU RAM. Returns the keycode in location >75, the Y-position of the joystick in location >76, and the X-position in location >77. The COND bit is set if a key is found depressed; however, the keypad or keyboard is "debounced" in the sense that if the same key is found depressed as was depressed upon the previous call to SCAN (on the same keyboard), the proper keycode is returned, but the COND bit is reset.

Status bits affected: / H / GT / $\overline{\text{cond}}$ / carry / OVF /

Execution results:
>75 := KEY value
>76 := JOYY
>77 := JOYX
COND := set if new key; reset if old key or no key

Applications notes: See Appendix D for details on handsets and keyboards.

Assembly Language: There is a keyboard scan subroutine that can be called while executing 9900 Assembly Language code. This subroutine is located at location >000E in the console ROM. The keyboard number (CPU >74) should be specified before calling the subroutine. A BL to this subroutine will serve the same purpose as a SCAN instruction in Graphics Language.

Op Code: >03

Format type: 5

4.5.7 EXECUTE MACHINE LANGUAGE

Syntax definition: XML IMM

Example: LAB1 XML >05

Definition: Begin execution of 9900 machine language. Use the IMM field to tell where.

Status bits affected: / H / GT / cond / carry / OVF /

Execution results: Execute 9900 machine language directly.

Application Notes: The immediate field of the XML instruction is split into a left nybble (table number) and a right nybble (index into table). There are 16 table addresses defined in the CPU address space. See Table 4.5.1 for a list of these hardcoded values (note that they have been arranged so as to insure that at least one table exists in any conceivable plug-in fast ROM). The left nybble specifies which of the 16 tables to get the Branch address from. The right nybble is then used to determine which of the 16 addresses in the table to use. Each table can contain up to 16 2-byte entry-point addresses. Note that one can have less if one wishes. As an example of XML,

```
XML >24
```

causes a branch to the address contained in the fifth entry of Table 2.

This technique makes it easy to have a plug-in ROM which contains up to 16 routines in it. One determines which table will reside in the ROM's address space; at the table address in the ROM, the entry vectors for the ROM routines should be stored. Upon entry to a routine, the 9900 workspace pointer is set to >83E0 and the 9900 ST is set to an unknown value. To return control to the interprer, make sure WP is still >83E0 and execute a "B *R11". GPL execution will continue out of the GROM from which he XML was seen, at the address specified by the internal GROM address.

To do a keyboard scan in 9900 Assembly Language, do a BL to a subroutine located at location >000E in console ROM.

Op Code: >0F

Format type: 2

TABLE 4.5.1
XML TABLE

TABLE #	FUNCTION	ADDRESS (>)
0	FLOATING POINT ROUTINES	"FLTTAB"
1	CONVERSION AND BASIC ROUTINES	"XTAB"
2	SYSTEM EXPANSION ROM/RAM	2000
3	BASIC ENHANCEMENT	3FC0 and "XTAB3"
4	BASIC ENHANCEMENT	3FE0 and "XTAB2"
5	PERIPHERAL/DSR ROM/RAM	4010
6	PERIPHERAL/DSR ROM/RAM	4030
7	GROM MODULE ROM/RAM	6010
8	GROM MODULE ROM/RAM	6030
9	GROM MODULE ROM/RAM	7000
10	FUTURE EXPANSION	8000
11	FUTURE EXPANSION	A000
12	FUTURE EXPANSION	B000
13	FUTURE EXPANSION	C000
14	FUTURE EXPANSION	D000
15	SCRATCH PAD RAM	8300

4.5.8 EXIT GPL

Syntax definition: EXIT

Example: EXIT

Definition: Terminate GPL execution; return control to the system monitor.

Status bits affected:

Execution results: The monitor performs a restart sequence.

Application notes: All GPL programs that terminate should use the EXIT command. See the Monitor Specification for details on system restart.

Op code: >0B

Format type: 5

4.5.9 I/O INSTRUCTION

Syntax definition: I/O GS,IMM

Example: I/O @BLOCK,2

Definition: This is an extended instruction in the sense that the action that occurs depends upon the value of the IMM field. Specifically, this instruction does SOUND, CRU input and output.

Status bits affected: / H / GT / cond / carry / OVF /

Execution result: See below.

Application notes: See Appendix F for currently supported uses for the I/O instruction.

Op code: >F6

Format type: 8

4.5.10 HOME INSTRUCTION

Syntax Definition: HOME

Example: HOME

Definition: SET XPT and YPT equal to zero

Status bits affected: None

Execution results: >7E (YPT): = 0
>7F (XPT): = 0

Application notes: The HOME instruction assembles the same as:

```
DCLR YPT
```

SOURCE FILE FORMAT

GPL source instructions are entered as card images to the assembler in a free field format with the restrictions that LABEL fields must begin in column 1 and Operand list fields must begin before column 25. No imbedded blanks are permitted within an operand list. Blank lines in the source are ignored.

The format of a typical instruction is:

(LABEL) (INSTRUCTION MNEMONIC) (OPERAND LIST) (COMMENT)

The LABEL field is always optional. It consists of an alpha-numeric string of up to 6 characters , the first of which must be non-numeric. Up to 1000 LABELs can be defined in any one source file. Any label that is defined in a source file can be referred to in the OPERAND LIST of any other instruction in that source file. Note that SYMBOLS (as defined using the EQU directive) are exactly like LABELs and their usage is the same.

The INSTRUCTION MNEMONIC must be one of the valid mnemonics as described in Section 4 of this manual. The OPERAND LIST must be of the type required by that particular instruction. For instructions that allow OPERAND LISTs of arbitrary length, this field may be continued up to 16 lines by terminating an OPERAND with a semicolon instead of a comma (the FMT is an example).

The comment field has no restrictions except that it cannot span lines. It must be separated from the OPERAND LIST by at least one space.

ASSEMBLER DIRECTIVES

These directives have a format similar to GPL instructions; they can include LABEL fields as well as comment fields.

DATA IMM,IMM,IMM,...

The DATA instruction is used to generate a sequence of bytes in the Graphics ROM. The address field contains a list of immediate values or LABELS. In conjunction with the MOVE instruction, the DATA statement provides a way to load up a sequential block of CPU or VDP RAM. For example:

```
MOVE  10 FROM ROM(#LAB1) TO RAM(>300)
```

where later on in the source:

```
LAB1   DATA 0,1,2,3,4,5,6,7,8,9
```

TITLE XXXXXXXX

The TITLE directive provides an 8 character string that is printed at the top of each page of listing and included in the object file. It generates no code and should be placed at the very beginning of the source file.

END

The END directive may be used to separate blocks of code. It also is required to terminate the source file.

(SYMBOL) EQU IMM

The EQU directive assigns the immediate field value to the symbol that starts in column one. A symbol may be a one or two byte value.

GROM IMM

The GROM directive selects which GROM the assembled program is to be in. In the current definition of the system the operand must be less than eight and the maximum length of the program in a GROM is 6K. The GROM directive sets the assembler location counter to the start of the selected ROM. Remember that if a program is longer than 6144 (>1800) bytes it must be partitioned into segments. The only way to transfer control from one GROM to another is through the long Branch instruction. However, references can be made to LABELs and SYMBOLs in different GROMS.

ORG IMM

The ORG directive sets the assembler location counter to the displacement within the currently selected ROM specified by the operand. This is useful for generating data in a different section of the GROM than the program. IMM must be a value from 0 to >17FF.

BASE IMM_1,IMM_2,IMM_3,IMM_4,IMM_5,IMM_6, IMM_7

The BASE directive specifies the base addresses for the various sub-blocks in VDP RAM. The seven operands are the base addresses for the Pattern Name Table, Pattern Generator Area, the Pattern Color Table, Sprite Attribute List, Sprite Descriptor Blocks, Sprite Velocity Table, and object code bias. The default values correspond to the standard configuration and are 0, >800, >380, >300, >400, >780 and 0. It is necessary to use a BASE directive only if one wishes to use the special mnemonics in the MOVE instruction, and base addresses other than the defaults listed above are used.

PAGE

The PAGE directive causes the listing to continue on a new page. The PAGE statement is not printed.

LIST

The LIST directive restores printing of the source listing. This directive is required only when UNL directive is in effect, to cause the assembler to resume printing. The LIST statement is not printed.

UNL

The UNL directive inhibits printing of the source listing. The UNL statement is not printed.

LISTM

The LISTM directive restores printing of multiple lines of object code. This directive is required only if a UNLM directive is in effect. This statement is not printed.

UNLM

The UNLM directive inhibits printing of multiple lines of object code. This statement is not printed.

GPL MACROS

These macro instructions are designed to allow implementation of control statements similar to those in high level languages like PASCAL. Table A.1 shows the GPL instructions which each macro expands to. The mnemonics for the statements are:

$END

Terminator for the $WHILE, $FOR, $IF, $ELSE, and $SELSE statements.

$SEND

Same as end $END. When used as a terminator for $WHILE and $FOR, it generates a BR instead of a B.

$WHILE GD .R. GS

Causes the following block to be executed as long as the comparison is true. A list of valid relations is given below.

$REPEAT

Causes the following block to be executed until the comparison in the terminating $UNTIL statement is true. The block is executed at least once.

$UNTIL GD .R. GS

Terminator for the $REPEAT statement.

$FOR GD = GS TO GS BY GS

Causes the following block to be executed as a loop. The loop is controlled by a counter specified by the first operand. The counter is initialized by the second operand and incremented by the optional fourth operand until it is greater than (arithmetic compare) the third operand. The range of each GS operand is 0- >7F. If there is no fourth operand specified,a default value of 1 is used to increment the second operand.

$FOR GD = GS DOWNTO GS BY GS

Same as previous statement except that the counter is decremented by the optional fourth operand until it is less than (arithmetic compare) the third operand.

$IF GD .R. GS GOTO LAB

The branch is taken if the comparison is true and otherwise execution continues at the next line. No END statement is required with this form of $IF. LAB must be a label in the same GROM because the compare generates a BS or BR instruction.

$IF GD .R. GS THEN

The following block is executed if the comparison is true. If false it is skipped. An END statement must terminate the block.

$ELSE

Terminates a block following an $IF statement. If the comparison was true causes a skip around the following block. If the comparison was false the block is executed.

$SELSE

Same as $ELSE, except it generates a BR instead of a B.

$CASE VAR OF LAB1, LAB2, ...

Branches to the label in the list whose position corresponds to the value of the operand. (If the value in VAR is 0, then the program branches to LAB1; if the value is 1, then the program branches to LAB2, etc.) All labels in the list must be contained in the same GROM as the $CASE statement.

$GOTO LAB

Branches to the label. Label can be anywhere in the program since the $GOTO generates a long branch.

$CALL LAB

Calls the label as a subroutine.

The comparisons may take the following relations:

.H. .HE. .L. .GT. .GE. .LT. .LE.

.DH. .DHE. .DL. .DGT. .DGE. .DLT. .DLE.

.EQ. .NE. .AND.

.DEQ. .DNE. .DAND.

These relations are used in the logical expressions. The relations .AND. or .DAND. generate a CLOG of the GS and GD. Additionally the relations .H., .GT., .OVF., .CARRY. may be used without arguments to test bits in the STATUS byte. The negating prefix .NOT. may be used before the relation or first argument to reverse the sense of the test.

Individual bits may be tested by using the prefix .BITn or .BIT(IMM) before the first argument where n is a bit number from 0 to 7 or IMM is equated to a number from 0 to 7. Only the .EQ. and .NE. relations may be used and the second argument must be a 0 or 1.

TABLE A.1
MACRO EXPANSIONS

INSTRUCTION		$IF-THEN	$IF-GOTO	$REPEAT-$UNTIL	$WHILE
RELATION	INSTRUCTION	BS/BR	BS/BR	BS/BR	BS/BR
TEST OF BITS IN THE STATUS BYTE:					
.H.	H	BR	BS	BR	BR
.GT.	GT	BR	BS	BR	BR
.OVF.	OVF	BR	BS	BR	BR
.CARRY.	CARRY	BR	BS	BR	BR
TEST RELATION OF GD TO GS:					
.EQ.	CEQ	BR	BS	BR	BR
.NE.	CEQ	BS	BR	BS	BS
.H.	CH	BR	BS	BR	BR
.HE.	CHE	BR	BS	BR	BR
.L.	CHE	BS	BR	BS	BS
.GT.	CGT	BR	BS	BR	BR
.GE.	CGE	BR	BS	BR	BR
.LT.	CGE	BS	BR	BS	BS
.LE.	CGT	BS	BR	BS	BS
.AND.	CLOG	BR	BS	BR	BR
TEST RELATION OF GD TO 0:					
.EQ.	CZ	BR	BS	BR	BR
.NE.	CZ	BS	BR	BS	BS

All other relations of GD to 0 are tested as GD to GS, using 0 as the GS.

Following are several MACRO instructions with their graphics language equivalents:

1. $REPEAT
   $UNTIL GD .NOT. .H. GS

```
CH      GS,GD
BS      (Code following $REPEAT)
```

2. $REPEAT
   $UNTIL GD .HE. GS

```
CHE     GS,GD
BR      (Code following $REPEAT)
```

3. $REPEAT
   $UNTIL .NOT. GD .AND. GS

```
CLOG    GS,GD
BS      (Code following $REPEAT)
```

4. $REPEAT
   $UNTIL .OVF.

```
OVF
BR      (Code following $REPEAT)
```

```
 5.  $WHILE GD .NE. GS                    CEQ       GS,GD
                                          BS        (Code following $END)
     $END                                 B         $WHILE

 6.  $WHILE .BIT5 GD .EQ. 1               CLOG      >20,GD
                                          BS        (Code following $END)
     $END                                 B         $WHILE

 7.  $WHILE .CARRY.                       CARRY
                                          BR        (Code following $END)
     $END                                 B         $WHILE

 8.  $IF GD. .EQ. GS THEN                 CEQ       GS,GD
                                          BR        (Code following $ELSE)
     $ELSE                                B         (Code following $END)
     $END

 9.  $IF GD .DGE. GS THEN                 DCGE      GS,GD
                                          BR        (Code following $END)
     $END

10.  $IF GD .L. GS THEN                   CHE       GS,GD
                                          BS        (Code following $END)
     $END

11.  $IF GD .NOT. .AND. GS THEN           CLOG      GS,GD
                                          BS        (Code following $END)
     $END

12.  $IF GD .GT. GS GOTO LABEL            CGT       GS,GD
                                          BS        LABEL

13.  $IF .H. THEN                         H
                                          BR        (Code following $END)
     $END

14.  $IF BIT7 GD .NE. 0 THEN              CLOG      >80,GD
                                          BS        (Code following $END)
     $END

15.  $FOR GD = GS1 TO GS2 BY GS3          ST        GS1,GD
                                          B         $ + 6
                                $FOR+6    ADD       GS3, GD
                                $+6       CGT       GS2, GD
                                          BS        (Code following $END)
     $END                                 B         $FOR+6
```

```
16.  $FOR GD = 0 TO GS                    CLR     GD
                                          B       $+5
                                $FOR+5    INC     GD
                                $+5       CGT     GS, GD
                                          BS      (Code following $END)
     $END                                 B       $FOR+5

17.  $FOR GD = GS DOWNTO 0                ST      GS,GD
                                          B       $+5
                                $FOR+6    DEC     GD
                                $+5       CGE     GS,GD
                                          BR      (Code following $END)
     $END                                 B       $FOR+6

18.  $CASE VAR OF LAB1, LAB2              CASE    VAR
                                          BR      LAB1
                                          BR      LAB2

19.  $GOTO LABEL                          B       LABEL

20.  $CALL LABEL                          CALL    LABEL
```

APPENDIX B AUTOMATIC SPRITE MOTION

Any number of Sprites from 1 to 32 can be set into motion in such a way that the direction and speeds of each Sprite are constant, and independent of each other. The MOTION byte in the STATUS BLOCK, which is normally 0, is set by the programmer to the number of Sprites he wants to be governed by auto-motion. If set to N, Sprite (0) thru Sprite (N-1) in the Sprite Attribute List are set in motion. The Sprites are moved by updating the Y and X pixel positions for each one in the Sprite Attribute List.

A motion control block must be set up in VDP RAM prior to making the MOTION byte non-zero. This control block must begin at >780 in the VDP RAM. Four bytes are required for each Sprite to be controlled:

byte 1: velocity in the vertical direction;

byte 2: velocity in the horizontal direction;

bytes 3,4: reserved for system use.

