SECTION III

GRAPHICS GENERATION SOFTWARE

The graphics generation process is structured on three levels of software. A typical application will use routines from all three levels. These are the chip driver level, the table level and the object level. Figure 3-1 shows the program flow, software structure and its relationship with the outside world.

3.1 Chip Driver Level

The graphics hardware consists of the VDP and 16K VRAM. The VDP has eight write-only control registers and one read-only status register. The chip driver level software interfaces with the VDP registers and VRAM through the VDP. For detailed configuration of the registers, refer to the TMS 9928A VDP Data Manual.

The chip driver level software consists of six subroutines:

READ_VRAM, WRITE_VRAM, READ_REGISTER, WRITE_REGISTER, FILL_VRAM and MODE_1. The first five routines allow programs to access the VDP registers and transfer information to and from VRAM blocks. The sixth routine, MODE_1, initializes the VDP into a standard configuration.

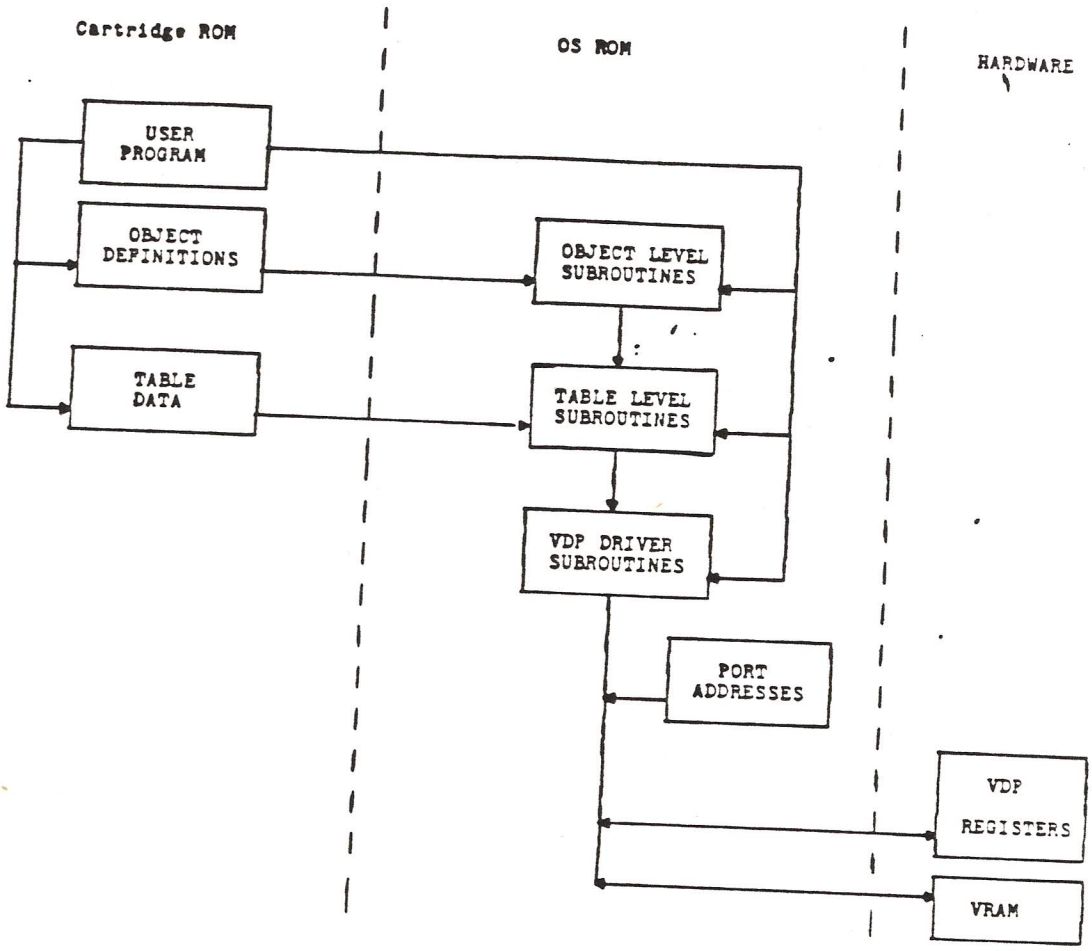

Figure 3-1
OS Graphics Software/VDP Interface

3.1.1. READ_VRAM

Calling Sequence:

```
LD    HL, BUFFER
LD    DE, SRCE
LD    BC, COUNT
CALL  READ_VRAM
```

Description:

READ_VRAM reads COUNT bytes from VRAM starting at SRCE and puts them in BUFFER.

Parameters:

BUFFER	This is the starting address of a CRAM buffer which is to receive the data read from VRAM.
SRCE	VRAM starting address to be read from.

COUNT — Number of bytes to be read from VRAM.

Side Effects:

- Destroys AF, BC, DE and HL.
- Cancels any previously initiated VDP operations.

3.1.2 WRITE_VRAM

Calling Sequence:

```
LD    HL, BUFFER
LD    DE, DEST
LD    BC, COUNT
CALL  WRITE_VRAM
```

Description:

WRITE_VRAM takes COUNT bytes from BUFFER and sends them through the VDP to VRAM. The starting address in VRAM for the write operation is given as DEST.

Parameters:

BUFFER	This is the starting address of a buffer where data to be sent to the VDP is located.

DEST	This is the VRAM address where the data is to be sent.
COUNT	This is the number of bytes that are to be transferred to VRAM. Count should be either less than 256 (100H) or even multiples of 256. (Ref. ColecoVision Bulletin No. 0002).

Side Effects:

- Destroys AF, BC, DE and HL.
- Cancels any previously initiated VDP operations.

3.1.3 READ_REGISTER

Calling Sequence:

```
CALL  READ_REGISTER
```

Description:

READ_REGISTER reads and returns the contents of the VDP status register in the accumulator. This value should be stored at VDP_STATUS_BYTE in CRAM. The information in this register can only be guaranteed valid during the vertical retrace time.