The velocity bytes are scaled in such a way that a value of 1 causes the Sprite to move in that direction once every 16 frame refreshes (or 16/60 second, about ¼ second). A value of 16 in a velocity byte causes a movement of one pixel every one-sixtieth of a second, or 60 pps. A positive Y velocity causes downward motion, a positive X velocity causes horizontal motion to the right. As an example, if the first two bytes are 1 and 8, then every 16 frame refreshes the Sprite will move 1 pixel down and 8 pixels to the right. The motion will appear to be continuous.

For a complete example of Sprite auto-motion, see sample program "SPRITES" on the following page.

```
                    1          TITLE SPRITES
                    2  *  STATUS BLOCK LOCATION TELLING NUMBER OF MOVING SPRITES
      007A          3  MPC     EQU  >7A
                    4  *                      COLOR NUMBER OF BLACK
      0001          5  BLACK   EQU  1
                    6  ************************************************************
                    7  *       '                  MAIN PROGRAM
                    8  ************************************************************
                    9  ***     TELLS BEGINNING LOCATION OF OBJECT CODE (>6000)    *
                   10          GROM 3
                   11          ORG  0
                   12  ***                          HEADER BLOCK                  *
6000 AA0101        13          DATA >AA,1,1
6003 000000        14          DATA 0,0,0
6006 6010          15          DATA #PROG1
6008 000000        16          DATA 0,0,0
600B 000000        17          DATA 0,0,0
600E 0000          18          DATA 0,0
6010 0000          19  PROG1   DATA 0,0
6012 601C          20          DATA #START
6014 075350        21          DATA 7,:SPRITES:
6017 524954
601A 4553
                   22  ***              STORE NUMBER OF SPRITES IN MPC           *
601C BE7A20        23  START   ST   32,@MPC       *32 SPRITES ALLOWED TO MOVE
                   24  ***        LOAD COLOR TABLE THAT CONTAINS THE SPACE       *
601F BEA384        25          ST   >01,RAM(>384)      *COLOR SPACE BLACK
6022 01
                   26  ***     LOAD VDP REGISTER 1 FOR DOUBLE-SIZED SPRITES      *
6023 390001        27          MOVE 1 FROM ROM(#VDPREG) TO VDP(1)
6026 016045
                   28  ***              ESTABLISH SPRITE ATTRIBUTE BLOCK         *
6029 310080        29          MOVE 128 FROM ROM(#SALINT) TO RAM(>300)
602C A30060
602F 46
                   30  ***              ESTABLISH SPRITE DESCRIPTOR BLOCK
6030 310080        31          MOVE 128 FROM ROM(#SHAPE) TO RAM(>400)
6033 A40061
6036 46
                   32  ***              ESTABLISH SPRITE VELOCITY BLOCK          *
6037 310080        33          MOVE 128 FROM ROM(#SMOTAB) TO RAM(>780)
603A A78060
603D C6
                   34  ***             MAKE ALL PATTERN NAME TABLE BLANK         *
603E 0720          35          ALL  >20
6040 0401          36          BACK BLACK          *BORDER IS BLACK
6042 056042        37          B    $
6045 62            38  VDPREG  DATA >62
```

```
                    40  ************************************************************
                    41  *            SPRITE ATTRIBUTE LIST INITIALIZATIONS
                    42  ************************************************************
6046 000080         43  SALINT DATA >00,>00,>80,>2,>06,>08,>84,>3
6049 020608
604C 8403
604E 0C1088         44         DATA >0C,>10,>88,>4,>12,>18,>8C,>5
6051 041218
6054 8C05
6056 182080         45         DATA >18,>20,>80,>6,>1E,>28,>84,>7
6059 061E28
605C 8407
605E 243088         46         DATA >24,>30,>88,>8,>2A,>38,>8C,>9
6061 082A38
6064 8C09
6066 304080         47         DATA >30,>40,>80,>A,>36,>48,>84,>B
6069 0A3648
606C 840B
606E 3C5088         48         DATA >3C,>50,>88,>C,>42,>58,>8C,>D
6071 0C4258
6074 8C0D
6076 486080         49         DATA >48,>60,>80,>E,>4E,>68,>84,>F
6079 0E4E68
607C 840F
607E 547088         50         DATA >54,>70,>88,>2,>5A,>78,>8C,>3
6081 025A78
6084 8C03
6086 608080         51         DATA >60,>80,>80,>4,>66,>88,>84,>5
6089 046688
608C 8405
608E 6C9088         52         DATA >6C,>90,>88,>6,>72,>98,>8C,>7
6091 067298
6094 8C07
6096 78A080         53         DATA >78,>A0,>80,>8,>7E,>A8,>84,>9
6099 087EA8
609C 8409
609E 84B088         54         DATA >84,>B0,>88,>A,>8A,>B8,>8C,>B
60A1 0A8AB8
60A4 8C0B
60A6 90C080         55         DATA >90,>C0,>80,>C,>96,>C8,>84,>D
60A9 0C96C8
60AC 840D
60AE 9CD088         56         DATA >9C,>D0,>88,>E,>A2,>D8,>8C,>F
60B1 0EA2D8
60B4 8C0F
60B6 A8E080         57         DATA >A8,>E0,>80,>2,>AE,>E8,>84,>4
60B9 02AEE8
60BC 8404
60BE B4F088         58         DATA >B4,>F0,>88,>6,>BA,>F8,>8C,>8
60C1 06BAF8
60C4 8C08
```

```
                    60  ********************************************************
                    61  *                        SPRITE MOTION TABLE
                    62  ********************************************************
60C6 021000         63  SMOTAB DATA 2,16,0,0,2,14,0,0
60C9 00020E
60CC 0000
60CE 020C00         64         DATA 2,12,0,0,2,10,0,0
60D1 00020A
60D4 0000
60D6 020800         65         DATA 2,8,0,0,2,6,0,0
60D9 000206
60DC 0000
60DE 020400         66         DATA 2,4,0,0,2,2,0,0
60E1 000202
60E4 0000
60E6 040200         67         DATA 4,2,0,0,6,2,0,0
60E9 000602
60EC 0000
60EE 080200         68         DATA 8,2,0,0,10,2,0,0
60F1 000A02
60F4 0000
60F6 0C0200         69         DATA 12,2,0,0,14,2,0,0
60F9 000E02
60FC 0000
60FE 100200         70         DATA 16,2,0,0,8,0,0,0
6101 000800
6104 0000
6106 00F800         71         DATA 0,-8,0,0,-2,-16,0,0
6109 00FEF0
610C 0000
610E FEF200         72         DATA -2,-14,0,0,-2,-12,0,0
6111 00FEF4
6114 0000
6116 FEF600         73         DATA -2,-10,0,0,-2,-8,0,0
6119 00FEF8
611C 0000
611E FEFA00         74         DATA -2,-6,0,0,-2,-4,0,0
6121 00FEFC
6124 0000
6126 FEFE00         75         DATA -2,-2,0,0,-4,-2,0,0
6129 00FCFE
612C 0000
612E FAFE00         76         DATA -6,-2,0,0,-8,-2,0,0
6131 00F8FE
6134 0000
6136 F6FE00         77         DATA -10,-2,0,0,-12,-2,0,0
6139 00F4FE
613C 0000
613E F2FE00         78         DATA -14,-2,0,0,-16,-2,0,0
6141 00F0FE
6144 0000
```

```
                     80  ************************************************************
                     81  *                  SPRITE DESCRIPTOR BLOCKS
                     82  *              (SQUARE, DIAMOND, CIRCLE, TRIANGLE)
                     83  ************************************************************
6146 FFFFC0          84  SHAPE  DATA >FF,>FF,>C0,>C0,>C0,>C0,>C0,>C0              SQUAR
6149 C0C0C0
614C C0C0
614E C0C0C0          85         DATA >C0,>C0,>C0,>C0,>C0,>C0,>FF,>FF
6151 C0C0C0
6154 FFFF
6156 FFFF03          86         DATA >FF,>FF,>03,>03,>03,>03,>03,>03
6159 030303
615C 0303
615E 030303          87         DATA >03,>03,>03,>03,>03,>03,>FF,>FF
6161 030303
6164 FFFF
6166 010306          88         DATA >01,>03,>06,>0C,>18,>30,>60,>C0              DIAMO
6169 0C1830
616C 60C0
616E C06030          89         DATA >C0,>60,>30,>18,>0C,>06,>03,>01
6171 180C06
6174 0301
6176 80C060          90         DATA >80,>C0,>60,>30,>18,>0C,>06,>03
6179 30180C
617C 0603
617E 03060C          91         DATA >03,>06,>0C,>18,>30,>60,>C0,>80
6181 183060
6184 C080
6186 071F3C          92         DATA >07,>1F,>3C,>70,>60,>E0,>C0,>C0              CIR..
6189 7060E0
618C C0C0
618E C0C0E0          93         DATA >C0,>C0,>E0,>60,>70,>3C,>1F,>07
6191 60703C
6194 1F07
6196 E0F83C          94         DATA >E0,>F8,>3C,>0E,>06,>07,>03,>03
6199 0E0607
619C 0303
619E 030307          95         DATA >03,>03,>07,>06,>0E,>3C,>F8,>E0
61A1 060E3C
61A4 F8E0
61A6 010103          96         DATA >01,>01,>03,>03,>06,>06,>0C,>0C              TRIANGL
61A9 030606
61AC 0C0C
61AE 181830          97         DATA >18,>18,>30,>30,>60,>60,>FF,>FF
61B1 306060
61B4 FFFF
61B6 8080C0          98         DATA >80,>80,>C0,>C0,>60,>60,>30,>30
61B9 C06060
61BC 3030
61BE 18180C          99         DATA >18,>18,>0C,>0C,>06,>06,>FF,>FF
61C1 0C0606
61C4 FFFF
                    100         END
```

ERRORS= 0

LENGTH= 454 (>01C6)

APPENDIX C AUTO-SOUND INSTRUCTION

The sound instruction allows the programmer to control the Sound Generator Chip (SGC) in the system console by means of a pre-defined table in GROM, or VDP RAM. Sound output is controlled by the table and the VDP interrupt service routine. A control byte at the end of the table can tell the interpreter to end sound output, or can cause control to loop back up in the table.

Table Format

The format of the table is the same regardless of where it resides. The table consists of a series of blocks, each of which contains a series of bytes which are directly output to the SGC chip. The exact format of each block is:

(block size in bytes)
byte 1 to output to SGC;
byte 2
.
.
byte N_1
Interrupt count (unsigned)

Since the VDP generates 60 interrupts per second, the interrupt count is expressed in units of one-sixtieth of a second. When the I/O instruction is called, upon the next occurring VDP interrupt, the first block of bytes is output to the SGC chip. The interpreter then waits for the requested number of interrupts (for example, if interrupt counts are 1, every

interrupt causes the next block to be output). Remember that interpretation of GPL continues normally while the SGC control is enabled.

The sound control can be terminated by using an interrupt count of 0 in the last block of the table. Alternatively, a primitive looping control is provided by using a block whose first byte is 0, and the next 2 bytes indicate an address in the same memory space of the next sound block to use. If the first byte is hexidecimal FF, the next two bytes indicate an address in the other memory space. These allow switching sound lists from GROM to VDP or VDP to GROM. By making this the beginning of the entire table, the sound sequence can be made to repeat indefinitely.

To initiate sound use the I/O instruction:

I/O GS, 0 for list in GROM

or I/O GS,1 for list in VDP RAM, e.g. I/O @FAC, 1

The GS points to two-byte block in CPU RAM which contains the address of the sound list.

GPL can also check for completion of an executing sound list by testing whether location >83CE (>CE in GPL) in CPU RAM is equal to 0 (this byte is a down-counter and is 0 only after table-driven execution is complete). Additionally, the address of the sound block currently executing is in CPU RAM locations >83CC and >83CD.

Executing a sound list while table-driven sound control is already in progress (from a previous sound list) causes the old

sound control to be totally supplanted by the new sound instruction.

Sound Generator CHIP (SGC) Control Summary

The SGC has 3 tone (square wave) generators - 0, 1, and 2 - all of which can be working simultaneously or in any combination. The frequency (pitch) and attenuation (volume) of each generator can be independently controlled. In addition, there is a noise generator which can output white or periodic noise.

Attenuation Control (for generators 0,1,2 or 3)

One byte must be transmitted to the SGC:

```
+  +  +  +  +  +  +  +  +
/1 /REG# /1 /    A      /
```

REG# = register number (0,1,2,3);

A = attenuation/2

(e.g. A=0000 = 0 db = highest volume;
A=1000 = 16 db = medium volume;
A=1111 = 30 db = off.)

examples: 1 10 1 0000: turn on gen. #2 to highest volume;
1 11 1 1111: turn off noise generator (#3).

You should not use all three tone generators at maximum attentuation at once.

Frequency Control (for generators 0,1,2)

Two bytes must be transmitted to the SGC to control the frequency of a given register. To compute the number of counts from the frequency f, use:

N = 111860 / f ;

```
      byte 1:                          byte 2:
+ + + + + + + + +     + + + + + + + + +
/1 /REG# /   N (ls 4 bits) /   / 0 0 /  N (ms 6 bits)/
```

Note that N must be split up into its least significant 4 bits and most significant 6 bits (10 bits total).

The lowest frequency possible is 110 Hz and the highest is 55,938 Hz.

Noise Control

One byte must be transmitted to the SGC:

```
+ + + + + + + + +
/1  1  1  0 /0 /T /  S  /
```

T = 0 for white, 1 for periodic noise;

S = Shift rate (0,1,2,3) =frequency center of noise.

S = 3 = frequency dependent on the frequency of tone generator 3.

For more information on controlling the SGC, see the TMS9919 SGC Specification.

```
Creates a Falling Sound (High to Low)
SOUND  EQU  >00
DTEMP1 EQU  >02
TR     EQU  >79
MUSIC  EQU  >400
      Move 8 bytes from ROM(#DROP) to RAM(MUSIC)
      DST  >0039, @DTMP1    >39 = Highest possible frequency
                                  @DTEMP1 = 2-byte temp area
```

```
      DST   MUSIC,@SOUND      (Music is a constant of >400 --
                               could be anywhere in RAM
B01   DST   @DTEMP1, RAM(MUSIC+1)
      DSRC  RAM(MUSIC+1),4
      SRL   RAM(MUSIC+1),4
      OR    &10000000,RAM(MUSIC+1)
      I/O   @SOUND,1           @SOUND = 2-byte area for ADDR
      CLR   @TR                @TR = Timing register ( >79)
B02   CZ    @TR
      BS    B02
      DINC  @DTEMP1
      DCGE  >0200,@DTEMP1      >0200 = Lowest frequency played
      BR    B01
      DST   #ENDROP,@SOUND     *Turns sound off
      I/O   @SOUND, 0
      B     $
DROP  DATA 3,>00, >00, >00, >92,1
ENDROP DATA 1, >9F, 0
```

A similar routine could be implemented to create a rising sound by storing a low frequency in DTEMP1 to begin with, do a DDEC to DTEMP1 and a compare low with a high frequency value.

```
CREATES AN EXPLOSION SOUND
      DST     #EXPL,@SOUND
      I/O     @SOUND,0
      B       $
```

```
EXPL  DATA 2, >E0, >F2, 5
      DATA 2, >E0, >F0, 18
      DATA 2, >E0, >F1, 16
      DATA 2, >E0, >F2, 14
      DATA 2, >E0, >F3, 12
      DATA 2, >E0, >F4, 10
      DATA 2, >E1, >F5, 9
      DATA 2, >E1, >F6, 8
      DATA 2, >E1, >F7, 7
      DATA 2, >E1, >F8, 6
      DATA 2, >E1, >F9, 5
      DATA 2, >E2, >FA, 4
      DATA 2, >E2, >FB, 3
      DATA 2, >E2, >FC, 2
      DATA 2, >E2, >FD, 1
      DATA 2, >E2, >FE, 1, 1, >FF,0
```

APPENDIX D HANDSET/KEYBOARD INTERFACE

As mentioned in Section 4 of this manual, the SCAN instruction is used to poll the state of the handsets and keyboards on the system. The byte KEYBRD in the STATUS BLOCK is used by the SCAN instruction to determine which peripheral device to look at, as well as how to interpret the results.

Presently, the following peripherals are supported by the SCAN instruction:

- 40-KEY KEYBOARD (KEYBRD = 0):

  When scanning this keyboard, only the bytes KEY, and the COND bit are affected. The layout of the keyboard and the codes returned by each key are shown in Figures D.1 and D.1.A. If more than one key is depressed at a given time, only one key will be read.