Return value:

Returns the contents of the VDP status register which has the following form (see VDP manual for further details):

4.1.7 EOS Entry Points

```
ADD816              EQU 0FD4DH ;P
BLK_STRT_PTR        EQU 0FDDCH ;D
BLOCKS_REQ          EQU 0FE0CH ;D
BUF_END             EQU 0FE0AH ;D
BUF_START           EQU 0FE08H ;D
BYTES_REQ           EQU 0FE02H ;D
BYTES_TO_GO         EQU 0FE04H ;D
CALC_OFFSET         EQU 0FD32H ;P
CLEAR_RAM_SIZE      EQU 00147H ;A
CLEAR_RAM_START     EQU 0FD60H ;D
COLORTABLE          EQU 0FD6CH ;D
CONTROLLER_0_PO     EQU 0FC2BH ;A
CONTROLLER_1_PO     EQU 0FC2CH ;A
CURRENT_DEV         EQU 0FD6FH ;D
CURRENT_PCB         EQU 0FD70H ;D
CURSOR              EQU 0FEA5H ;D
CUR_BANK            EQU 0FD6EH ;D
DCB_IMAGE           EQU 0FD8BH ;D
DECLSN              EQU 0FD44H ;P
DECMSN              EQU 0FD47H ;P
DEFAULT_BT_DEV      EQU 0FD6FH ;D
DEVICE_ID           EQU 0FD72H ;D
DIR_BLOCK_NO        EQU 0FDD9H ;D
EFFECT_OVER         EQU 0FD5CH ;P
EOS_DAY             EQU 0FDE2H ;D
EOS_MONTH           EQU 0FDE1H ;D
EOS_STACK           EQU 0FE58H ;D
EOS_YEAR            EQU 0FDE0H ;D
FCB_BUFFER          EQU 0FDBAH ;D
FCB_DATA_ADDR       EQU 0FDFFH ;D
FCB_HEAD_ADDR       EQU 0FDFDH ;D
FILENAME_CMPS       EQU 0FDD8H ;D
FILE_COUNT          EQU 0FDD4H ;D
FILE_NAME_ADDR      EQU 0FD73H ;D
FILE_NUMBR          EQU 0FDD7H ;D
FILL_VRAM           EQU 0FD26H ;P
FMGR_DIR_ENT        EQU 0FDE3H ;D
FNUM                EQU 0FE01H ;D
FOUND_AVAIL_ENT     EQU 0FDD8H ;D
GET_VRAM            EQU 0FD2FH ;P
INIT_TABLE          EQU 0FD29H ;P
INT_VCTR_TBL        EQU 0FBFFH ;A
KEYBOARD_BUFFER     EQU 0FD75H ;D
LINEBUFFER_         EQU 0FE7EH ;D
LOAD_ASCII          EQU 0FD38H ;P
MEM_CNFG00          EQU 0FC17H ;A
MEM_CNFG01          EQU 0FC18H ;A
MEM_CNFG02          EQU 0FC19H ;A
MEM_CNFG03          EQU 0FC1AH ;A
MEM_CNFG04          EQU 0FC1BH ;A
MEM_CNFG05          EQU 0FC1CH ;A
MEM_CNFG06          EQU 0FC1DH ;A
MEM_CNFG07          EQU 0FC1EH ;A
MEM_CNFG08          EQU 0FC1FH ;A
MEM_CNFG09          EQU 0FC20H ;A
MEM_CNFG0A          EQU 0FC21H ;A
MEM_CNFG0B          EQU 0FC22H ;A
MEM_CNFG0C          EQU 0FC23H ;A
MEM_CNFG0D          EQU 0FC24H ;A
MEM_CNFG0E          EQU 0FC25H ;A
MEM_CNFG0F          EQU 0FC26H ;A
MEM_SWITCH_PORT     EQU 0FC27H ;A
MOD_FILE_COUNT      EQU 0FDD5H ;D
MSNTOLSN            EQU 0FD4AH ;P
NET_RESET_PORT      EQU 0FC28H ;A
NEW_HOLE_SIZE       EQU 0FE1AH ;D
NEW_HOLE_START      EQU 0FE16H ;D
NUM_COLUMNS         EQU 0FEA0H ;D
NUM_LINES           EQU 0FE9FH ;D
OLDCHAR_            EQU 0FE79H ;D
PATTRNGENTBL        EQU 0FD6AH ;D
PATTRNNAMETBL       EQU 0FD68H ;D
PCB                 EQU 0FEC0H ;A
PERSONAL_DEBOUN     EQU 0FE5AH ;D
PLAY_IT             EQU 0FD56H ;P
POLLER              EQU 0FD3EH ;P
PORT_COLLECTION     EQU 0FD11H ;P
PORT_TABLE          EQU 0FC27H ;A
PRINT_BUFFER        EQU 0FD76H ;D
PTRN_NAME_TBL       EQU 0FEA3H ;D
PTR_TO_LST_OF_S     EQU 0FE6EH ;D
PTR_TO_S_ON_0       EQU 0FE70H ;D
PTR_TO_S_ON_1       EQU 0FE72H ;D
PTR_TO_S_ON_2       EQU 0FE74H ;D
PTR_TO_S_ON_3       EQU 0FE76H ;D
PUT_ASCII           EQU 0FD17H ;P
PUT_VRAM            EQU 0FD2CH ;P
PX_TO_PTRN_POS      EQU 0FD35H ;P
QUERY_BUFFER        EQU 0FDA0H ;D
READ_REGISTER       EQU 0FD23H ;P
READ_VRAM           EQU 0FD1DH ;P
RETRY_COUNT         EQU 0FDD6H ;D
REV_NUM             EQU 0FD60H ;D
SAVE_CTRL           EQU 0FE78H ;D
SECTORS_TO_INIT     EQU 0FD86H ;D
SECTOR_NO           EQU 0FD87H ;D
SOUNDPORT           EQU 0FC2FH ;A
SOUNDS              EQU 0FD59H ;P
SOUND_INIT          EQU 0FD50H ;P
SPIN_SW0_CT         EQU 0FE58H ;D
SPIN_SW1_CT         EQU 0FE59H ;D
SPRITEATTRTBL       EQU 0FD64H ;D
SPRITEGENTBL        EQU 0FD66H ;D
START_BLOCK         EQU 0FE12H ;D
STROBE_RESET_PO     EQU 0FC2EH ;A
STROBE_SET_PORT     EQU 0FC2DH ;A
SWITCH_MEM          EQU 0FD14H ;P
SWITCH_TABLE        EQU 0FC17H ;A
TEMP_STACK          EQU 0FE6EH ;D
TURN_OFF_SOUND      EQU 0FD53H ;P
UPDATE_SPINNER      EQU 0FD41H ;P
UPPER_LEFT          EQU 0FEA1H ;D
USER_BUF            EQU 0FE06H ;D
USER_NAME           EQU 0FE10H ;D
VDP_CTRL_PORT       EQU 0FC29H ;A
VDP_DATA_PORT       EQU 0FC2AH ;A
VDP_MODE_WORD       EQU 0FD61H ;D
VDP_STATUS_BYTE     EQU 0FD63H ;D
VECTOR_08H          EQU 0FBFFH ;A
VECTOR_10H          EQU 0FC02H ;A
VECTOR_18H          EQU 0FC05H ;A
VECTOR_20H          EQU 0FC08H ;A
VECTOR_28H          EQU 0FC0BH ;A
```

Bit 7	Bit 6	Bit 5	Bits 4..0
Interrupt	Fifth Sprite	Coincidence	Fifth Sprite No.

Figure 3-2

VDP Status Register

Side Effects:

This routine has no effect at all in the processor memory or register space. However, a status read has a significant side effect to the VDP.

It acts as an interrupt acknowledge operation, i.e., it clears the interrupt flag and enables further generation of interrupts.

Thise side effect must be treated with care for two reasons. First of all, as is pointed out in the VDP manual, asynchronous reads may cause the interrupt flag in the status register to be reset before it is detected; this may cause problems in systems that expect to perform synchronization using the interrupt flag.

The second reason concerns interrupts which halt the execution of routines while they are accessing VRAM. In order to re-enable interrupts, a service routine must read the status register. However, to prevent the NMI from re-interrupting the service routine, the user should avoid reading the status register until all of its work is done. A defer interrupt routine, DEF_INT, has been developed to assist the user in handling this situation. Refer to ColecoVision Bulletin No. 0010 for additional information.

3.1.4 WRITE_REGISTER

Calling Sequence:

```
LD    B, REGISTER
LD    C, VALUE
CALL  WRITE_REGISTER
```

Description:

WRITE_REGISTER takes VALUE and writes it to the VDP register numbered REGISTER.

WRITE_REGISTER also maintains two bytes in CRAM starting at address VDP_MODE_WORD. The first is intended to duplicate the current contents of VDP Register 0, and the second to duplicate Register 1. When writing to a register using WRITE_REGISTER, the appropriate half of VDP_MODE_WORD is updated.

Parameters:

REGISTER	This is the VDP register number (0 - 7) to be written.
VALUE	This is the value to be written to REGISTER.

Side Effects:

- Destroys the AF register pair.

3.1.5 FILL_VRAM

Calling Sequence:

```
LD    HL, ADDRESS
LD    DE, COUNT
LD    A, VALUE
CALL  FILL_VRAM
```

Description:

FILL_VRAM writes COUNT copies of VALUE to VRAM starting at ADDRESS.

Parameters:

ADDRESS	VRAM address to start fill operation.
COUNT	Number of bytes to fill.

VALUE 8-bit value to fill with.

Side Effects:

- Destroys AF and DE.
- Cancels all previously initiated VRAM operations.

Calls to other OS routines:

- READ_REGISTER (Ref. Sec. 3.1.3)

3.1.6 MODE_1

Calling Sequence:

```
CALL  MODE_1
```

Description:

MODE_1 sets the VDP to graphics mode 1 and sprite size 0. It also uses the INIT_TABLE routine to define the VRAM table addresses as follows:

- Sprite Generator Table	- 3800H
- Patter Color Table	- 2000H
- Sprite Attribute Table	- 1B00H
- Pattern Name Table	- 1800H
- Pattern Generator Table	- 0000H

When MODE_1 returns, the screen is blanked and the backdrop plane color is set to black.

Side Effects:

- Destroys AF, BC and HL.

Calls to other OS routines:

- WRITE_REGISTER
- INIT_TABLE

3.2 Table Level

The VDP requires various table areas within VRAM to operate. These tables are interrelated, each controlling its own aspect of the graphics generation process. The table level software provides routines which will read or write VRAM with respect to these table areas. The routines also provide the capability of reading and writing entire tables entries or sections of these entries up to and including the whole table. This level also has special functions which were found helpful.

The major difference between the table level and the chip driver level is that the applications programmer is no longer required to manipulate VRAM addresses on the table level. Instead, each of the VRAM tables is assigned a number or table code as listed in Table 3-1.

Table Name	Code
Sprite attribute table	0
Sprite generator table	1
Pattern name table	2
Pattern generator table	3
Pattern color table	4

Table 3-1

VRAM Table Code

When an applications program needs to operate on a table, only a table code needs to be passed to the applicable table processing the routine.

Furthermore, in graphics mode 1 and graphics mode 2, which are supported by the OS graphics software, the tables have more or less fixed shapes. The entry numbers and bytes per entry for each of the five tables, as well as their boundaries, is given in Table 3-2.

TABLE CODE	MODE(S)	ENTRIES	BYTES/ ENTRY	HEX EQUIVALENTS
0	1 & 2	32	4	80H
1	1 & 2	256	8	800H
2	1 & 2	768	1	400H
3	1	256	8	800H
3	2	768	8	800H
4	1	32	1	40H
4	2	768	8	2000H

Table 3-2
Table Entries and Boundaries

The table management software takes advantage of this regularity by letting application programs address table entries as integral entities. Let us take, for example, the task of getting the 14th sprite attribute entry from VRAM. In terms of the chip driver software, the task appears as follows:

- Get sprite attribute table address.
- Calculate offset into table (14 * table row length).
- Add offset to address.
- Read one table entry (4 bytes) from VRAM at off set + attribute table address.

On the other hand, when using the table level software, the task is now reduced to the following:

- Give offset into table (14).
- Give table code.
- Give item count (1).
- Call GET_VRAM (places the desired bytes at a user-defined area).

In a video program that requires accessing the sprite attribute table frequently (for example, an action-oriented game), the table level method constitutes a significant savings in cartridge code.

Software in the table level may be further subdivided into three groups of routines as follows:

- Table Managers
- Table-oriented Graphics Routines
- Sprite Reordering Software

3.2.1 Table Managers

There are three routines in this group: INIT_TABLE, GET_VRAM and PUT_VRAM. As the names imply, they deal with table initialization, getting data from tables and placing data into tables, respectively.

Table initialization is a very simple operation which involves assigning a base address to a table. The base addresses are "saved" for later use by GET_VRAM and PUT_VRAM for address calculations and remain fixed until they are reinitialized. GET_VRAM and PUT_VRAM both take a table code, an entry number, as well as an element count and a buffer address in CRAM as parameters when they perform their respective transfers of information between CRAM and VRAM.