- REMOTE HANDSETS (KEYBRD = 0,1,2,3,4)

  See Figures D.2, and D.2.A for keycodes assigned to the Remote Handsets. Note that if KEYBRD = 0, Handsets 1 and 2 are assumed to be adjacent to each other and thus simulate the 40-key keyboard. If KEYBRD = 1,2,3 or 4, the correspondingly numbered handset is scanned; in addition, the joystick is scanned, and each of JOYY and JOYX will get a value returned in them ranging from -7 to +7, depending upon the amount of deviation from the neutral position in the Y and X axes respectively.

- REMOTE KEYBOARD (KEYBRD = 5)

  Remote handsets 3 and 4 can be mapped into a 40-key keyboard in the same manner as handsets 1 and 2.

- WIRED HANDSETS (KEYBRD = 1,2):

  See Table D.3 for keycodes assigned to this type of handset. The joystick behaves similarly to the remote handsets except that the range of JOYY and JOYX is limited to values of -4,0, or +4. These values were chosen to make the remote and wired joysticks as compatible as possible. Note there is a pushbutton mounted on the joystick. This button is electrically and logically the same as the key corresponding to keycode >C. The console keyboard may be used to simulate two 20-key keypads for the wired handsets. Note that if the joystick pushbutton is depressed, it will always be recognized, as it has the highest priority. The depression of more than 2 keys causes undefined values to be returned.

Since the GPL program is not immediately alerted that the state of a handset has altered, it is necessary to regularly scan the handsets, if input from them is desired. The COND bit in the STATUS byte is set only the first time a given key is found depressed. If the same key is found depressed on successive scans, the successive calls to SCAN reads the keycode properly, but resets the COND bit. Thus in applications like the above, where we wish to recognize fresh keystrokes only, the following code sequence can be used:

```
LOOP1   SCAN
        BR LOOP1
```

The above code causes GPL to loop until a fresh keystroke is seen.

In order to debounce the Fire button a routine must be implemented to make sure it has been lifted before it is detected as being down again. An example of this routine would be:

```
SCANIT      SCAN
            $IF @KEY .EQ. @OLDKEY THEN
            B     SCANIT
            $ELSE
            ST    @KEY, @OLDKEY
            (operate on KEY)
            $END
            B     SCANIT
```

FIGURE D.1

CONSOLE KEYBOARD

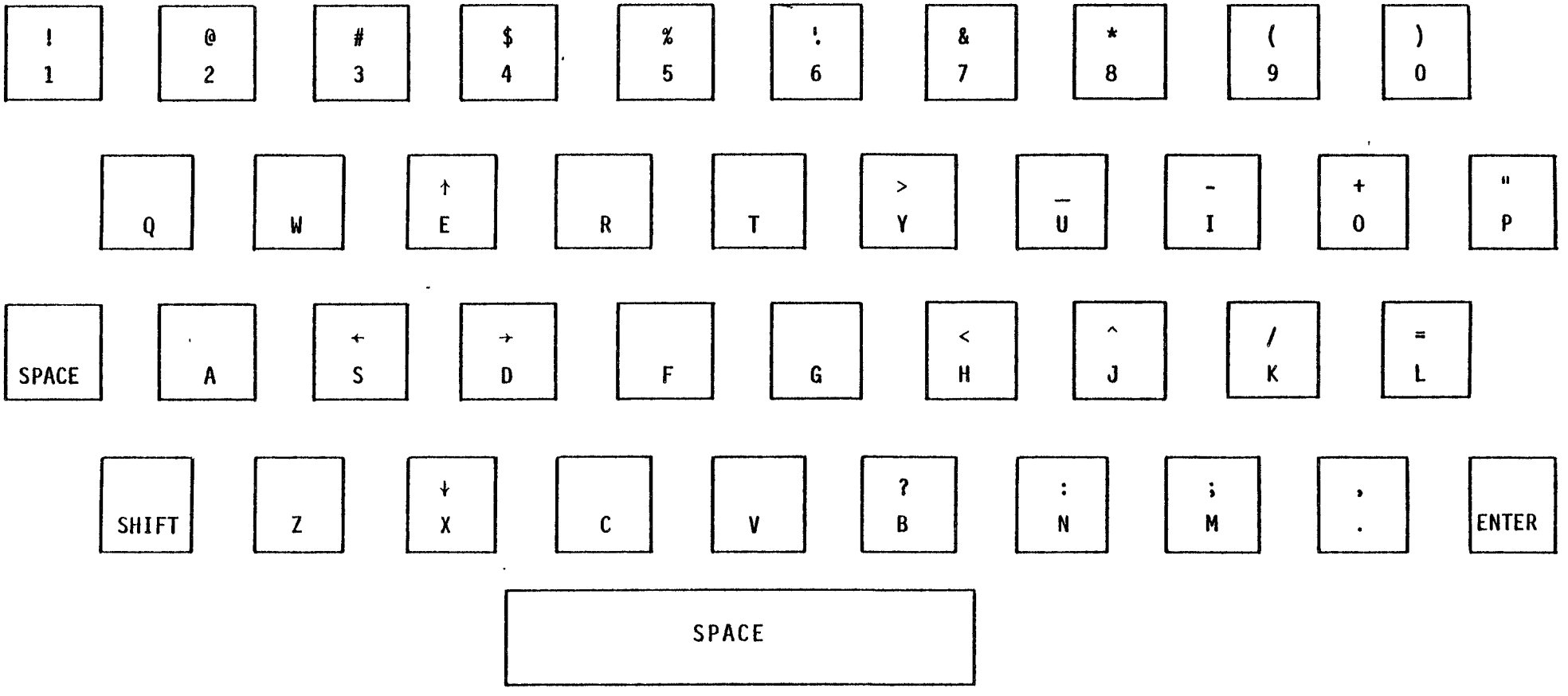
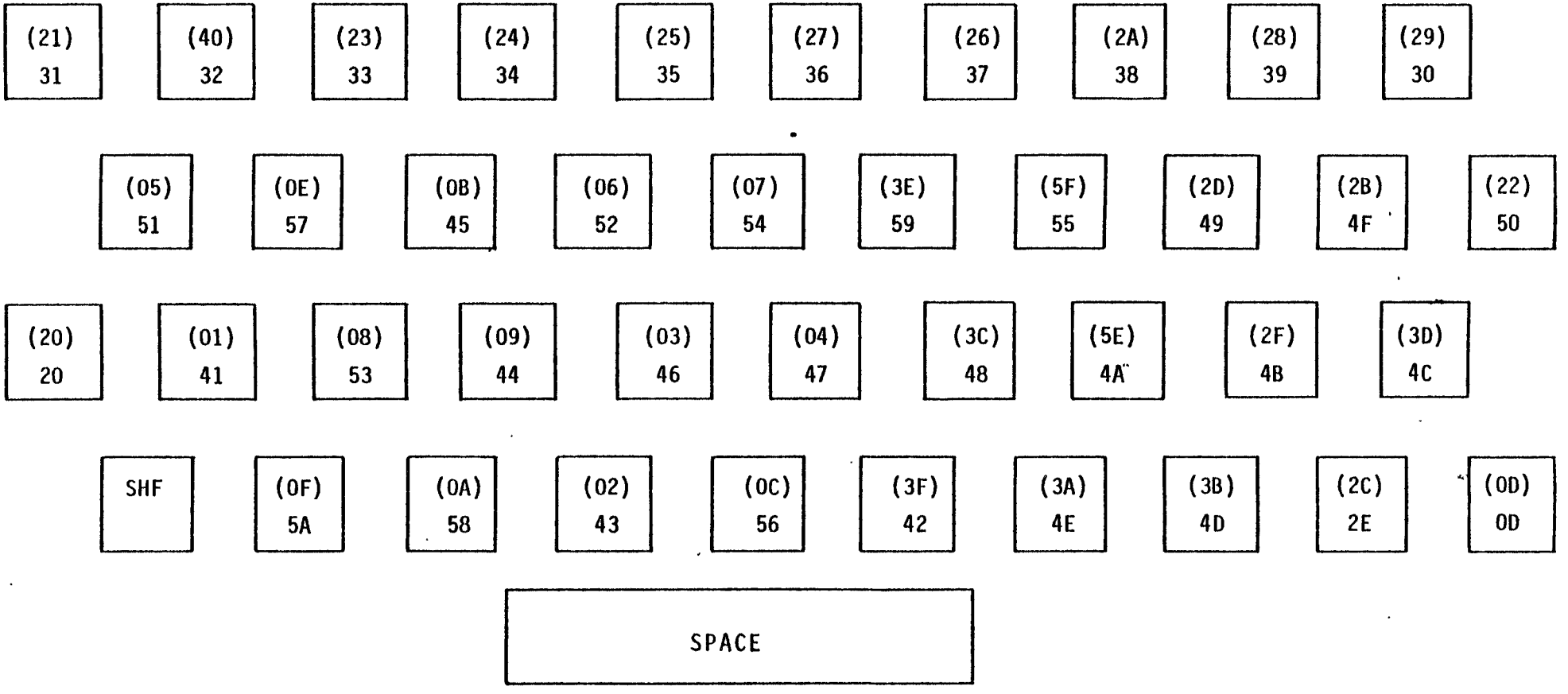
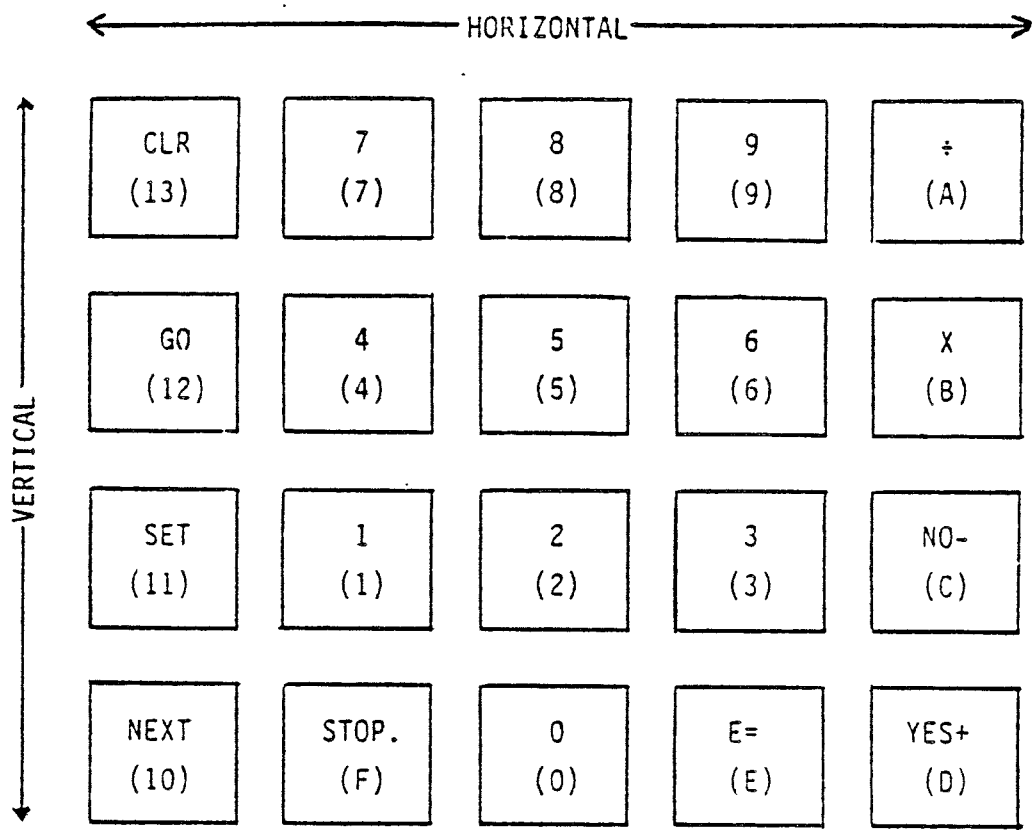


FIGURE D.1.A

CONSOLE KEYBOARD HEXDECIMAL-CODE ASSIGNMENTS



HANDHELD UNIT KEYBOARD



NOTE 1: Hexidecimal numbers in parenthesis correspond to keycodes returned by console system software.

FIGURE D.2.A

CONSOLE KEYBOARD MAPPED AS TWO HANDHELD UNITS

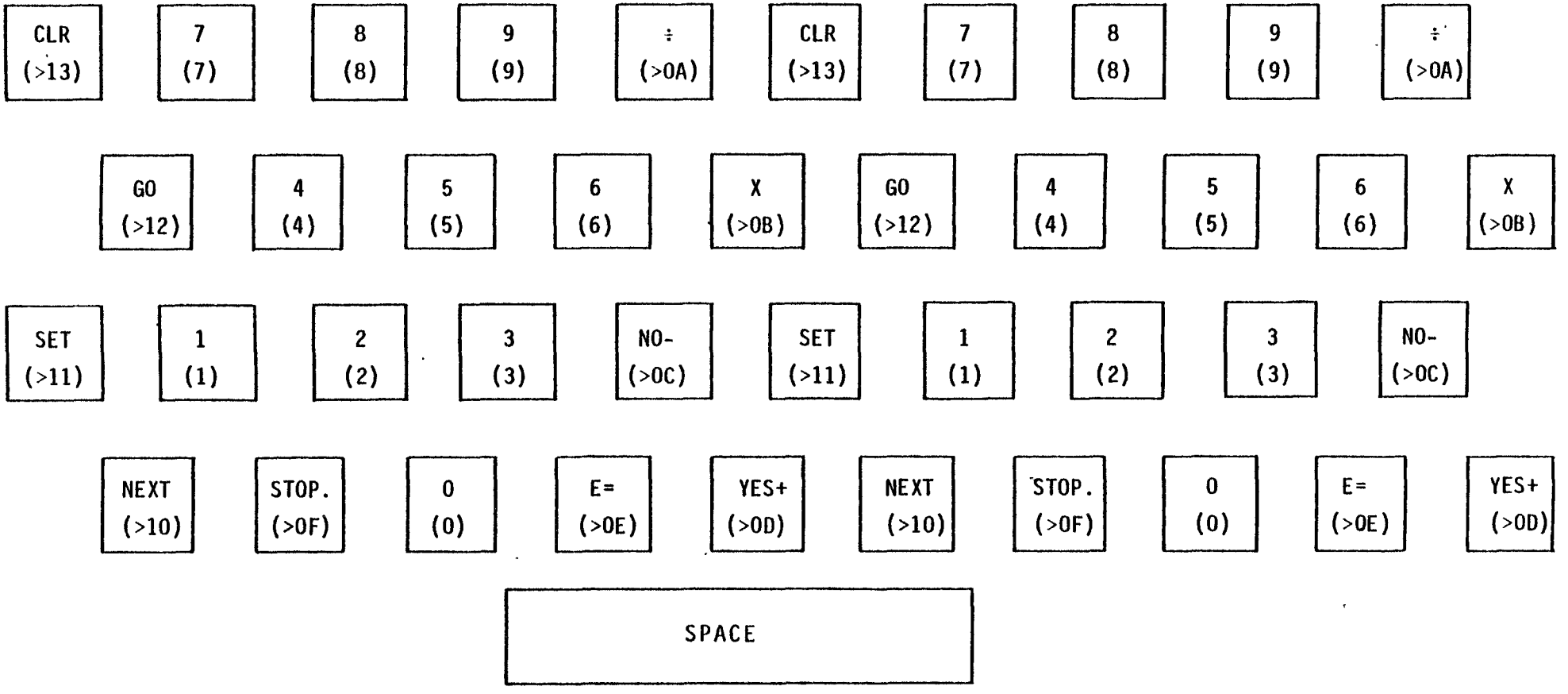


TABLE D.3

JOYSTICK CODES

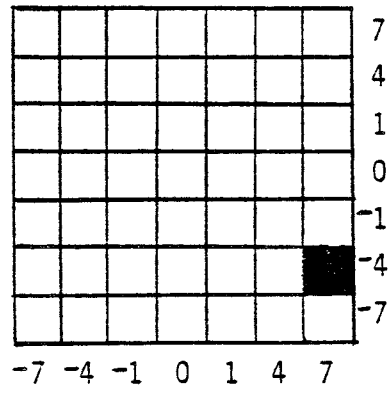
HEXIDECIMAL CODE	X POSITION (HORIZONTAL)	Y POSITION (VERTICAL)
7	Full Right	Full Up
4*	Medium Right	Medium Up
1	Near Right	Near Up
0	Center Off	Center Off
>FF (-1)	Near Left	Near Down
>FC* (-4)	Medium Left	Medium Down
>F9 (-7)	Full Left	Full Down
>F8	Illegal	Illegal

*These codes to be returned if joystick has only single bit resolution in any direction.

JOYSTICK CODES

Example

Full Right and Medium Down 7,>FC



APPENDIX E COINCIDENCE DETECTION

The VDP provides a bit in the VDP status register that is set whenever any Sprites are in coincidence with one another (in this instance, coincidence means that they overlap by at least one pixel of foreground). From GPL, this bit is most easily checked by the instruction:

```
CLOG >20,@VSTAT
```

The VDP Status byte in CPU RAM is copied from the VDP Status register every frame interrupt; the third bit is the sprite coincidence bit.

The COINC instruction in GPL allows the user to check for coincidence between any 2 objects. These may be 2 Sprites, a Sprite and another object, or any 2 generalized objects. The strict definition of coincidence can be dictated by a bit table the user must provide in GROM; one might desire coincidence to be true when the objects just touch, or a one dot overlap may be required. Or perhaps coincidence may be true only when object 1 overlaps object 2 exactly.

Coincidence statements must be followed by a one-byte mapping value, plus a 2-byte address pointing to a table in GROM. Mapping = 0 gives the highest resolution coincidence checking, but requires the largest table. Mapping = 1 yields a table of ¼ the size; however, coincidence errors of ± 2 pixels are possible. Mapping = 2 yields one-sixteenth the table size but can have errors of ±4 pixels.

Let an "object type" be a set of identical objects. Then 2 sprites which have identical dot patterns are actually of the

same object type. To detect coincidence between objects of 2 types (may be the same type) a unique table for this type combination is necessary.

Coincidence screening is done on 2 levels. The first, range checking, involves looking at the distance between the objects as well as their individual dimensions (in pixels). If they are out of range, coincidence is terminated by resetting the condition bit and terminating. If in range, a table lookup is used to determine coincidence. The delta-y and delta-x are found; using them as indices, a bit is read from the table. If it is a 1, coincidence is true, and the cond bit is set; otherwise coincidence is false. Remember that a unique table is required for each combination of object types; e.g. for 2 object types, say girls and boys, three tables are required for complete coincidence detection:

- girls : girls
- girls : boys
- boys : boys

Coincidence must always be called with its arguments in the same order that the table was constructed for.

CONSTRUCTING COINCIDENCE TABLES FOR MAPPING = 0

Let V_1 and H_1 be the dimensions of object type 1 in pixels (for irregular objects, these are the dimensions of the circumscribing rectangle). Likewise, let V_2 and H_2 be the vertical and horizontal dimensions of object type 2. Let Y_1, X_1

be the dot position of object type one, and Y_2, X_2 the same for object type 2 (the origin is at the top left of the TV screen; object position is the position of the top left dot of the circumscribing rectangle).

Then let DY be $Y_1 - Y_2$, and DX be $X_1 - X_2$. Then DX, DY, V_1, H_1, V_2 , H_2, and the object shapes completely specify whether coincidence is true or not. See Figure E.1 (page E-6).

Imagine object 2 fixed in one place on the screen, and object 1 mobile. It is not hard to see that after object 1 is more than H_1 dots to the left of object 2, coincidence is no longer possible. Similarly after it is more than H_2 dots to the right of object 1, coincidence is not possible. Applying the same logic to the vertical axis, we arrive at the rules for the range check:

$$-H_1 \text{ .LE. } DX \text{ .LE. } H_2$$

$$-V_1 \text{ .LE. } DY \text{ .LE. } V_2 \text{ ;}$$

After finding DX and DY to be within this range, they are used to compute a unique bit index into the bit table:

$$INDEX = (DX + H_1) + (DY + V_1)*(H_1 + H_2 + 1).$$

The bit table is most easily visualized as seen in Figure E.2 (page E-6). This table is then encoded by bytes; starting at the top left, working to the right, then going to the second row and repeating.

The easiest way to manually construct a bit-table is to draw object 2 on graph paper, letting each square represent a pixel. Then cut out object 1 from another sheet. Starting with object 1 at the top left corner of object 2 (circumscribing

rectangles just touching), move it to the right, generating a 1 or a 0 each movement. An example is shown in Figure E.3 (page E-7). Then repeat the same with object 1 down by one dot. This technique is repeated through the row in which object 1 is just touching the bottom of object 2.

The table in GROM requires a 4 byte header. The exact structure is:

```
(label)     DATA   (vertical size of table less 1)
            DATA   (horiz.    "    "    "    "  ")
            DATA   (V1)
            DATA   (H1)
            DATA   (bits, grouped in 8's)
```

In the example from Figure E.3, we have

```
EXAMP DATA 4
      DATA 5
      DATA 2
      DATA 2
      DATA >73,>FF,>FF,>FF          (2 scrap bits at end);
```

HIGHER MAPPING VALUES

By specifying MAPPING values greater than 0, one can make the bit table more compact; however, accuracy of detection suffers. In the case of MAPPING =1, instead of a one-to-one mapping of DX,DY pairs onto bits of the table, 4 combinations of DX,DY all map into one bit. Thus the bit table is smaller, but it is necessary to lose detection resolution.

To construct le for MAPPING = 1, construct a magnification 0 t first (see Figure E.4 (page E-8); note that the objects ot shown in this example). Then draw 2x2 boxes on the table, ting at the box corresponding to DX=0,1, DY=0,1. A new tablthen constructed; each group of 4 bits in a box are compacted to one bit. Note that:

-small 2x1 bon the edges reduce a single bit;

-the single bould reflect the predominant value of the cluster;

-in the cases ones and 2 zeros, the user has a choice, depending one wants coincidence to predominate.

To make a tal MAPPING =2, do the same process, but make 4x4 boxes on tlimensional table (the first box is at DX = 0,1,2,3, DY = 0,). Note that resolution of coincidence suffers greatly becaow we have 16 combinations of DX and DY mapping into one bi

The concept PPING lends itself well to changing the magnifications of s. If a MAPPING = 0 table is designed for two mag 0 sprithe same table can be used for checking coincidence when ma by merely calling COINC with MAPPING= 1. Remember though coincidence resolution goes down.

Note that <u>any</u> t can be used in coincidence detection; all that is requir that a Y,X byte pair exists in the VDP RAM for that objectte also that the object can be purely fictitious as far a TV screen is concerned.

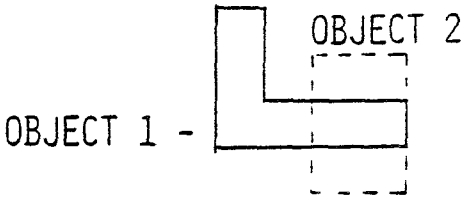


FIGURE E.1

DY=-1
DX=-2
V1= 3
H1= 4
V2= 3
H2= 2

	$-H_1$	$-H_1+1$	$-H_1+2$			-1	0	+1				H_2-1	H_2
$-V_1$	0	0	1	1	0	1	0	1	1	0	0	1	1
$-V_1+1$	1	1	1	1	1	0	0	0	0	0	0	1	1
$-V_1+2$	1	0	0										
:	0	0		1									
:	0	0			1								
-1	0	0				0							
0	0	0					1						
+1	0	0						1					
:	1	0							0				
:	1	1								1			
V_2-1	0	1									1		
V_2	0	1										0	

FIGURE E.2

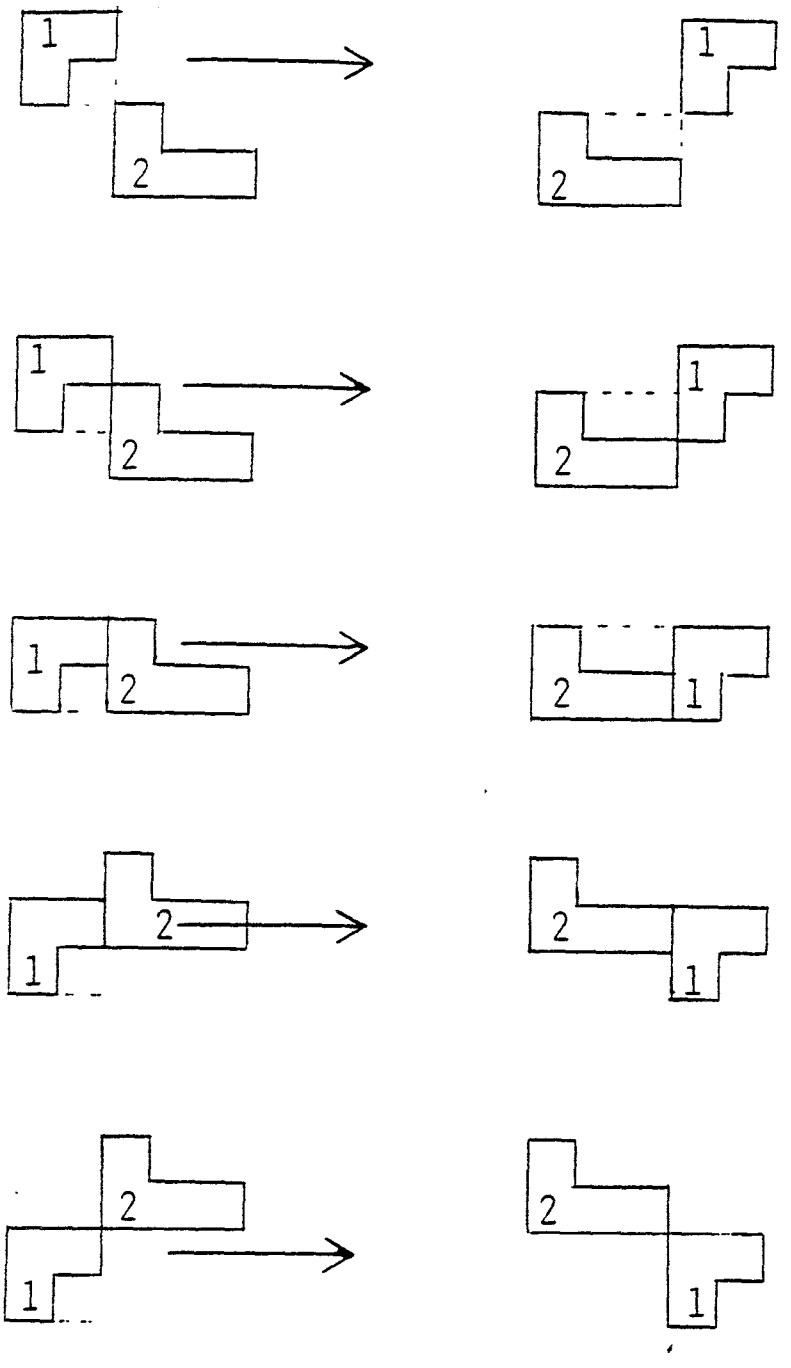
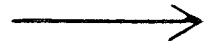


FIGURE E.3

DX

	-2	-1	0	+1	+2	+3
-2	0	1	1	1	0	0
-1	1	1	1	1	1	1
0	1	1	1	1	1	1
+1	1	1	1	1	1	1
+2	1	1	1	1	1	1

	-3	-2	-1	0	1	2	3	4	5	
-6	0	0	1	1	0	1	0	1	1	0
-5	0	1	1	1	0	1	1	0	0	1
-4	0	0	1	1	1	0	0	0	0	0
-3	1	0	1	1	1	1	0	1	1	1
-2	1	1	1	1	1	0	1	1	1	1
-1	1	1	1	0	1	0	1	1	1	1
0	0	1	0	0	1	0	1	1	1	1
1	1	0	0	1	1	1	1	1	1	1
2	0	0	0	0	0	1	1	1	1	0
3	1	0	1	1	1	1	1	1	0	1



0	1	1	1	1	1
1	1	1	0	1	1
1	1	1	1	1	1
1	0	1	1	1	1
1	0	1	1	1	1

FIGURE E.4

```
TABLE:  DATA 4
        DATA 5
        DATA 6
        DATA 3
        DATA >7F, >BF,
             >EF, >BC
```

APPENDIX F I/O INSTRUCTION

The I/O instruction is used to control a variety of input-output devices including cassettes, speech, sound, and CRU.

The format of the I/O instruction is:

```
I/O  GS, IMM
```

where

GS is the address of a list whose format depends on the value of IMM.

IMM specifies what type of input-output currently supported values of IMM are:

0 = Sound in GROM

1 = Sound in VDP RAM

2 = CRU input

3 = CRU output

4 = Cassette write

5 = Cassette read

6 = Cassette verify

The format of the list specified by GS for sound I/O instructions is given in Appendix C.

The format of the lists for CRU output is the same. GS points to a 4 byte block in CPU RAM. The format of the block is:

bytes 0 and 1 - CRU base address. The interpreter will double this for you since the 9901 ignores the least significant bit of the base register.

byte 2 - The number of bits to input or output (1-16)

byte 3 - A pointer to a one or two byte area in CPU RAM to write from or read to. If the number of bits to read or write is greater than 8 then this address must be even.

The CRU data to be written should be right justified in the byte or word. The least significant bit will output to or input from the CRU address specified by the CRU base address. Subsequent bits will come from or go to sequentially higher CRU addresses. If the CRU input reads less than 8 bits, the unused bits in the byte are reset to zero. If the CRU input reads less than 16 but more than 8 bits, the unused bits in the word will be reset to zero.

The three different cassette I/O instructions use the same list format. This list must be in CPU RAM.

bytes 0, 1 - are the length of the data transfer (or the number of bytes to verify). This length is rounded up to the nearest multiple of 64.

bytes 2, 3 - are the source or destination address in VDP RAM or the address of the bytes to verify the tape.

The read and write instructions physically perform I/O to the cassette. The verify instruction will read a tape and compare it, byte for byte, against what is in the specified VDP RAM area. It will set the status in CPU RAM location >7C if any differences are detected.

The I/O instructions for cassette will not generally be used by the application programs. There is a cassette program written in GPL that should be used by the application programs. This program will uniformly request the user to perform certain manual operations necessary to the operation of the cassette. This cassette program is described in Appendix I.

APPENDIX G TEXT AND MULTICOLOR MODE

When the Text Mode bit (bit 4) in VDP register #1 is set, 40-character mode is selected. The screen is 40 x 24 characters with each character being 6 x 8 dots. The Pattern Name Table is now 960 bytes long and is in locations 0 - >3BF in VDP RAM. Each byte in the Pattern Name Table corresponds to a pattern position on the screen (0 - >27, first row; >28 - >4F, second row; etc.). The pattern numbers are still 0 - 255, corresponding to VDP >800 ->FFF, but in text mode the last 2 bits of each byte in the patterns are ignored, making the 6 x 8 dot patterns. The only means of changing the screen in text mode is to write the pattern numbers to the Pattern Name Table position. There is not a color table to use with text mode. The only way to give color to the patterns is by loading VDP (7) with the foreground/background combination desired.

When the MCMD bit (bit 3) in VDP register #1 is set, the multicolor mode is selected. Each 8 x 8 dot pattern on the screen is now divided into four quadrants (4 x 4 dots each). Each quadrant must be given a nybble assignment in the pattern generator block before you can use multicolor mode correctly. The nybbles used in the pattern generator block are from RAM 800 thru DFF. The nybble assignments are made with a format statement as follows:

```
HOME
FMT     4(' >00, >01, >02... >IF'), 4(' >20, >21, >22... >3F'),
        4(' >40, >41, >42... >5F'), 4(' >60, >61, >62... >7F'),
        4(' >80, >81, >82... >9F')  4(' >A0, >A1, >A2... >BF')
```

This format statement puts 24 rows of 32 characters in the Pattern Name Table (VDP RAM >0 - >2FF), but it puts 48 rows of 64 characters on the screen (each byte in the PNT corresponds to a 2 x 2 block of 4 x 4 dots on the screen) VDP RAM locations 0, >20, >40, and >60 all have the value 0, but RAM (0) uses the nybbles at >800 and >801; RAM (>20) uses the nybbles at >802 and >803; RAM (>40) uses the nybbles and >804 and >805; RAM (>60) uses the nybbles at >806 and >807.

The value in each byte of the PNT is the number of the character in the Pattern Generator. Although each character in the Pattern Generator consists of 8 bytes, the system has a pointer for each byte in the PNT which tells it which two bytes of that charater it uses to color the quadrants. The nybbles in these two bytes are used as follows:

- The first byte's MSN describes the upper left quadrant's color
- The first byte's LSN describes the upper right quadrant's color
- The second byte's MSN describes the lower left quadrant's color.
- The second byte's LSN describes the lower right quadrant's color.