3.2.1.1 INIT_TABLE

Calling Sequence:

```
LD    A, TABLE_CODE
LD    HL, ADDRESS
CALL  INIT_TABLE
```

Description:

INIT_TABLE takes a table code and a VRAM address at which that table is to reside, and initializes the VDP base address register for the given table. It also stores the unconverted form of the address in an array called VRAM_ADDR_TABLE for later use in address arithmetic. This address is stored at VRAM_ADDR_TABLE [TABLE_ CODE].

INIT_TABLE makes use of the current graphics mode in determining the actual value written to the base address register in some cases. It determines the graphics mode

by looking at the VDP_MODE_WORD. Thus, it is imperative that the graphics mode be set up using WRITE_ REGISTER before INIT_TABLE.

Parameters:

TABLE_CODE	Number of the table to be initialized. TABLE_CODE must be one of the legal table codes defined in Table 3-1.
ADDRESS	Intended VRAM address of table. Each table has its own boundary defined by the table base address in the VDP control register. The user should refer to Table 3-2 for the proper table boundary.

Side Effects:

- Destroys AF, BC, HL, IX and IY.

Calls to other OS routines:

- REG_WRITE

3.2.1.2 GET_VRAM

Calling Sequence:

```
LD    A, TABLE_CODE
LD    DE, START_INDEX
LD    HL, DATA
LD    IY, COUNT
CALL  GET_VRAM
```

Description:

GET_VRAM reads into the CRAM buffer DATA, COUNT entries from the table specified by TABLE_CODE, which starts at the table entry number START_INDEX.

GET_VRAM uses the VDP_MODE_WORD and VRAM_ADDR_TABLE to calculate VRAM addresses and byte counts. It is imperative, before calling GET_VRAM, that the graphics mode be initialized using WRITE_REGISTER, and that the table being accessed be initialized using INIT_TABLE.

Parameters:

TABLE_CODE — VRAM table code (Table 3-1) to be read.

START_INDEX — START_INDEX is a two-byte number that indicates the starting entry of the table.

The range of START_INDEX is table dependent. However, no boundary checking is done; therefore, if an index is given that is outside the range of the table, but still a legal VRAM address, the specified number of "entries" will be extracted from that location in VRAM.

Both the pattern generator and the color tables in graphics mode 2 are 768 entries long and they are

segmented into three sections corresponding to the three sections of the display. When addressing these tables, the high order byte (D) of the two-byte START_INDEX value is a "segment specifier" (0 <= D <= 2), while the low order byte (E) specifies the index of the entry in that segment.

In the case of the sprite generator table, please note that COUNT refers to 8-byte shape for entries whether one is using size 0 or size 1 sprites.

DATA — Starting address of a CRAM data buffer to receive data from VRAM.

COUNT — Number of entries to be read from the VRAM table.

The restrictions on COUNT are again table dependent. In other words, it should always be the case that START_INDEX + COUNT <= Table Size.

Side Effects:

- Destroys AF, BC, DE, HL, IX and IY.
- This routine uses the local storage area SAVED_COUNT and is therefore not re-entrant.

Calls to other OS routines:

- READ_VRAM

3.2.1.3 PUT_VRAM

Calling Sequence:

```
LD    A, TABLE_CODE
LD    DE, START_INDEX
LD    HL, DATA
LD    IY, COUNT
CALL  PUT_VRAM
```

Description:

PUT_VRAM w ites from the buffer DATA, COUNT entries to the table specified by TABLE_CODE, which starts at the table entry number START_INDEX.

PUT_VRAM uses the VDP_MODE_WORD and VRAM_ADDR_TABLE to calculate VRAM address and byte counts. It is imperative that the graphics mode be set up using WRITE_ REGISTER and the table being accessed be initialized using INIT_TABLE before PUT_VRAM is called.

The table level of graphics software contains a sprite reordering feature where the major effect is in the operation of PUT_VRAM. When the MUX_SPRITES flag is set to TRUE (1), PUT_VRAM writes sprite entries to a CRAM copy of the sprite attribute table instead of writing them to VRAM. It locates this table through a pointer in low cartridge ROM called LOCAL_SPR_TBL. The sprite entries will then be re-ordered before being written to VRAM.

Parameters:

TABLE_CODE	VRAM table code (Refer to Table 3-1) to be written.
START_INDEX	START_INDEX is a two-byte number which indicates the starting entry number of the table. For other considerations, refer to the START_ INDEX parameter of GET_VRAM in Section 3.2.1.2.

DATA — Starting address of a data buffer where data to be written to VRAM resides.

COUNT — Number of entries to be put to the VRAM table.

The restrictions on COUNT are again table dependent. In other words, it should always be the case that START_INDEX + COUNT <= Table Size.

Side Effects:

- Destroys AF, BC, DE, HL, IX and IY.
- Uses local storage locations, SAVE_TEMP and SAVED_COUNT.

Calls to other OS routines:

- WRITE_VRAM

3.2.2 Table-Oriented Graphics Routines

A number of routines are included in the table level graphics software that perform useful operations on generators. Each of these takes a table code, a source index from that table, a destination index in the same table, and the number of entries to be processed. The routines work in read-modify-write mode, that is, they pull the generators out of the table one at a time, process them and put them back. They use a CRAM buffer for their scratch area. This buffer is allocated by the applications programmer and accessable only through the pointer at WORK_BUFFER in cartridge ROM.

With one exception, the routines in this package always process generators one at a time, and write them to the destination block in the same order in which they are extracted from the source block. This has important implications for their use with size 1 sprites.

When the sprite size is 1, the hardware accesses four generators at the index found in a sprite's attribute

table entry and displays them so that they appear on the screen as shown in Figure 3-3.

Sprite Screen Location

first generator	third generator
second generator	fourth generator

Figure 3-3
Sprite Size 1 Orientation

Thus, OS routines operating on the individual generators for a size 1 sprite will not be sufficient to orient the entire object. The four generators that make up the sprite will have to be permuted as well. The applications program will have to include a small routine that performs the required permutation in tandem with the OS call.

The following operations are available in the table-oriented graphics package:

- Reflection about the vertical axis
- Reflection about the horizontal axis
- 90-degree rotation
- Enlargement by a factor of two

3.2.2.1 REFLECT_VERTICAL

Calling Sequence:

```
LD    A, TABLE_CODE
LD    DE, SOURCE
LD    HL, DESTINATION
LD    BC, COUNT
CALL  REFLECT_VERTICAL
```

Description:

REFLECT_VERTICAL takes each generator in a block of COUNT generators following SOURCE in the table indicated by TABLE_CODE and modifies it in such a way that the new generator thus created will appear to be a reflection about the vertical screen axis of the old. The created generators are put back into a block of COUNT generators following DESTINATION in the same table.