Figure G.1 shows the ranges of XPT and YPT and the VDP nybble assignments. As you can see from this drawing there is a type of indexing of the bytes in an 8-byte pattern generator block which corresponds to YPT. For example:

Index into Pattern Generator	YPT Values
0	0, 8, 16, 24, 32, 40
1	1, 9, 17, 25, 33, 41
2	2, 10, 18, 26, 34, 42
3	3, 11, 19, 27, 35, 43
4	4, 12, 20, 28, 36, 44
5	5, 13, 21, 29, 37, 45
6	6, 14, 22, 30, 38, 46
7	7, 15, 23, 31, 39, 47

When XPT is even, then MSN of each byte is used; and when XPT is odd, the LSN of each byte is used.

After the screen has been initialized with the format statement as described above, bit 1 of CPU RAM location >FD must be set. Once this bit is set, you cannot use format statements to change the screen. All changes to the screen must be done by setting XPT and YPT to specific values and storing the color you wish for that block in the character buffer (CB - CPU RAM >7D). For example, the instructions:

```
ST  37,@XPT
ST  13,@YPT
ST   4,@CB
```

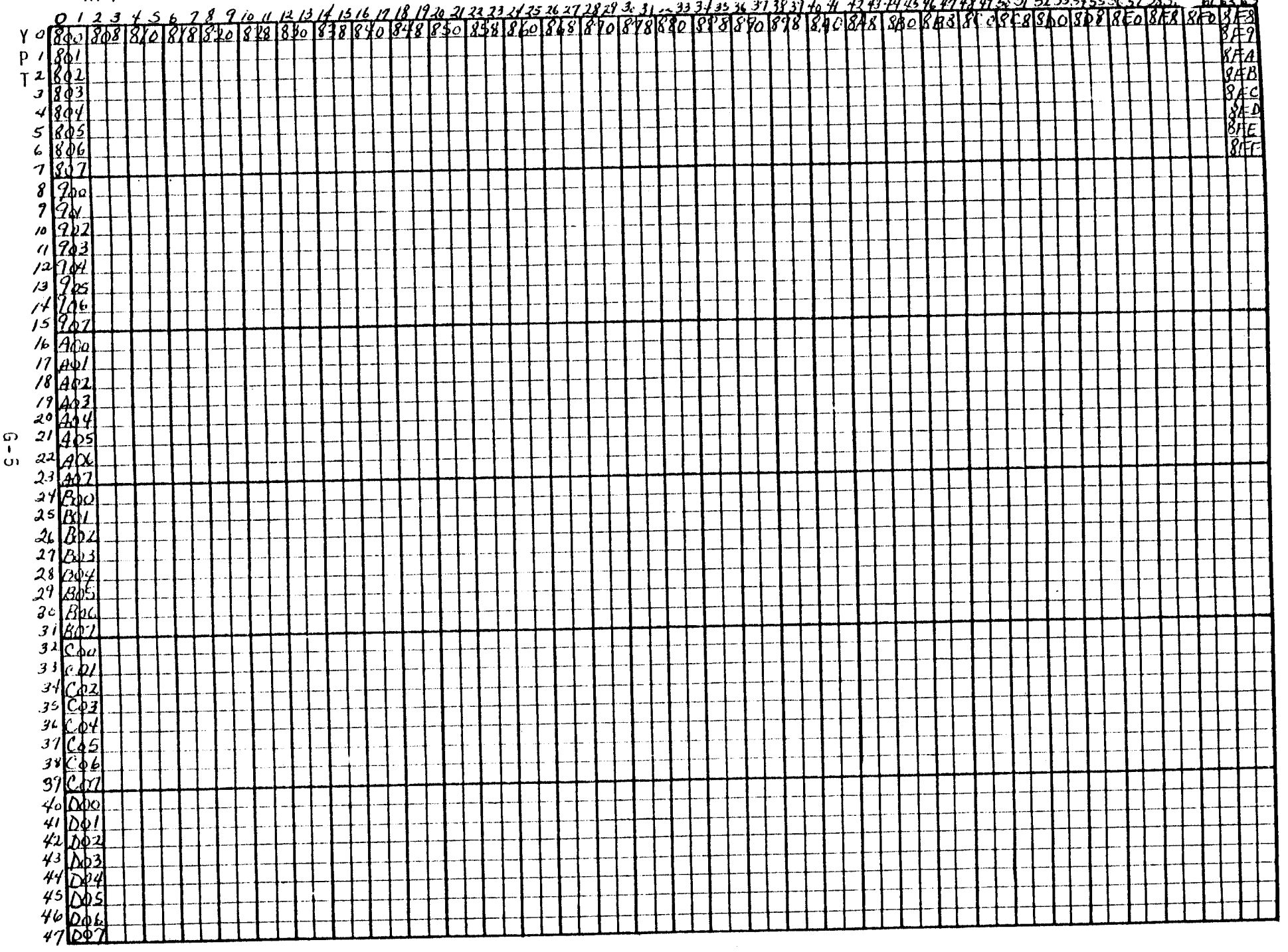
would put a 4 x 4 dot block of color blue 2 at the specified place on the screen and also put a 4 in the right nybble of VDP RAM (>995). A store in CB does not affect the PNT since the values from the initial format statement are the only ones which allow MCMD to work correctly.

The ALL instruction may be used in this mode to change screen. For example:

```
ALL >24
```

will look at VDP RAM (>920->927) and fill the screen with th colors in 2 x 8 blocks of 4 x 4 dots. It will also store >24 VDP RAM (0->2FF). Since the ALL instruction changes the val in the PNT, before successful use of MCMD can be made, programmer must reset bit 1 of CPU RAM location >FD re-initialize the screen with the format statement above. T set bit 1 at location >FD and proceed as above with a store CB of a color.

MULTI-CO' OR MODE



Each GROM or ROM that contains programs that may be accessed by programs outside of that ROM or GROM need a header. There are 6 types of programs currently defined. They are power up, user application, device service, subroutine links BASIC subprogram libraries, and interrupt service programs. Every type of program except user application programs, BASIC subprogram libraries, and interrupt service routines can be in either ROM or GROM. User application programs and BASIC subprogram libraries can only be in GROM. For every type of program in a GROM or ROM, there is a chained list of program headers. The first program header of each type is pointed to by an entry in the GROM/ROM header. GROM/ROM headers must be located at the beginning of a GROM or ROM. Program headers can be located anywhere. Within a multi-GROM package the GROM headers and program headers may be in the same or different GROMs. Table H.1 shows a GROM/ROM header and Table H.2 shows a program header.

1. SYSTEM INITIALIZATION

The monitor will start every application program with all of RAM in a defined state. CPU RAM will be zeroed except for >70 through >81. Location >70, >71 contains the highest address in VDP RAM. Location >72 will contain >9E and is the data stack pointer. Location >73 (the subroutine stack pointer) is initialized to >7E. Location >74 is zero. The other locations (>75 to >81) have undefined values.

TABLE H.1

GROM HEADER

LOCATION	SIZE	CONTENTS
X000	byte	>AA valid identification
X001	byte	version number
X002	byte	number of program
X003	byte	reserved
X004	word(2 bytes)	address of first power up routine header
X006	word	address of first user program header
X008	word	address of first DSR header
X00A	word	address of first subroutine link header
X00C	word	address of first interrupt link
X00E	word	address of first BASIC subprogram libraries

The address of any program types should be 0 in the GROM/ROM header if there are no programs of that type. The number of programs and verison number are not currently being used but should be used for future expansion.

TABLE H.2

PROGRAM HEADER

SIZE	CONTENTS
word	pointer of next program header of the same program type (0 if end of list)
word	entry address of program
byte	number of characters in program name (N)
N. bytes	ASCII character representation of program name

VDP RAM will have the 6 X 8 character set loaded. The VDP registers will be set for the standard locations (see Table 3.4, page 3-14). The screen will be blanked and the color table will contain all >17. All the rest of VDP RAM will be zeroes.

2. POWER-UP ROUTINES

The monitor initializes the system by calling power-up routines. It searches peripheral ROM and GROM headers for power-up routine addresses and executes them as it finds them. After each power-up routine is executed, a search is made for the next one. When there are no more power-up routines found, the console power-up routine is invoked. This routine puts up the initial screen and menu and calls the selected program. The selected program is started with the system initialized as described in Section 1.

Each ROM power-up can use R0 - R10. R12 will be set up with the proper CRU address to address the attached peripheral's CRU. The ROM power-up routine should end with a INCT R11 and a B *R11 to return to the system.

GROM power-up routines are called from GPL. They can be located in any slot of the library peripheral. They may not use subroutine links or call DSR's.

Power-up routines can use CPU RAM > 4 to > 71 for whatever it needs. They may also use all of VDP RAM. They must not change the data or subroutine stack pointers upon return to the monitor.

3. SUBROUTINE AND DSR CALLS

Subroutines and DSR's may be called through the monitor. The monitor is passed the name of the routine in VDP RAM. The name location in VDP RAM is pointed to by a 2-byte value in CPU RAM >56. The VDP locations contain a one-byte count of the number of characters in the name followed by the ASCII representation of the name with a "." (period) and some more characters. This may be repeated any number of times. The routine name the monitor uses consists of the string up to the first period, if any. The routine itself is called by

```
CALL    LINK    LINK EQU >10
DATA    BYTE
```

where byte is 8 for a DSR and > A for a subroutine link. The subroutine or DSR should return by

```
CALL    RETN    EQU > 12
```

If the routine is in ROM, R1 will contain a version number starting with 1. Every time a routine is found with the right name, R1 is incremented. This enables a routine to determine its position relative to other routines of the same name. If the version number is wrong, the routine should B *R11 without changing any registers. If the routine is executed , it should return by incrementing R11 by 2, moving a >12 to the top of the subroutine stack overwriting the address that is there, and branching indirect on R11. Registers R0 - R10 can be used. R11 has the return address for ROM code and R12 will be pointing to the peripheral CRU space.

For GROM programs, the subroutine or DSR may reside in another library peripheral slot. The subroutine or DSR calls may be nested. Each GROM subroutine or DSR call takes 4 bytes of subroutine stack. ROM subroutines and DSR's called through the monitor may not be nested.

4. GENERAL SUBROUTINES PROVIDED BY THE MONITOR

The monitor provides a group of subroutines that are of general use in many applications. These include mathematical functions, character sets, certain sounds, and application exit. The mathematical functions are described in Appendix K.

There are two routines to load VDP RAM with either a 6 x 8 or 5 x 6 character set. They are called by:

```
CHR1    EQU   >16
        CALL  CHR1        (6 x 8 characters)
CHR2    EQU   >18
        CALL  CHR2        (5 x 6 characters)
```

When they are called, CPU RAM location FAC should be pointing to the VDP RAM location of the first character (space).

There are two routines that give positive and negative acknowledge tones. These are used primarily for acknowledging good and invalid key pushes. The two routines are called by:

```
TON1    EQU   >34
        CALL  TON1        (positive acknowledge)
TON2    EQU   >36
        CALL  TON2        (negative acknowledge)
```

EXIT - RETURN TO MONITOR

An application program may exit and return to the monitor by:

```
EXIT
```

This instruction causes a software reset of the system. All power-up routines are executed and the initial screen displayed. This should not be confused with a hardware reset.

BIT REVERSAL ROUTINE >3B

Purpose: Provide a quick way to form mirror image bytes in VDP RAM

Input: FAC address of data in VDP (CPU RAM location >4A)

FAC+2 number of bytes to reverse

Call:

```
BITRVR  EQU  >3B
        CALL  BITRVR
```

Output: Every byte in VDP RAM from the first address pointed to by FAC to the byte pointed to by the address + numbers of bytes in FAC+2 is bit reversed. This means bits 0 and 7 are exchanged, bits 1 and 6 are exchanged, bits 2 and 5 are exchanged, and bits 3 and 4 are exchanged to give a mirror image of the byte.

Exceptions: None

Side Effects: CPU RAM from >00 to >40 will be destroyed.

APPENDIX I CASSETTE DSR

DEFINITION

A file consists of a collection of data groupings called logical records. This division of the file into logical records does not necessarily correspond to the physical division of data on the medium (like a block on a disk). Thus, there are two types of records:

- Logical records - the data grouping of a file as seen by the BASIC interpreter or other application programs.
- Physical records - the buffers physically transferred between memory and medium.

File I/O from a program is done on a logical record basis. The manipulation of physical records is done by the DSR.

For relative files, the logical records are fixed length. This enables the system to easily locate the physical position of a logical record, relative to the beginning of the file.

Sequential files allow variable length logical records.

When a file is created, the logical record size must be specified. For relative record files this size must be exact. For sequential files the specification is optional. If specified, the logical record size is used as an upper limit for any logical record size of that file.

The physical record size for any medium is specified within the DSR and is implementation dependent.

MODE OF OPERATION

A file is opened for a specific mode of operation, specified in the OPEN I/O call. The three modes of operation are:

- INPUT - the contents of the file may be read, but they may not be altered.
- OUTPUT - the file is being created. It's contents may be written but not read.
- APPEND - new data may be added at the end of the file, but the contents of the file may not be read.

This is the same on the cassette as output mode.

Each DSR decides whether or not a specific mode for an I/O operation can be accepted by the corresponding device.

IMPLEMENTATION

As mentioned, the DSR's should present a uniform interface between the File Management System and the peripherals. This section will give implementation details on this interface.

PERIPHERAL ACCESS BLOCK DEFINITION

All DSR's are accessed through a so called Peripheral Access Block (PAB). The definition for these PAB's is the same for every peripheral. The only difference between peripherals is that some peripherals will not support every option provided for in the PAB.

All PAB's are physically located in VDP RAM. They are created before the OPEN call, and are not to be released until the I/O has been closed for that device or file.

Figure I.1. (page I-6) shows the layout of a PAB. The PAB has a variable length, depending upon the length of the file descriptor.

The meaning of the bytes and bits within the PAB is:

BYTE	BIT	MEANING
0	-	I/O opcode - contains opcode for the current I/O-call.
1	-	Flagbyte/status - all the information the system needs about file-type, mode of operation, and data-type, is stored in this byte. The meaning of the bits within this flagbyte is (bit 7 is most significant bit, bit 0 is least significant bit).
	0	Filetype - indicated file-type 0 = Sequential file 1 = Relative record file Cassettes are always sequential.
	1-2	Mode of operation - indicates operation mode file has been opened for: 00 = UPDATE 01 = OUTPUT 10 = INPUT 11 = APPEND Cassette DSR does not support update or append.

BYTE	BIT	MEANING
	3	Datatype - indicates type of data stored in the file. DISPLAY type data comprises standard ASCII data. INTERNAL type data is implementation dependent. 0 = DISPLAY 1 = INTERNAL
	4	Recordtype - indicates type of record used. 0 = Fixed length records 1 = Variable length records
	5-7	Errorcode - these three bits indicate, in combination with the I/O opcode, the error type that has occurred (0 = no error).
2-3	-	Data buffer address - address of the data buffer the data has to be written to or read from. The buffer is always in VDP RAM.
4	-	Logical record length - indicates the logical record length for fixed length records, or the maximum length for a variable length record (see flagbyte). It is rounded up to the next highest multiple of 64.
5	-	Character count - number of characters to be transferred in write mode, or the number of bytes actually read in read mode. It is used by the cassette DSR only for reads and writes.

BYTE	BIT	MEANING
6-7	-	For cassettes, the record number is used for the number of bytes to load or save. This number must be larger than the number of bytes on the cassette record. This number is rounded up to the nearest multiple of 64 by the cassette DSR.
8	-	Screen offset - offset of the screen characters in respect to their normal ASCII value. This is used if your characters are not at the default positions in VDP RAM. It enables the cassette DSR to use your character set for messages. The cassette DSR messages look best using the small character set.
9	-	Name length - length of the file descriptor following the PAB.
10+	-	File descriptor - devicename. The length of this descriptor is given in byte 9. There are two valid names for cassettes: CS1 - cassette unit 1 CS2 - cassette unit 2

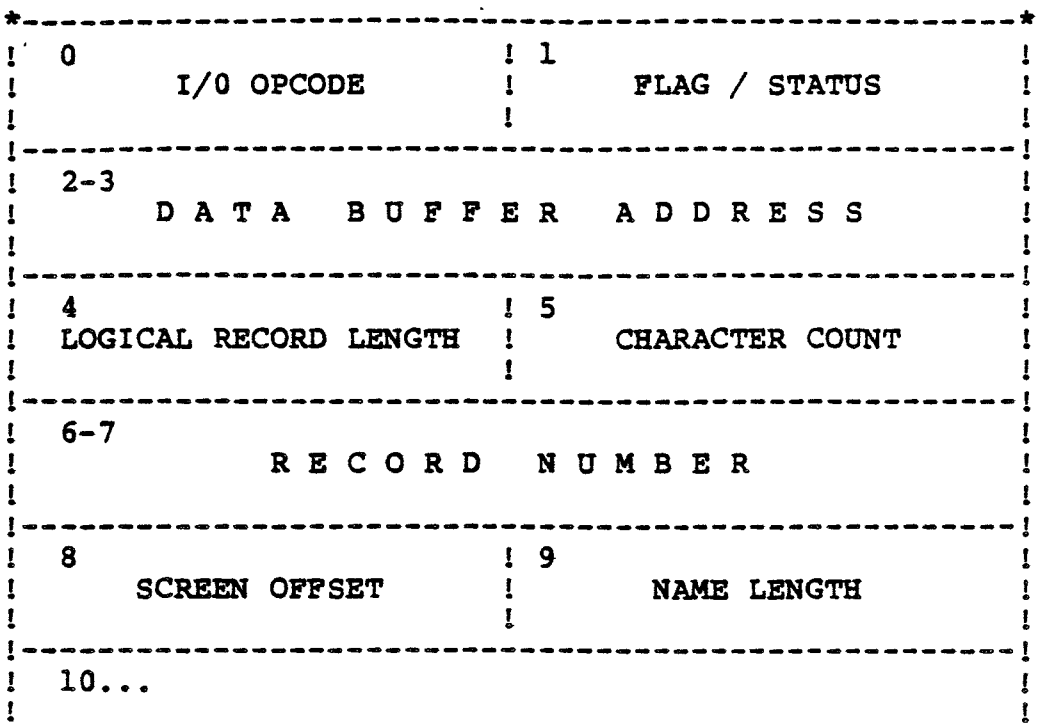


FIGURE I.1 PAB LAYOUT

This section describes the valid opcodes that can be used in the PAB. These valid opcodes are shown in Table I.2 (page I-6)

The following section will describe the general actions caused by an I/O-call with each of the I/O-opcodes. Each I/O-call returns any error-codes in the FLAG/STATUS byte of the PAB.

OPCODE	MEANING
00	OPEN
01	CLOSE
02	READ
03	WRITE
04	RESTORE/REWIND (not supported)
05	LOAD
06	SAVE
07	DELETE FILE - NO OPERATION FOR CASSETTE
08	SCRATCH RECORD - NOT USED BY CASSETTE
09	END OF FILE TEST (not supported)

TABLE I.2 I/O OPCODES

Open

The OPEN operation should be performed before any data transfer operation. The file remains open until a CLOSE operation is performed. The mode of operation for which the file has to be OPENed should be indicated in the flag byte of the PAB. In case this mode is OUTPUT, APPEND or INPUT, the record length (64) is returned in byte 4.

An OPEN operation must be performed before any other operation except LOAD or SAVE. Consistent use of OPEN and CLOSE is recommended for all files and devices; however, neither the OPEN nor the CLOSE operation is required for devices.

Close

The CLOSE operation informs the DSR that the current I/O sequence to that DSR has been completed.

After the CLOSE operation, the PAB is no longer needed, so it can be released. As long as no CLOSE operation is performed on an active PAB, this PAB has to be preserved.

Read

The READ operation reads a record from the selected device and stores the bytes in the specified buffer. The buffer address is specified in PAB entry 2 and 3, and the buffer size is specified in PAB entry 4. If the length of the input record exceeds the buffer size, the record is not read and an error is returned.

Write

The WRITE operation writes a record to the specified device from the buffer specified in the PAB. The number of bytes to be written is specified in byte 5 of the PAB.

Restore/Rewind

The RESTORE/REWIND operation repositions the file read pointer to the beginning of the file.

A RESTORE can only be used if the file is opened for INPUT mode. RESTORE itself does not perform any READ operation.

Load

The LOAD operation loads an entire program from an external device or file into program memory. All the control information for BASIC is contained in the load file. Since all information is directly written to program memory without intermediate buffering, no buffer memory needs to be assigned.

The LOAD operation is a stand alone operation, i.e., the LOAD operation can be used without previous OPEN operation.

For the LOAD operation, the PAB needs to contain the following information:

Bytes 2 and 3 should contain the start address of the program memory.

Bytes 6 and 7 should contain the maximum number of bytes available for the program.

Aside from the I/O opcode and the file descriptor, no more information is required for the LOAD operation.

Save

SAVE is the complementary operation for LOAD. Instead of loading a program from a device or file, it writes a program from program memory to a device or file. Again, only a small part of the PAB is used. Aside from the usual information (I/O opcode and file descriptor), the PAB should contain the start address of the program to be SAVEd in bytes 2 and 3, and the number of bytes to be SAVEd in bytes 6 and 7.

BASIC automatically saves all the control information necessary for reloading of the program, together with the program code.

Delete

The DELETE operation deletes the specified file from the specified device. This operation also CLOSEs the I/O sequence. The DELETE operation can only be used in UPDATE, APPEND or OUTPUT mode.

Scratch Record

The SCRATCH RECORD operation scratches the specified record from the specified (relative record) file. The record to be scratched is specified in byte 6 and 7 of the PAB. This operation will cause an error for sequential files and devices.

VERIFY

The VERIFY command allows the record on tape to be compared against what is in VDP RAM. It will return an error code if the

record is unreadable or if there is a difference between the tape's data and the VDP data.

ERROR CODES

The File Management System shall support the following error codes:

1. BAD DEVICE NAME

   the device indicated is not in the system.

2. ILLEGAL OPERATION

   either an invalid operation was specified,

   or a conflict with the OPEN mode has occurred.

3. DEVICE ERROR

   covers all hard device errors, such as parity and bad medium errors.

ISSUING THE COMMAND TO THE CASSETTE DSR

After the PAB is set up, the cassette DSR is called by putting the address of the name length (byte 9 of the PAB) in CPU RAM location >56 and then calling a subroutine at location >10 in GROM 0. This is illustrated as follows for a save routine:

```
DSR     EQU >10     Address of subroutine
NAMLEN  EQU >56     Address of byte 9 of PAB
.
.
.
```

```
        MOVE  13 FROM ROM(#PABCAS) TO RAM ( >500)
        DST   # >509, @NAMLEN    Address of byte 9 of PAB in VDP
        CALL    DSR
        DATA    8               *Tells subroutine this is a DSR
        .
        .
        .
PABCAS  DATA  >06               Opcode for save
        DATA  >02               Sets output status for save
        DATA #>600              Address in VDP of data buffer
        DATA  >40               Fixed record length size for cassette
        DATA  >00               Character count for cassette
        DATA #>6F0              Number of bytes to be read
        DATA  >00               Bias for ASCII characters
        DATA  >03               Length of name of device
        DATA  :CS1:             Name of device*
```

* For cassettes the name of the device is predefined as CS1 or CS2 and these are the only names you are allowed to use.

AUDIO GATE

CRU bit 24 is the audio gate bit which allows data being read to be heard. If the bit is set to 0, the data being read is heard, and if the bit is set to 1, the data is not heard. Setting this bit to a 0 or 1 is done with an I/O instruction.

MOTOR CONTROL

There are two CRU bits (22 and 23) used to control cassettes 1 and 2, respectively. When there is no Cassette I/O being done,

both motors remain on. When Cassette I/O is specified, the DSR will control the data being read. If there are two motor units connected, the data will be read simultaneously, or you may have the option of reading data from one motor unit and playing the recorded voice from another motor unit through the TV speaker.

APPENDIX J - LIST OF INSTRUCTIONS

PART 1 ALPHABETIC

MNEMONIC	OPCODE(>)	FORMAT	STATUS AFFECTED	INSTRUCTION	SECTION
A	A0	1	ALL	ADD	4.4.1
ABS	80	6	NONE	ABSOLUTE VALUE	4.49
ADD	A0	1	ALL	ADD	4.4.1
ALL	07	2	NONE	LOAD SCREEN	4.5.3
AND	B0	1	ALL	LOGICAL AND	4.4.12
B	05	3	COND	LONG BRANCH	4.2.3
BACK	04	2	NONE	LOAD BORDER COLOR	4.5.2
BR	40	4	COND	BRANCH ON RESET	4.2.2
BS	60	4	COND	BRANCH ON SET	4.2.1
CALL	06	3	COND	CALL SUBROUTINE	4.2.5
CARRY	0C	5	COND	CARRY STATUS TO COND	4.1.3
CASE	8A	6	COND	CASE BRANCH	4.2.4
CEQ	D4	1	COND	COMPARE EQUAL	4.1.5
CGE	D0	1	COND	COMPARE GREATER OR EQUAL	4.1.9
CGT	CC	1	COND	COMPARE GREATER	4.1.8
CH	C4	1	COND	COMPARE HIGH	4.1.6
CHE	C8	1	COND	COMPARE HIGH OR EQUAL	4.1.7
CLOG	D8	1	COND	COMPARE LOGICAL	4.1.10
CLR	86	6	NONE	CLEAR	4.4.15
COINC	ED	1	COND	COINCIDENCE	4.5.1
CONT	10	5	NONE	BASIC CONTINUE	
CZ	8E	6	COND	COMPARE TO ZERO	4.1.11
D	AC	1	ALL	DIVIDE	4.4.4
DEC	92	6	ALL	DECREMENT BY ONE	4.4.7
DECT	96	6	ALL	DECREMENT BY TWO	4.4.8

MNEMONIC	OPCODE(>)	FORMAT	STATUS AFFECTED	INSTRUCTION	SECTION
DIV	AC	1	ALL	DIVIDE	4.4.4
EX	C0	1	NONE	EXCHANGE	4.4.17
EXEC	11	5	ALL	BASIC EXECUTE	
EXIT	0B	5	NONE	EXIT PROGRAM	4.5.8
FETCH	88	6	NONE	FETCH FROM CALL	4.2.6
FMT	08	7	----	FORMAT SCREEN	4.5.4
GT	0A	5	COND	GREATER STATUS TO COND	4.1.2
H	09	5	COND	HIGH STATUS TO COND	4.1.1
INC	90	6	ALL	INCREMENT BY ONE	4.4.5
INCT	94	6	ALL	INCREMENT BY TWO	4.4.6
I/O	F6	8	NONE	SPECIAL I/O	4.5.9
INV	84	6	NONE	INVERT (ONE'S COMPLEMENT)	4.4.11
MOVE	20	9	NONE	MOVE DATA	4.4.20
M	A8	1	NONE	MULTIPLY	4.4.3
MUL	A8	1	NONE	MULTIPLY	4.4.3
NEG	82	6	NONE	NEGATE(TWO'S COMPLEMENT)	4.4.10
OR	B4	1	ALL	LOGICAL OR	4.4.13
OVF	0D	5	COND	OVERFLOW STATUS TO COND	4.1.4
PARSE	0E	2	ALL	BASIC PARSE	
PUSH	8C	6	NONE	PUSH DATA STACK	4.4.18
RAND	02	2	NONE	RANDOM NUMBER	4.5.5
RB	B0	1	ALL	RESET BIT	4.3
RTN	00	5	COND	RETURN FROM SUBROUTINE	4.2.7
RTNB	12	5	ALL	BASIC RETURN	
RTNC	01	5	NONE	RETURN FROM SUBROUTINE	4.2.8
S	A4	1	ALL	SUBTRACT	4.4.2

MNEMONIC	OPCODE(>)	FORMAT	STATUS AFFECTED	INSTRUCTION	SECTION
SB	B4	1	ALL	SET BIT	4.3
SCAN	03	5	COND	SCAN KEYBOARD	4.5.6
SLL	E0	1	NONE	SHIFT LEFT LOGICAL	4.4.21
SRA	DC	1	NONE	SHIFT RIGHT ARITHMETIC	4.4.22
SRC	E8	1	NONE	SHIFT RIGHT CIRCULAR	4.4.24
SRL	E4	1	NONE	SHIFT RIGHT LOGICAL	4.4.23
ST	BC	1	NONE	STORE	4.4.16
SUB	A4	1	ALL	SUBTRACT	4.4.2
TBR	D8	1	COND	TEST BIT RESET	4.3
XML	0F	2	NONE	EXECUTE MACHINE LANGUAGE	4.5.7
XOR	B8	1	ALL	LOGICAL EXCLUSIVE OR	4.4.14

The following instructions are used to access BASIC Language:

CONT	BASIC Continue
PARSE	BASIC Parse
RTNB	BASIC Return
EXEC	BASIC Execute

PART 2 INSTRUCTION MAP

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	RTN	RTNC	RAND	SCAN	BACK	B	CALL	ALL	FMT	H	GT	EXIT	CARRY	OVF	PARSE	XML
1	CONT	EXEC	RTNB	UNUSED												
2	MOVE															
3																
4	BR															
5																
6	BS															
7																
8	ABS		NEG		INV		CLR		FETCH		CASE		PUSH		CZ	
9	INC		DEC		INCT		DECT		UNUSED				UNUSED			
A	ADD				SUB				MUL				DIV			
B	AND				OR				XOR				ST			
C	EX				CH				CHE				CGT			
D	CGE				CEQ				CLOG				SRA			
E	SLL				SRL				SRC				UNUSED	COINC	UNUSED	
F	UNUSED						I/O		UNUSED							

APPENDIX K FLOATING POINT OPERATIONS

There are several subroutines in the monitor which can be called from a GPL program. These subroutines are described in this appendix. It is important the programmer realize that when one of these subroutines is called the contents of CPU RAM locations >4A through >6F may be used, and VDP RAM locations >3CO through >3DF will be used for roll out.

The mathematical function subroutines provided in the monitor include convert number to string, greatest integer, involution, square root, exponential, natural log, cosine, sine, tangent, and arctangent. They are called as follows:

FAC is CPU RAM >4A (8 bytes)

ARG is CPU RAM >5C (8 bytes)

STATUS is CPU RAM >7C

SGN is CPU RAM >75

EXP is CPU RAM >76

VSPTR is CPU RAM >6E (2 bytes)

FPERAD is CPU RAM >6C

Error Codes:

WRNOV	>01	- warning, overflow
DIVZER	>01	- division by zero
ERRSNN	>02	- syntax error
ERRIOV	>03	- integer overflow on conversion
ERRSQR	>04	- square root of negative number
ERRNIP	>05	- negative number to non-integral power
ERRLOG	>06	- log of negative number or zero
TRIGER	>07	- invalid argument in trig function

CNS - CONVERT NUMBER TO STRING

Purpose: Convert a floating point number to an ASCII string.

Input:

FAC The floating point value.

FAC+11 If set to 0, the output string will be in BASIC format. If greater than 0, represents output in CALCULATOR mode. The contents are the effective calculator width, exclusive of decimal point. The following two cells are also required in CALCULATOR mode.

FAC+12 If zero, express overflow from calculator range by + or - EE...E. Underflow is expressed as 0. If positive, under- or over-flow from calculator range is expressed in E-format using the number of significant digits specified by this cell.

FAC+13 The number of digits to fix to the right of the decimal point. A negative value disables the FIX mode.

```
CNS     EQU   >14
```

CALL:

```
        CALL  CNS
```

Output: FAC The FAC contents will be modified due to rounding performed for display purposes.

FAC+11 Points to the beginning of the result string. The string will be entirely contained within the floating point scratch area between FAC and FPERAD.

FAC+12 The length of the string, in bytes.

Exceptions: None

INT - GREATEST INTEGER FUNCTION

Purpose: Compute the greatest integer contained in a floating point value.

Input: FAC The floating point value.

```
INT   EQU   >22
```

Call: CALL INT

Output: FAC The greatest integer contained in the floating point value. For positive numbers the integer is the truncated value. For negative numbers the integer is the truncated value plus one.

STATUS The status byte is set according to the contents of FAC after the operation.

Exceptions: None

PWR - INVOLUTION ROUTINE

Purpose: Raise a number, B, to a specified power, E

Input: FAC The exponent, E.

STACK The base, B.

PWR EQU >24

Call: CALL PWR

Output: FAC The result, B**E. The result is computed as EXP (E * LOG(ABS(B))). If B is negative and E is an odd integer, the result is negated.