The user must provide the permutation for size 1 sprite generators as diagrammed in Figure 3-4 below:

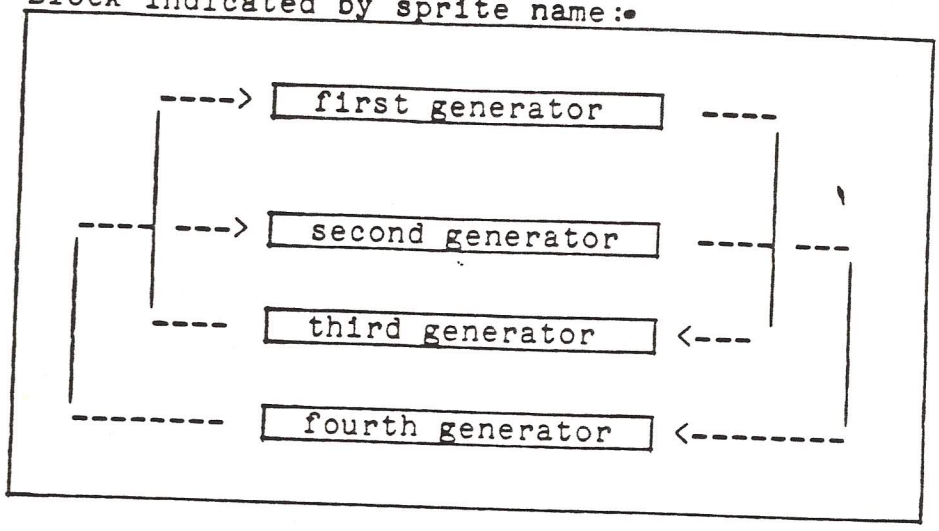

Figure 3-4
REFLECT_VERTICAL Size 1 Sprite Permutation

If TABLE_CODE is 3 (indicating the pattern generator table) and graphics mode 2 is used, REFLECT_VERTICAL also copies the color table entries for each generator it processes. Thus, when it is complete, the two-color table blocks indexed by SOURCE and DESTINATION will be identical. This means that the color scheme for the reflected generators will be the same as that for the originals.

Parameters:

TABLE_CODE — VRAM table code (Ref. Table 3-1) to be operated upon.

SOURCE — SOURCE is the two-byte index of the first entry in the specified table to be operated on.

For table operations of sprite generator or pattern generator in graphics mode 1, SOURCE should be in the range 0 <= SOURCE <= 255. For pattern generators in mode 2, it should be in the range 0 <= SOURCE <= 767. In either case, if a value of SOURCE supplied is outside the table's range but still is a legal VRAM address, the specified number of "entries" will be read and modified from the VRAM location (table location) + 8 *

SOURCE. For the proper table entries and table boundary, refer to Table 3-2.

Sprite size has no effect on the range of SOURCE.

DESTINATION (HL)

DESTINATION indexes the place where REFLECT_VERTICAL will start putting generators back into VRAM after modifying them.

The same restrictions apply to the value of DESTINATION as to the value of SOURCE. They are both intended to be indices into the same generator table.

COUNT (BC)

A two-bytes count of the number of entries to be processed sequentially after SOURCE.

The legal value for COUNT is dependent on the size of the table being operated on and the values of SOURCE and DESTINATION. In general, both of the following statements should be true:

COUNT + SOURCE <= (table size)
COUNT + DESTINATION <= (table size)

Side Effects:

- Destroys AF, AF', BC, DE, DE', HL, HL', IX and IY.
- Uses the first 16 bytes of the data area pointed to by WORK_BUFFER.

Calls to other OS routines:

- GET_VRAM
- PUT_VRAM

3.2.2.2 REFLECT_HORIZONTAL

Calling Sequence:

```
LD    A, TABLE_CODE
LD    DE, SOURCE
LD    HL, DESTINATION
LD    BC, COUNT
CALL  REFLECT_HORIZONTAL
```

Description:

REFLECT_HORIZONTAL takes each generator in a block of COUNT generators following SOURCE in the table indicated by TABLE_CODE and modifies it in such a way that the new generator created will appear to be a reflection about the horizontal screen axis of the old. The created generators are placed back into a block of COUNT generators following DESTINATION in the same table.

The user has to provide the permutation for size 1 sprite generators as diagrammed in Figure 3-5.

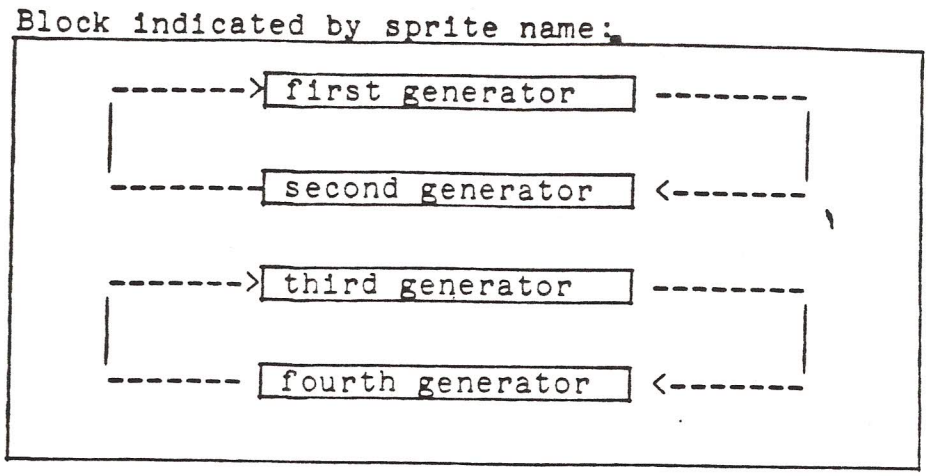

Figure 3-5
REFLECT_HORIZONTAL Size 1 Sprite Permutation

If TABLE_CODE is 3 (indicating the pattern generator table) and the graphics mode is 2, REFLECT_HORIZONTAL also performs the identical reflection on the corresponding color table entry for each generator it processes. This means that the reflected generators will be colored in a way that is consistent with their unreflected counterparts. When in mode 1, the color table is untouched.

Parameters:

TABLE_CODE — VRAM table code (Ref. Table 3-1) to be operated upon.

SOURCE — SOURCE is the two-byte index of the first entry in the specified table to be operated on.