STATUS The status byte is set according to the contents of FAC.

Exceptions: Negative number to non-integer power. Zero raised to a negative power. Overflow if result greater than maximum value.

Side Effects: SGN and EXP are destroyed. The previous FAC contents are destroyed and the contents of VSPTR are decremented by 8.

SQR - SQUARE ROOT ROUTINE

Purpose: Compute the square root of a number.

Input: FAC The input value.
SQR EQU >26

Call: CALL SQR

Output: FAC The square root of the number.
STATUS Set according to the contents of FAC.

Exceptions: If the input value is negative, the ERRSQR condition results.

Side Effects: SGN and EXP are destroyed. The contents of VSPTR are unchanged.

EXP - EXPONENTIAL ROUTINE

Purpose: Compute the inverse natural logarithm.

Input: FAC The input value.
EXP EQU >28

Call: CALL EXP

Output: FAC The inverse natural logarithm.
STATUS Set according to the contents of FAC.

Exceptions: Overflow of the result causes the WRNOV condition.

Side Effects: SGN and EXP are destroyed. The contents of VSPTR are unchanged.

LOG - NATURAL LOGARITHM ROUTINE

Purpose: Compute the natural log of a number.

Input: FAC The input value.
LOG EQU >2A

Call: CALL LOG

Output: FAC The natural log of the number.
STATUS Set according to the contents of FAC.

Exceptions: If the input value is zero or negative, the ERRLOG condition results.

Side Effects: SGN and EXP are destroyed. The contents of VSPTR are unchanged.

COS - COSINE ROUTINE

Purpose: Compute the cosine of a number (in radians).

Input: FAC The input value.

COS EQU >2C

Call: CALL COS

Output: FAC The cosine of the number.

STATUS Set according to the contents of FAC

Exceptions: None

Side Effects: SGN and EXP are destroyed. The contents of VSPTR are unchanged.

SIN - SINE ROUTINE

Purpose: Compute the sine of a number (in radians)

Input: FAC The input value.

SIN EQU >2E

Call: CALL SIN

Output: FAC The sine of the number.

STATUS Set according to the contents of FAC.

Exceptions: None

Side Effects: SGN and EXP are destroyed. The contents of VSPTR are unchanged.

TAN - TANGENT ROUTINE

Purpose: Compute the tangent of a number (in radians).

Input: FAC The input value.

TAN EQU >30

Call: CALL TAN

Output: FAC The tangent of the number (in radians).

STATUS Set according to the contents of FAC.

Exceptions: If the input value causes an overflow the WRNOV condition results.

Side Effects: SGN and EXP are destroyed. The contents of VSPTR are unchanged.

ATN - ARCTANGENT ROUTINE

Purpose: Compute the arctangent of a number (in radians)

Input: FAC The input value.

ATN EQU >32

Call: CALL ATN

Output: FAC The arctangent of the number.

STATUS Set according to the contents of FAC.

Exceptions: None

Side Effects: SGN and EXP are destroyed. The contents of VSPTR are unchanged.

The floating point routines provided in ROM are convert string to number, convert floating to integer, floating add, floating subtract, floating multiply, floating divide, floating compare, stack add, stack subtract, stack multiply, stack divide, and stack compare. All numbers are 8-bits.

As a number is used on the value stack, the stack pointer is incremented by 8. All errors are returned in location FAC + 10. Only overflow errors are detected and the code is 1 for a floating point overflow and 3 for integer overflow.

CSN - CONVERT STRING TO NUMBER

Purpose: Convert an ASCII string to a floating point number.

Input: FAC Address of the string.

```
CSN     EQU   >10
```

Call:

```
XML     CSN
```

(The instruction FLTPT will generate the same code as XML)

Output: FAC Number returned here. All numbers are returned in internal format which is

radix 100. CPU RAM space FAC thru FAC+9 should be reserved for the answer.

FAC+10 Error code (>01 - overflow)

CFI - CONVERT FLOATING POINT TO INTEGER

Purpose: A rounded conversion of a floating point number to an integer.

Input: FAC Floating point number

CFI EQU >12

Call: XML CFI

Output: FAC Integer value returned in first two bytes.

FAC+10 Error code (>03 - overflow)

Exceptions: Range of integer must be -32,768 to 32,767

FADD - FLOATING POINT ADDITION

Purpose: Perform addition in base 100.

Input: ARG Left-hand term

FAC Right-hand term

FADD EQU >06

Call: XML FADD

Output: FAC Result of addition problem.

FAC+10 Error code (>01 - overflow)

FSUB - FLOATING POINT SUBTRACTION

Purpose: Perform subtraction in base 100.

Input: ARG Left-hand term

FAC Right-hand term

FSUB EQU >07

Call: XML FSUB

Output: FAC Result of subtraction problem.

FAC+10 Error code (>01 - overflow)

FMUL - FLOATING POINT MULTIPLICATION

Purpose: Perform multiplication in base 100.

Input: ARG Multiplicand

FAC Multiplier

FMUL EQU >08

Call: XML FMUL

Output: FAC Result

FAC+10 Error code (>01 - Overflow)

FDIV - FLOATING POINT DIVISION

Purpose: Perform division in base 100.

Input:
```
ARG     Dividend
FAC     Divisor
FDIV    EQU    >09
```

Call:
```
XML     FDIV
```

Output:
```
FAC     Result
FAC+10    Error code (  01 - Overflow)
```

FCOMP - FLOATING POINT COMPARE

Purpose: Compare two base 100 numbers.

Input:
```
ARG     First argument to compare
FAC     Second argument to compare
FCOMP   EQU    >0A
```

Call:
```
XML     FCOMP
```

Output:
```
STATUS    Bits set according to the compare --
          High bit is set if ARG is logically
          higher than FAC, greater than bit is
          set if ARG is arithmetically
          greater than FAC, condition bit is
          set if ARG and FAC are equal.
```

SADD - VALUE STACK ADDITION

Purpose: Perform base 100 addition of the top value on the value stack in VDP RAM with another value.

Input:

```
ARG     Top number on the value stack (VDP RAM
        address pointed to by VSPTR) is left-
        hand term.
FAC     Right-hand term
SADD    EQU   >0B
```

Call:

```
XML     SADD
```

Output:

```
FAC  Result
FAC+10    Error code (>01 - Overflow)
```

SSUB - VALUE STACK SUBTRACTION

Purpose: Perform base 100 subtraction of a number from the top of the value stack.

Input:

```
ARG     TOP number on the value stack is
         left-hand term
FAC     Right-hand term
SSUB    EQU   >0C
```

Call:

```
XML     SSUB
```

Output:

```
FAC     Result
FAC+10    Error code (>01 - overflow)
```

SMUL - VALUE STACK MULTIPLICATION

Purpose: Perform base 100 multiplication of a number from the top of the value stack with another number.

Input:
```
ARG     TOP number on the value stack is
        multiplicand.
FAC     Multiplier
SMUL    EQU   >0D
```

Call:
```
XML     SMUL
```

Output:
```
FAC     Result
FAC+10      Error code ( >01 - Overflow)
```

SDIV - VALUE STACK DIVISION

Purpose: Perform base 100 division of a number from the top of the value stack by another number.

Input:
```
ARG     Top number on the value stack-dividend
FAC     Divisor
SDIV    EQU   >0E
```

Call:
```
XML     SDIV
```

Output:
```
FAC     Result
FAC+10      Error code ( >01 - overflow)
```

SCOMP - VALUE STACK COMPARE

Purpose: Compare the top number on the value stack to another number

Input:
- ARG — TOP number on the value stack - first argument
- FAC — Second argument
- SCOMP EQU >0F

Call: XML SCOMP

Output: STATUS — ARG is compared to FAC and the high, greater than, and condition bits are set accordingly.

RADIX 100

The internal format for all floating point numbers is radix 100. Each number consists of 8 bytes., The first byte is the exponent byte with a bias of >40. The exponent for the number is added to >40 to get the exponent needed. The following bytes are the base-100 digits of the number. The first word of the number is negated if the number is negative. An even number of digits must follow the decimal (radix) point. Only the most significant digits of the number should be used and at least one digit and not more than 2 digits should be to the left of the radix point. For example, the number 12,500 has 3 most significant digits, 1, 2, and 5. Using only these digits and having an even number of digits following the radix point, we must write 1.25×100^2. In radix 100 it would look like:

>42	>01	.	>19	>00	>00	>00	>00	>00
byte1	byte2	radix point	byte3	byte4	byte5	byte6	byte7	byte8

The VDP chip is accessed by writing to the appropriated memory mapped location (see Home Computer System Memory, CRU, and Interrupt Mapping Specification). First, the VDP address pointer is loaded by writing out, sequentially, two bytes (low byte first) to the VDP address location. (If the full operation is to be a WRITE data to VDP, then the 2 byte address must be ORed with >4000).

Because of timing considerations on the VDP, there should be a delay of at least 6 μsec between a read or write operation and loading the address pointer (or between any two VDP operations).

Data may then be moved from (to) the VDP read-(write)-data address which will contain the content of VDP memory pointed to by the VDP address register. After each operation the VDP address pointer automatically increments and points to the next location. Therefore, the address pointer does not have to be reloaded to move blocks of VDP memory.

```
R1 = @MSB(LSB) two byte VDP address
R2 = @VDPWA I/O write address
R3 = @VDPWD address to write data
R4 = @VDPRD address to read data
{ORI  R1,>4000}     write option
MOVB @R1LSB,*R2
MOVB R1,*R2
SLA  R8,6           delay
MOVB *R4,@LOC       read data R4 = @VDPRD
{MOVB @LOC,*R3}     write R3 = @VDPWD
```

GROM is accessed by writing a two-byte address (high order byte first) to the appropriate memory-mapped GROM write address location. Data may then be moved from (to) the GROM read (write) data address which will contain the contents of GROM memory pointed to by the GROM address location. After each operation, the GROM address pointer automatically increments and points to the next location. Therefore, the address pointer does not have to be reloaded to move blocks of GROM data.

```
R1 = @MSB(@LSB)   two-byte GROM address
R2 = @GRMWA - GROM write address
R3 = @grmwd - address to write GROM data
R4 = @GRMRD - address containing current GROM data
     MOVB R1,*R2
     MOVB @R1LSB,*R2
     SLA  R8,16          delay
     MOVB *R4,R6         move data from GROM address to R6
     MOVB  R6,*R3        move data from R6 to GROM address
```

To create sound in an Assembly Language program, you create a sound list exactly as you would in Graphics Language. The address of this sound list should be stored in locaion >83CC which is CPU RAM location >CC. If this address is in VDP RAM, the low order of R14 should be a 1; if the address is in GROM, the low order bit should be a 0. Location >83CE (CPU RAM >CE - number of sound bytes) should contain a 1. To allow for interrupt detection, you should do two LIMI instructions about every 400 instructions.

```
R14LB    EQU    >83FD
ONE      BYTE   >01
```

```
SOUND   DATA   >700       *Sound list in VDP RAM
        MOV    @SOUND,@>83CC
        MOVB   @ONE,@>83CE
        SOCB   @ONE,@R14LB
          .
          .  (400 - 500 Instructions)
          .
        LIMI   2   Sees interrupts greater or equal to 2
        LIMI   0   No interrupts except reset or load
```

assist the user at a particular point in the program. The form of this help is completely application-dependent; most of the time aid will not be available. A message that AID is active should be displayed when such is the case.

D. PAGE FORWARD (FWD)

ENTER is to be used to go forward to the next display. The situation arises when a message is on the screen and the program is waiting to be told that the user is ready to go on; this may also be used to add more information to the current display under user control. A message to "PRESS ENTER" can be dynamically added to the display at the appropriate time. An extra ENTER to go forward is required after the ENTER used to record a data entry field at the end of the page.

E. PAGE BACKWARD (BACK)

SHIFT-Z returns the user to the nearest previous logical point in the program. This will usually be the display seen immediately prior to the current one. When properly implemented, the user should be able to back out to the console display with successive presses of SHIFT-Z.

F. REDO (REDO)

SHIFT-R is appropriate to replot or restart the current display. This is particularly useful to allow the user to restart at the top of a multi-prompt display.

assist the user at a particular point in the program. The form of this help is completely application-dependent; most of the time aid will not be available. A message that AID is active should be displayed when such is the case.

D. PAGE FORWARD (FWD)

ENTER is to be used to go forward to the next display. The situation arises when a message is on the screen and the program is waiting to be told that the user is ready to go on; this may also be used to add more information to the current display under user control. A message to "PRESS ENTER" can be dynamically added to the display at the appropriate time. An extra ENTER to go forward is required after the ENTER used to record a data entry field at the end of the page.

E. PAGE BACKWARD (BACK)

SHIFT-Z returns the user to the nearest previous logical point in the program. This will usually be the display seen immediately prior to the current one. When properly implemented, the user should be able to back out to the console display with successive presses of SHIFT-Z.

F. REDO (REDO)

SHIFT-R is appropriate to replot or restart the current display. This is particularly useful to allow the user to restart at the top of a multi-prompt display.

Keys should be referred to by function name in the GROM rather than by how they are implemented, i.e., "PRESS AID" rather than "PRESS SHIFT-A." The user's instruction manual will list the correspondence between key and name. This will allow the GROM to be used in a later version of the machine that may have those functions implemented in different keys. See Figure M.1 for a list of key code assignments.

G. INSERT (INS)

Data will be inserted in front of the character which the cursor is over when SHIFT-G is pressed.

H. DELETE (DEL)

SHIFT-F is pressed to delete the character under the cursor.

I. ERASE (ERASE)

SHIFT-T is pressed to delete a line which is being typed in BASIC.

J. COMMAND (CMD)

SHIFT-V is a special key that is program definable.

K. CLEAR (CLEAR)

SHIFT-C is used to clear a field in an application program; also acts as the "BREAK" key in BASIC.

DIRECTION ARROWS

Up and down non-destructive movement of the cursor is accomplished by means of the SHIFT-X and SHIFT-E keys. Use of either of these keys substitutes for an ENTER at the end of a line.

Basically, the scroll or cursor control keys, when implemented, specify the direction to move the user's line of sight:

a. SHIFT-E is the up arrow key, and will cause the screen to scroll down, revealing previous lines.

b. SHIFT-X is the down arrow key, and will cause the screen to scroll up, revealing new lines.

c. SHIFT-S is the left arrow key, and will cause the screen to move to the right, revealing the area on the left.

d. SHIFT-D is the right arrow key, and will cause the screen to move to the left, revealing the area on the right.

When circumstances make such usage unambiguous, the unshifted arrow keys, as well as contiguous keys, may be used to give a joystick control on the console keyboard. The mapping for this is:

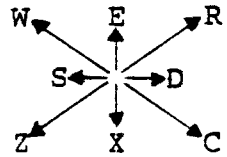
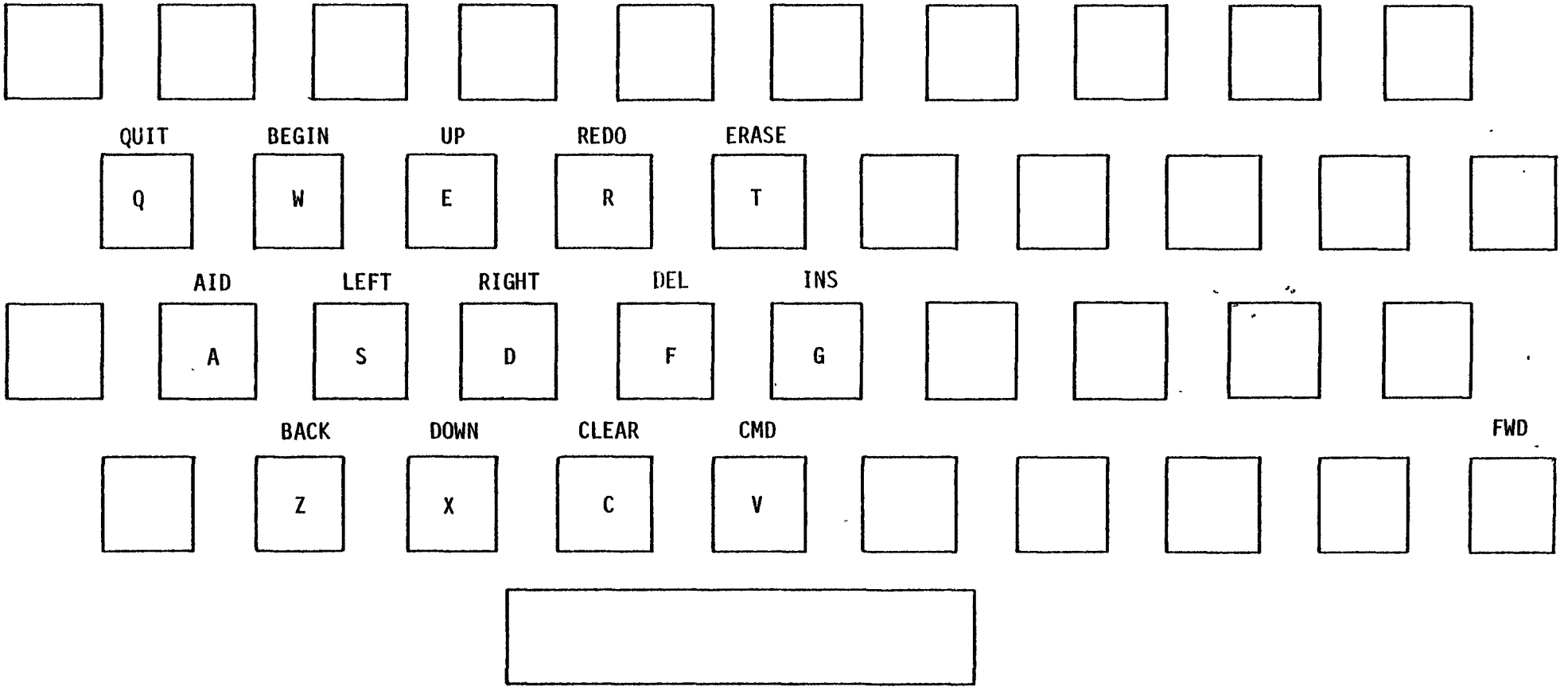


FIGURE M.1

CONSOLE FUNCTION OVERLAY



The charts on pages M-7 through M-9 may be used as an aid to the programmer to keep track of CPU and VDP RAM. The chart on page M-7 is a representation of CPU RAM locations >00 through >BF which are locations available to the programmer. Page M-8 is a chart with locations for the Sprite Attribute Block, Sprite Descriptior Block, and Sprite Velocity Block. Page M-9 shows the first 32 sets of patterns in the Pattern Generator area. The beginning VDP RAM location for each set is given with the pattern numbers for each pattern.

CPU-RAM CHART

FIGURE 1.2

00	20	40	60	80 (sub stack)	A0 (data stack)
01	21	41	61	81	A1
02	22	42	62	82	A2
03	23	43	63	83	A3
04	24	44	64	84	A4
05	25	45	65	85	A5
06	26	46	66	86	A6
07	27	47	67	87	A7
08	28	48	68	88	A8
09	29	49	69	89	A9
0A	2A	4A	6A	8A	AA
0B	2B	4B	6B	8B	AB
0C	2C	4C	6C	8C	AC
0D	2D	4D	6D	8D	AD
0E	2E	4E	6E	8E	AE
0F	2F	4F	6F	8F	AF
10	30	50	70 } RAM SIZE	90	B0
11	31	51	71 }	91	B1
12	32	52	72 DATSTK (>A0)	92	B2
13	33	53	73 SUBSTK (>80)	93	B3
14	34	54	74 KEYBRD	94	B4
15	35	55	75 KEY	95	B5
16	36	56	76 JOY Y	96	B6
17	37	57	77 JOY X	97	B7
18	38	58	78 RANDOM	98	B8
19	39	59	79 TIMER	99	B9
1A	3A	5A	7A MOTION	9A	BA
1B	3B	5B	7B VDPSTT	9B	BB
1C	3C	5C	7C STATUS	9C	BC
1D	3D	5D	7D CB	9D	BD
1E	3E	5E	(7E) YPT	9E	BE
1F	3F	5F	7F XPT	(9F)	BF

FIGURE M.3

SPRITE TABLE _____ VDP REG (1) = _____

SPRITE =		YPT	XPT	CHAR	COL.	VELO-CITY	Y	X
0	>300					>780		
1	>304					>784		
2	>308					>788		
3	>30C					>78C		
4	>310					>790		
5	>314					>794		
6	>318					>798		
7	>31C					>79C		
8	>320					>7A0		
9	>324					>7A4		
10	>328					>7A8		
11	>32C					>7AC		
12	>330					>7B0		
13	>334					>7B4		
14	>338					>7B8		
15	>33C					>7BC		

SPRITE =		YPT	XPT	CHAR.	COL.	VELO-CITY	Y	X
16	>340					>7C0		
17	>344					>7C4		
18	>348					>7C8		
19	>34C					>7CC		
20	>350					>7D0		
21	>354					>7D4		
22	>358					>7D8		
23	>35C					>7DC		
24	>360					>7E0		
25	>364					>7E4		
26	>368					>7E8		
27	>36C					>7EC		
28	>370					>7F0		
29	>374					>7F4		
30	>378					>7F8		
31	>37C					>7FC		

CHAR.	RAM	DATA	CHAR	RAM	DATA	CHAR.	RAM	DATA	CHAR.	RAM	DATA
>80	>400		>98	>4C0		>B0	>580		>C8	>640	
>81	>408		>99	>4C8		>B1	>588		>C9	>648	
>82	>410		>9A	>4D0		>B2	>590		>CA	>650	
>83	>418		>9B	>4D8		>B3	>598		>CB	>658	
>84	>420		>9C	>4E0		>B4	>5A0		>CC	>660	
>85	>428		>9D	>4E8		>B5	>5A8		>CD	>668	
>86	>430		>9E	>4F0		>B6	>5B0		>CE	>670	
>87	>438		>9F	>4F8		>B7	>5B8		>CF	>678	
>88	>440		>A0	>500		>B8	>5C0		>D0	>680	
>89	>448		>A1	>508		>B9	>5C8		>D1	>688	
>8A	>450		>A2	>510		>BA	>5D0		>D2	>690	
>8B	>458		>A3	>518		>BB	>5D8		>D3	>698	
>8C	>460		>A4	>520		>BC	>5E0		>D4	>6A0	
>8D	>468		>A5	>528		>BD	>5E8		>D5	>6A8	
>8E	>470		>A6	>530		>BE	>5F0		>D6	>6B0	
>8F	>478		>A7	>538		>BF	>5F8		>D7	>6B8	
>90	>480		>A8	>540		>C0	>600		>D8	>6C0	
>91	>488		>A9	>548		>C1	>608		>D9	>6C8	
>92	>490		>AA	>550		>C2	>610		>DA	>6D0	
>93	>498		>AB	>558		>C3	>618		>DB	>6D8	
>94	>4A0		>AC	>560		>C4	>620		>DC	>6E0	
>95	>4A8		>AD	>568		>C5	>628		>DD	>6E8	
>96	>4B0		>AE	>570		>C6	>630		>DE	>6F0	
>97	>4B8		>AF	>578		>C7	>638		>DF	>6F8	

GLOSSARY

Bit Reset - The bit has value 0

Bit Reversal - Create a mirror image of a byte

Bit Set - the bit has value 1

CPU - Central Processor Unit

DSR - Device Service Routine

GROM - Graphics Read Only Memory; P-Channel ROM

LSB - Least significant bit

LSBy - Least significant byte

LSN - Least significant nybble

MSB - Most significant bit

MSBy - Most significant byte

MSN - Most significant nybble

Nybble - half of a byte

PAB - Peripheral Access BLock

Pixel - Dot on the screen

PNT - Patern Name Table - Each byte contains the character number for each pattern position on the screen. The first 32 bytes correspond to the first row of patterns on the screen the next 32 bytes to the next row, etc.

RAM - Random Access Memory

SAB - Sprite Attribute Block - contains four bytes for each sprite telling y-pixel position, x-pixel position, character number, and early Clock/color.

SDB - Sprite Descriptor Block - contains 8-byte characters used for sprites

Sprite - objects which exist in planes in front of the pattern plane. Moved on pixel-by-pixel basis.

SVB - Sprite Velocity Block - Contains four bytes for each moving sprite on the screen. The first two bytes give Y velocity and X velocity, respectively. The last two bytes must be initialized to zero by the system, but are used by the system as counters for auto-motion.

VDP - Video Display Processor

VSPTR - Floating point stack pointer

FIGURE 4

TITLE _____

SET 0	SET 1	SET 2	SET 3	SET 4	SET 5	SET 6	SET 7	SET 8	SET 9	SET 10
RAM > 800	RAM > 840	RAM > 880	RAM > 8C0	RAM > 900	RAM > 940	RAM > 980	RAM > 9C0	RAM > A00	RAM > A40	RAM > A80
COLOR:	COLOR:	COLOR:	COLOR:	COLOR:	COLOR:	COLOR:	COLOR:	COLOR:	COLOR:	COLOR:
>00	>08	>10	>18	Sp >20	(>28	0 >30	8 >38	@ >40	H >48	P >50
>01	>09	>11	>19	! >21	) >29	1 >31	9 >39	A >41	I >49	Q >51
>02	>0A	>12	>1A	" >22	* >2A	2 >32	: >3A	B >42	J >4A	R >52
>03	>0B	>13	>1B	# >23	+ >2B	3 >33	; >3B	C >43	K >4B	S >53
>04	>0C	>14	>1C	$ >24	, >2C	4 >34	< >3C	D >44	L >4C	T >54
>05	>0D	>15	>1D	% >25	- >2D	5 >35	= >3D	E >45	M >4D	U >55
>06	>0E	>16	>1E	& >26	. >2E	6 >36	> >3E	F >46	N >4E	V >56
>07	>0F	>17	>1F	' >27	/ >2F	7 >37	? >3F	G >47	O >4F	W >57

SET 11	SET 12	SET 13	SET 14	SET 15	SET 16	SET 17	SET 18	SET 19	SET 20	SET 21
RAM > AC0	RAM > B00	RAM > B40	RAM > B80	RAM > BC0	RAM > C00	RAM > C40	RAM > C80	RAM > CC0	RAM > D00	RAM > D40
COLOR:	COLOR:	COLOR:	COLOR:	COLOR:	COLOR:	COLOR:	COLOR:	COLOR:	COLOR:	COLOR:
X >58	>60	>68	>70	>78	>80	>88	>90	>98	>A0	>A8
Y >59	>61	>69	>71	>79	>81	>89	>91	>99	>A1	>A9
Z >5A	>62	>6A	>72	>7A	>82	>8A	>92	>9A	>A2	>AA
[>5B	>63	>6B	>73	>7B	>83	>8B	>93	>9B	>A3	>AB
\ >5C	>64	>6C	>74	>7C	>84	>8C	>94	>9C	>A4	>AC
] >5D	>65	>6D	>75	>7D	>85	>8D	>95	>9D	>A5	>AD
^ >5E	>66	>6E	>76	>7E	>86	>8E	>96	>9E	>A6	>AE
_ >5F	>67	>6F	>77	>7F	>87	>8F	>97	>9F	>A7	>AF

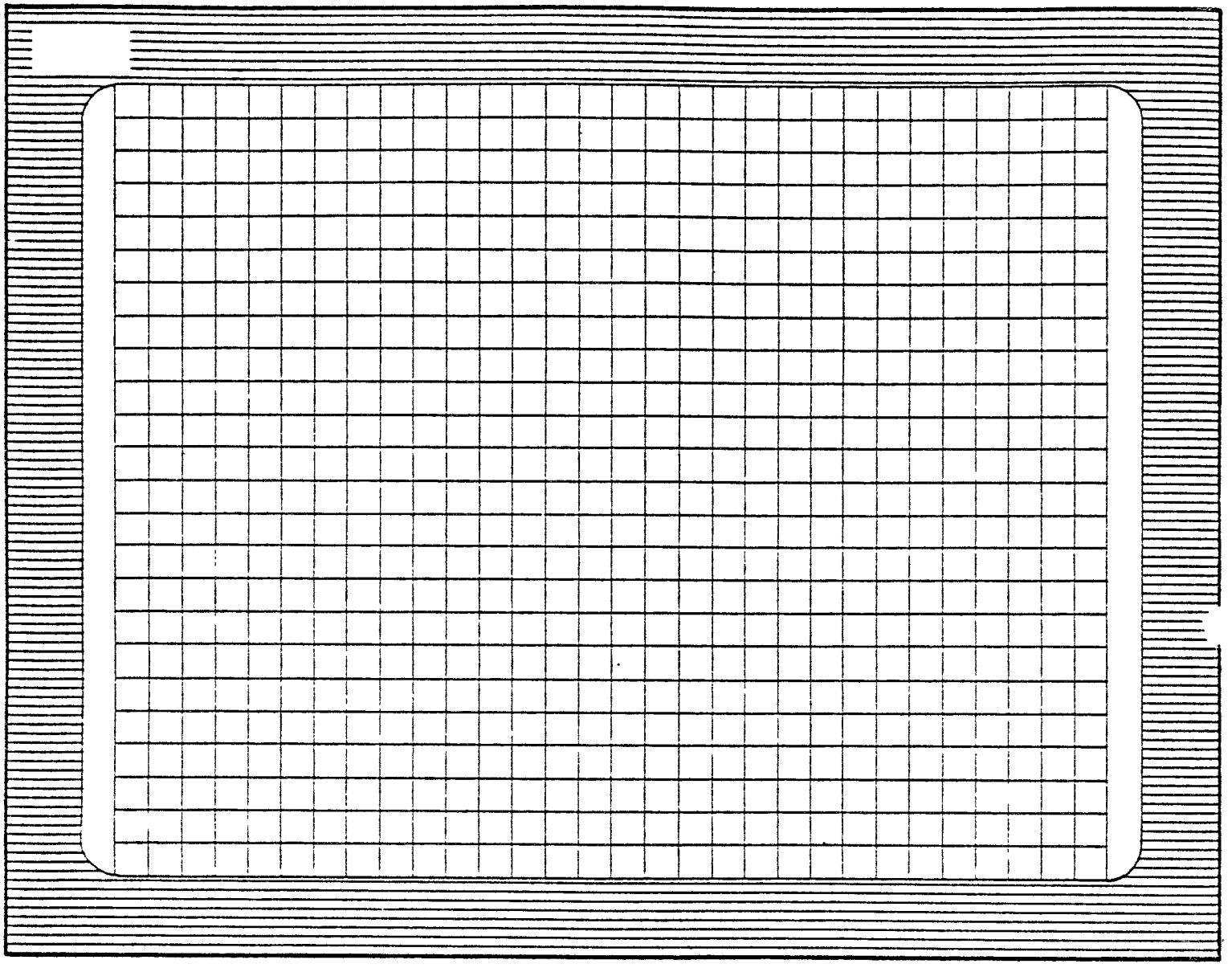
SET 22	SET 23	SET 24	SET 25	SET 26	SET 27	SET 28	SET 29	SET 30	SET 31
RAM > B80	RAM > DC0	RAM > E00	RAM > E40	RAM > E80	RAM > EC0	RAM > F00	RAM > F40	RAM > F80	RAM > FC0
COLOR:	COLOR:	COLOR:	COLOR:	COLOR:	COLOR:	COLOR:	COLOR:	COLOR:	COLOR:
>B0	>B8	>C0	>C8	>D0	>D8	>E0	>E8	>F0	>F8
>B1	>B9	>C1	>C9	>D1	>D9	>E1	>E9	>F1	>F9
>B2	>BA	>C2	>CA	>D2	>DA	>E2	>EA	>F2	>FA
>B3	>BB	>C3	>CB	>D3	>DB	>E3	>EB	>F3	>FB
>B4	>BC	>C4	>CC	>D4	>DC	>E4	>EC	>F4	>FC
>B5	>BD	>C5	>CD	>D5	>DD	>E5	>ED	>F5	>FD
>B6	>BE	>C6	>CE	>D6	>DE	>E6	>EE	>F6	>FE
>B7	>BF	>C7	>CF	>D7	>DF	>E7	>EF	>F7	>FF

FIGURE M.5

TEXAS INSTRUMENTS
INCORPORATED

HOME COMPUTER "GROM" DEVELOPMENT

PROJECT:



VIDEO/COPY:

AUDIO/TONES:

ALPHABETICAL INDEX

USER'S RESPONSE SHEET

Graphics Programming Language User's Guide

April 30, 1979

User's Name_____ Telephone_____

COMPANY_____ Date_____

Please list any discrepancy found in this manual by page, paragraph, figure, or table number in the following space. If there are any other suggestions that you wish to make, feel free to include them. Thank you.

LOCATION IN MANUAL	COMMENT/SUGGESTION