For table operations on sprite generator or pattern generator in graphics mode 1, SOURCE should be in the range 0 <= SOURCE <= 255. For pattern generators in mode 2, it should be in the range 0 <= SOURCE <= 767. In either case, if a value of SOURCE is supplied and is outside the table's range but still a legal VRAM address, the specified number of "entries" will be read and modified from the VRAM location (table location) + 8 * SOURCE. For the proper table entries and table boundary, refer to Table 3-2.

Sprite size has no effect on the range of SOURCE.

DESTINATION

DESTINATION indexes the place where REFLECT_VERTICAL will start putting generators back into VRAM after modification.

The same restrictions apply to the value of DESTINATION as to the value of SOURCE. They are both intended to be indices into the same generator table.

COUNT

A two-byte count of the number of entries to be processed sequentially after SOURCE.

A legal value for count depends on the size of the table being operated on and the values of SOURCE and DESTINATION. In

general, both of the following statements should be true:

COUNT + SOURCE <= (table size)
COUNT + DESTINATION <= (table size)

Side Effects:

- Destroys AF, AF', BC, DE, DE', HL, HL', IX and IY.
- Uses the first 16 bytes of the data area pointed to by WORK_BUFFER.

Calls to other OS routines:

- GET_VRAM
- PUT_VRAM

3.2.2.3 ROTATE_90

Calling Sequence:

```
LD    A, TABLE_CODE
LD    DE, SOURCE
LD    HL, DESTINATION
LD    BC, COUNT
CALL  ROTATE_90
```

Description:

ROTATE_90 takes each generator in a block of COUNT generators following SOURCE in the table indicated by TABLE_CODE and modifies it in such a way that the new generator thus created will appear to be a 90-degree clockwise rotation of the old. The created generators are put back into a block of COUNT generators following DESTINATION in the same table.

The user must provide the permutation for size 1 sprite generators as diagrammed in Figure 3-6 below:

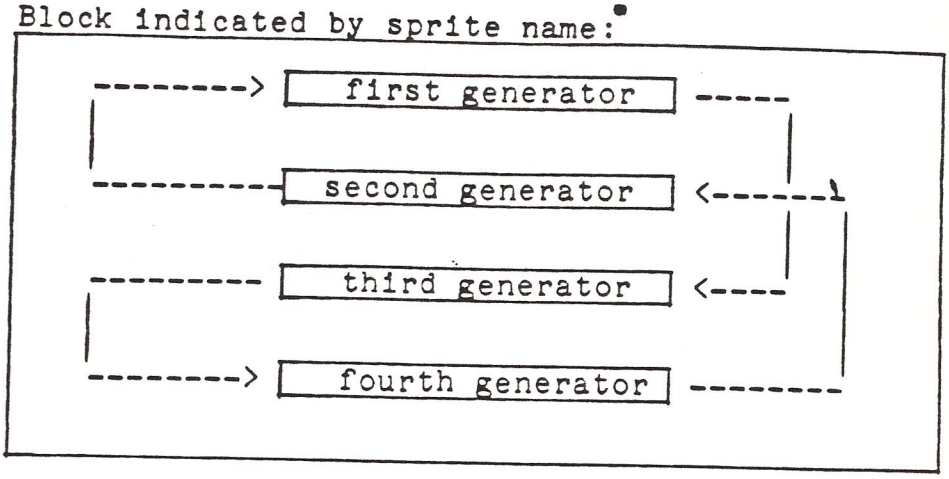

Figure 3-6
ROTATE_90 Size 1 Sprite Permutation

This routine should be used with great care when applied to pattern generators in mode 2. In this mode, the VDP allows arbitrary color combinations along vertical lines while it is still limited to two colors along a given 8-pixel horizontal line. The problem is that if the user attempts to rotate a figure that has more than two colors on a vertical line, ROTATE_90 will exhibit color problems after rotation. There is no way around this problem except to keep any generators that are intended for rotation simple. If the TABLE_CODE is 3 (pattern

generator table) and the mode is 2, ROTATE_90 will copy the corresponding color table entries indexed by SOURCE to the block indexed by DESTINATION.

Parameters:

TABLE_CODE — VRAM table code (Ref. Table 3-1) to be operated upon.

SOURCE — SOURCE is the two-byte index of the first entry in the specified table to be operated on.

For table operations of sprite generator or pattern generator in graphics mode 1, SOURCE should be in the range 0 <= SOURCE <= 255. For pattern generators in mode 2, it should be in the range 0 <= SOURCE <= 767. In either case, if a value of SOURCE is supplied and is outside the table's range but

still is a legal VRAM address, the specified number of "entries" will be read and modified from the VRAM location (table location) + 8 * SOURCE. For the proper table entries and table boundary, refer to Table 3-2.

Sprite size has no effect on the range of SOURCE.

DESTINATION

DESTINATION indexes the place where REFLECT_VERTICAL will start putting generators back into VRAM after modifying them.

The same restrictions apply to the value of DESTINATION as to the value of SOURCE. They are both intended to be indices into the same generator table.

COUNT

A two-byte count of the number of entries to be processed sequentially after SOURCE.

The legal value for count is dependent on the size of the table being operated on and the values of SOURCE and DESTINATION. In general, both of the following statements should be true:

COUNT + SOURCE <= (table size)
COUNT + DESTINATION <= (table size)

Side Effects:

- Destroys AF, AF', BC, DE, DE', HL, HL' IX and IY.
- Uses the first 16 bytes of the data area pointed to by WORK_BUFFER.

Calls to other OS routines:

- GET_VRAM
- PUT_VRAM

3.2.2.4 ENLARGE

Calling Sequence:

```
LD    A, TABLE_CODE
LD    DE, SOURCE
LD    HL, DESTINATION
LD    BC, COUNT
CALL  ENLARGE
```

Description:

ENLARGE takes each generator in a block of COUNT generators following SOURCE in the table indicated by TABLE_CODE and from it creates four generators as shown below in Figure 3-7.

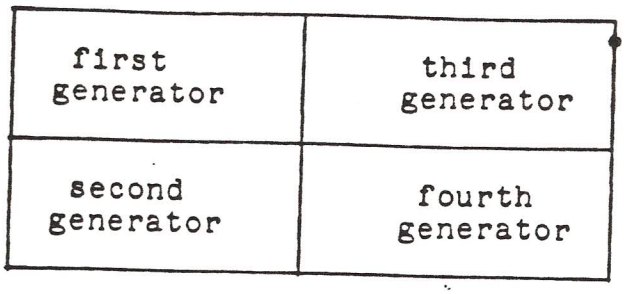

Figure 3-7
ENLARGE Generators Layout

The enlarged object will appear to be a double-sized version of the original. The created generators are put back into a block of 4 * COUNT generators following DESTINATION in the same table.

Note that since the ordering of the expanded generators is the same as that for the four generators needed to produce a size 1 sprite, ENLARGE lends itself well to use with sprites as long as the programmer is willing to dedicate four times as many sprites to the expanded object as to the orginal.

If TABLE_CODE is 3 (indicating the pattern generator table) and the graphics mode is 2, ENLARGE makes four

copies of the color table entry for each source generator and places them in the color table so that they correspond to the four destination generators. This should mean that the color scheme for the enlarged object will be the same as that of the original. If the mode is 1, the color table is untouched.

Parameters:

TABLE_CODE — VRAM table code (Ref. Table 3-1) to be operated upon.

SOURCE — SOURCE is the two-byte index of the first entry in the specified table to be operated on.

For table operations on a sprite generator or a pattern generator in graphics mode 1, SOURCE should be in the range 0 <= SOURCE <= 255. For pattern generators in mode 2, it should be in the range

0 <= SOURCE <= 767. In either case, if a value of SOURCE is supplied and is outside the table's range but still a legal VRAM address, the specified number of "entries" will be read and modified from the VRAM location (table location) + 8 * SOURCE. For the proper table entries and table boundary, refer to Table 3-2.

Sprite size has no effect on the range of SOURCE.

DESTINATION

DESTINATION indexes the place where ENLARGE will start placing generators back into VRAM after modifying them.

The same restrictions apply to the value of DESTINATION as to the

value of SOURCE. They are both intended to be indices into the same generator table.

COUNT

A two-byte count of the number of entries to be processed sequentially after SOURCE.

The most important factor limiting the size of COUNT in the case of the ENLARGE routine is that ENLARGE actually produces four generators for every generator that it reads.

The legal value for count depends on the size of the table being operated on and the values of SOURCE and DESTINATION. Both of the following statements should be true:

COUNT + SOURCE <= (table size)
DESTINATION + 4 * COUNT) <=
(table size)

Side Effects:

- Destroys AF, AF', BC, DE, DE', HL, HL', IX and IY.
- Uses the first 40 bytes of the data area pointed to by WORK_BUFFER.

Calls to other OS routines:

- GET_VRAM
- PUT_VRAM

3.2.3 Sprite Reordering Software

Probably the most significant hardware limitation of the VDP is the so-called "fifth sprite problem." This problem arises when more than four sprites occur on a single horizontal scan line. Because the chip only has four registers for dealing with the lower order sprites, the sprites with the higher sprite attribute indices cannot be generated on that scan line and therefore disappear.

One solution to this problem is to use a reordering scheme on the offending sprites which involves swapping the priorities of the sprite that is being blanked out with that of one of the higher order sprites in the group on successive video fields. The result is that while the sprites that are being reordered tend to flicker in the area of overlap, they are still quite visible. The degree of flicker depends on many factors including the color of the sprites in question and the background color and complexity.

The OS supports this solution by allowing the application to adjust the order of sprite attribute entries with minimum effort.

Two tables are used in implementing the sprite reordering feature. The first of these is simply a local CRAM version of the VRAM sprite attribute table. It must be allocated by the application program and made accessible to the OS by placing a pointer to it at the predetermined cartridge ROM location LOCAL_SPR_TBL. This local sprite attribute table need only contain the active sprite entries needed by the application and therefore may be shorter than the 128 bytes required for the VRAM version. The other table is called the sprite order table. It is also allocated by the application program through a pointer, SPRITE_ORDER, located in cartridge ROM. The sprite order table should contain one byte for each entry in the local sprite attribute table, and the bytes should take on values in the range $0 <= b <= 31$.

When the flag MUX_SPRITES is false (0), PUT_VRAM writes sprite attribute entries directly to VRAM. However,

when this flag becomes true (1), they are written instead to the local sprite attribute table. Then, a routine called WR_SPR_NM_TBL will map the local sprite attribute entries to VRAM according to the sprite order table.

An example of the relationship between the three tables may be illustrated as follows:

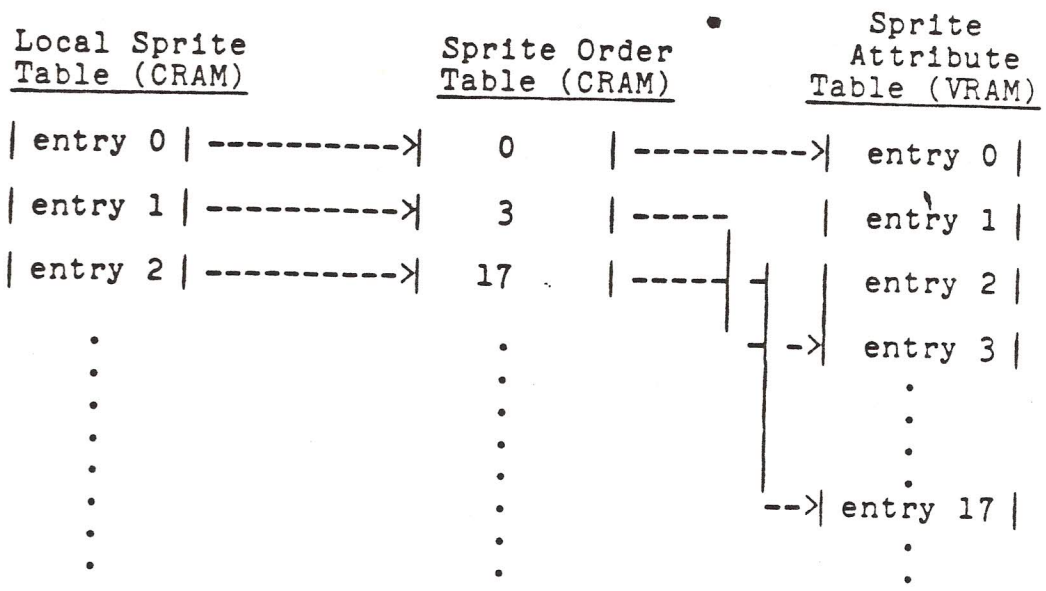

Figure 3-8

Sprite Reordering Table Mapping

The advantage of this method lies in the fact that it takes a lot less work to reorder the bytes in the sprite order table than it does to move around the entries in the VRAM or CRAM sprite attribute tables.

3.2.3.1 INIT_SPR_ORDER

Calling Sequence:

```
LD    A, SPRITE_COUNT
CALL  INIT_SPR_ORDER
```

Description:

INIT_SPR_ORDER looks at the pointer SPRITE_ORDER in low cartridge ROM which should contain the address of a free area SPRITE_COUNT bytes long in CRAM. It sets this area up as a sprite order table by initializing it with zero through SPRITE_COUNT - 1.

Parameters:

SPRITE_COUNT	The length of the sprite order table, which whould be the same as the intended number of entries in the local sprite attribute table.

This number must always be in the range 1 <= SPRITE_COUNT <= 32.

Side Effects:

- Destroys AF, BC, and HL.

3.2.3.2 WR_SPR_NM_TBL

Calling Sequence:

```
LD    A, COUNT
CALL  WR_SPR_NM_TBL
```

Description:

WR_SPR_NM_TBL writes COUNT entries from the local sprite attribute table, which it accesses through the pointer LOCAL_SPR_TBL in low cartridge ROM, to the VRAM sprite attribute table. The transfer is mapped through the sprite order table which it accesses through the pointer SPRITE_ORDER in low cartridge ROM.

Parameters:

COUNT	This is the number of sprite attribute entries to be written to VRAM.

COUNT should not be larger than the initialized length of the sprite order table.

Side Effects:

- Destroys AF, BC, DE, HL, IX and IY.
- Cancels any previously established VDP operations.

3.3 Object Level

The object level software constitutes the top level of the graphics generation software, which appears to the user as a collection of screen objects with well-defined shape, color scheme, and location at any given moment. The software supports four distinct object types, each of which has its own capabilities and limitations. Once objects are defined, however, the rules for manipulating them are fairly type-independent. In fact, only one routine (PUTOBJ),is used to display objects of all types.

Brief descriptions are given in the following sections in regard to object types, object data structures and two user-accessible routines (ACTIVATE, PUTOBJ). For further information, refer to Appendix B.

3.3.1 Object Types

There are four different types of objects defined by the OS. A brief description for each type is given below.

3.3.1.1 Semi-Mobile

Semi-mobile objects are rectangular arrays of pattern blocks which are always aligned on pattern boundaries. Their animation capability is limited. In most cases they are used to set up background pattern graphics.

3.3.1.2 Mobile

The size of a mobile object is fixed in two-by-two pattern blocks. They belong to the pattern plane but can be moved from pixel to pixel in X,Y directions like a sprite superimposed on the background. However, the speed of mobile objects are too slow when compared to the sprites.

3.3.1.3 Sprite

Sprite objects are composed of an individual sprite.

3.3.1.4 Complex

Complex objects are collections of other "component" objects which may be of any type including other complex objects.

3.3.2 Object Data Structure

Each of the above mentioned objects has its definition in cartridge ROM. This high-level definition links together several different data areas which specify all aspects of an object. The data structure is described in detail in Appendix B.

3.3.2.1 Graphics Data Area

This data area is located in cartridge ROM. Pattern and color generators for semi-mobile, mobile and sprite objects and frame data for all objects are located in the graphics data area. The data structure within each graphics area depends on the type of object with which it is associated. If, however, two or more objects of the same type are graphically identical, they may share

the same graphics area. This will reduce the amount of graphics data that needs to be stored in cartridge ROM.

3.3.2.2 Status Area

Each object will have its own status area in CRAM. The game program uses this area to manipulate the object. It does this by altering the location within status which determines which frame is to be displayed as well as the locations which define the position of the object on the display. The graphics routine, PUTOBJ, when called, will access the object's status area and place the object accordingly.

3.3.2.3 OLD_SCREEN

Mobile and semi-mobile objects appear in the pattern plane. They are displayed by altering some of the names in the pattern name table. The original names represent a background which is "underneath" the object. When the object moves or is removed from the pattern plane, the original names must be restored to the name table.

Before placing a semi-mobile or mobile object on the display, PUTOBJ will restore any previously saved names and also save the names which constitute the background underneath the new location of the object. Sprite and complex objects do not need OLD_SCREEN areas.

3.3.3 ACTIVATE

Calling Sequence:

```
LD    HL, OBJ_DEF
SCF
CALL  ACTIVATE
```

or

```
LD    HL, OBJ_DEF
OR    A
CALL  ACTIVATE
```

Description:

The primary purpose of this routine is to move the pattern and color generators from the graphics data area into the pattern and color generator tables in VRAM. Each object must be "activated" before it can be displayed. ACTIVATE also initializes the first byte in an object's OLD_SCREEN data area with the value 80H. PUTOBJ tests this location before restoring the background names to the name table. If the value 80H is found, it is an indication that there are no background names to restore.

Parameters:

OBJ_DEF	High level definition of an object. See Appendix B for further details.
SCF	Carry flag should be set if user wishes to load the generators specified for this object.

```
        OR A
```

Carry flag should be reset if user knows that the generators are already in VRAM.

3.3.4 PUTOBJ

Calling Sequence:

```
        LD      IX, OBJ_DEF
        LD      B, BKGND_SELECT
        CALL    PUTOBJ
```

Description:

PUTOBJ is called when an object's frame or its location on the display is to be changed. The routine tests the type of object and then branches to one of several subroutines designed to handle that particular object type. These routines are not accessible to the user. Their functions are as follows:

1. PUT_SEMI

Semi-mobile objects are placed on the display by writing the generator names specified by one of the object's frames into the pattern name table in VRAM. The pattern and color generators which are needed to create the frame must already be in their respective generator tables.

2. PUT_MOBILE

Mobile objects are displayed by producing a new set of pattern and color generators which depict the frame to be displayed on the background. These new generators are then moved to the locations in the VRAM pattern and color generator tables which are reserved for the object; the names of the new generators are then written into the pattern name ta

3. PUT_SPRITE0

PUT_SPRITE0 handles the display of size 0 sprite objects.

4. PUT_SPRITE1

PUT_SPRITE1 handles the display of size 1 sprite objects.

5. PUT_COMPLEX

PUT_COMPLEX calls PUTOBJ for each of its component objects.

Parameters:

OBJ_DEF	High level definition of an object. See Appendix B for further details.
BCKGND_SELECT	Used with mobile objects or complex objects with a mobile-type component. Can be ignored otherwise. For methods of selecting background colors in a mobile object. Refer to Appendix B for additional information.