# HA-8-3 Graphics and Sound Support Package for the Software Toolworks C/80 Compiler

## Glenn Roberts

## 16 October 2022

# Introduction

The HA-8-3 is a graphics and sound accessory board for the Heathkit H8 digital computer, originally developed by New Orleans General Data Services (NOGDS) and sold by Heath Company. The board performs color video graphics, complex sound waveform generation, analog and digital I/O, and hardware-based floating-point arithmetic [1]. There is also a recently developed clone of the HA-8-3 which performs all the functions of the original board [2].

This document describes a software library (HA83.MAC) that provides support for the Video Display Processor (VDP), Programmable Sound Generator (PSG) and analog-to-digital converter (ADC) functions of the HA-8-3. These routines are designed to be called from the Software Toolworks C/80 language [3]. The routine names are patterned after a library originally developed for the Lucidata Pascal system [4], originally marketed in the 1980s by Polybytes and later placed in the public domain. This library also includes some new routines which expand capability beyond the original Lucidata library.

# Video Display Processor Routines

The HA-8-3 uses the TMS9918A VDP from Texas Instruments and can optionally accommodate the TMS9928A [5]. The 9918 is designed to output composite video signals (NTSC or PAL) suitable for display on a 1980s vintage color monitor (or RF modulator and television). The 9928 is capable of outputting component video (Y, R-Y and B-Y) and, with appropriate hardware such as the Open-Source Scan Converter (OSSC) [6], can be used to produce excellent high-resolution graphics on modern displays via an HDMI interface.

The reader should have at least some familiarity with the TMS9918 documentation [5] prior to using this library.

The VDP library includes four general classes of routines:

**9918 primitives**: Routines to read and set VDP registers and options including the starting address in video RAM (VRAM) for various VDP tables.

**VRAM I/O**: Routines to allocate VRAM and to read and write data to VRAM either as blocks of data or individual bytes.

**Sprites**: Routines to define and manipulate graphical sprites, which are special animation patterns that provide smooth motion and multilevel pattern overlaying.

**Graphics**: Routines to initialize graphics modes, draw and erase pixels and lines, change drawing colors, and fill or erase graphical areas. Two graphics modes are supported: a low resolution (64 x 48 position) Multicolor mode and a high resolution (192 x 256 position) two-color Graphics 2 mode (as defined in the TI documentation [5]). Though there are not explicit routines for the Text and Graphics 1 modes, those modes can still be accessed via the appropriate primitives and VRAM I/O.

## 9918 Primitive Routines

int vdpstatus();

Returns the current value of the VDP status register. This is most useful in conjunction with sprite processing since the status register provides indication of sprite coincidence (overlap of one or more pixels between any two sprites) and excess sprites on a horizontal line (only four are allowed on any line of the screen).

vdpoptions(opt);

Uses the int opt to populate the two options registers in the VDP. The most significant byte of opt is written to VDP register 0 and the least significant byte to register 1. These two registers define key behaviors of the VDP including the type of VRAM allocated, whether the screen is blanked, enabling of interrupts, and graphics mode.

It is important to use the predefined constants from the file ha83.h when using this routine.

Example:

/\* Initialize VDP registers:

\*\*

\*\* No External Video

\*\* 16K RAM

\*\* Disable display

\*\* Pattern mode (Graphics I)

\*\* S1 sprites (16x16)

\*\* M1 Magnified sprites (32x32)

\*/

vdpoptions(VPNEV+VP16K+VPDDP+VPPM+VPS1+VPM1);

The VDP has five registers specifying the base addresses (in VRAM) of five respective tables and data structures used to define the display. For each of these registers there is a routine to specify the associated base address, as shown below. Care must be taken to specify non-overlapping segments for the different tables. The VRAM address space is 16K (0 .. 16,383, or 214) so all tables must be accommodated in that space. Though address values are passed as 16-bit (int) values, only the lower 14 bits are used in defining the address values in VRAM, and since the VRAM sub-blocks must fall on boundaries that are a specified power of two, only a portion of the 14 bits are used in each register.

pattnametable(addr);

Defines the base address of the Name Table sub-block in VRAM. The upper 4 bits (of the 14-bit address) are stored in register 2 of the VDP. The entries in the Name Table are used to index colors and patterns in the respective tables defined below.

colgentable(addr);

Defines the base address of the Color Table sub-block in VRAM. The upper 8 bits (of the 14-bit address) are stored in register 3 of the VDP.

pattgentable(addr);

Defines the base address of the Pattern, Text or Multicolor Generator sub-block in VRAM. The upper 3 bits (of the 14-bit address) are stored in register 4 of the VDP. This table contains the actual pixel patterns for each named entry.

sprnametable(addr);

Defines the base address of the Sprite Attribute Table sub-block in VRAM. The upper 7 bits (of the 14-bit address) are stored in register 5 of the VDP. The Sprite Attribute Table specifies information about each of the sprites. Each entry is 4 bytes and there are a maximum of 32 sprites, so the table is 128 bytes long.

sprpatrntable(addr);

Defines the base address of the Sprite Pattern Generator sub-block in VRAM. The upper 3 bits (of the 14-bit address) are stored in register 6 of the VDP. This table contains the actual pixel patterns for each named sprite.

Example:

#define PGTAB 0

#define PNTAB 6144

#define SNTAB 7168

#define CGTAB 8192

#define SPTAB 14336

…

/\* set up table addresses \*/

pattgentable(PGTAB);

colgentable(CGTAB);

pattnametable(PNTAB);

sprnametable(SNTAB);

sprpatrntable(SPTAB);

There are recommended table addresses that are specified to work well with the Multicolor and Graphics 2 modes implemented in this library. These addresses are shown in the table below (where K=1024):

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Sub-Block** | **Boundary** | | **Size** | |
| **(Decimal)** | **(Hex)** | **(Decimal)** | **(Hex)** |
| Pattern Gen. | 0 | 0 | 6K | 0x1800 |
| Pattern Name | 6K | 0x1800 | 768 | 0x0300 |
| Sprite Attrib. | 7K | 0x1C00 | 128 | 0x0080 |
| Pattern Color | 8K | 0x2000 | 6K | 0x1800 |
| Sprite Gen. | 14K | 0x3800 | 2K | 0x0800 |

The easiest way to use these default settings is to call one of the built-in initialization routines, either vramallocate(), initmcmode() or initg2mode().

vramallocate();

Calls the routines to initialize the VDP registers according to the table above. This routine also performs the following initializations:

* Display is disabled.
* Mode is set to Graphics 1.
* Background color is set to black.
* Sprite size is set to small with no magnification.
* The sprite table is disabled (no sprites displayed).

bordercolor(color);

Sets the screen backdrop color to the color specified in the lower 4 bits of the color argument by storing the value in register 7 of the VDP. The 16 possible color values (c\_transp, c\_black … c\_white) are defined in the include file ha83.h.

## VRAM I/O Routines

The HA-8-3’s video RAM is not addressable via the H8 buss, rather data must be moved in and out via the VDP. Four routines are provided for this purpose: two to move blocks of data (read and write) and two to store and retrieve individual bytes. Since the VDP constantly updates the display contents from the data stored in VRAM, changing the VRAM contents immediately changes the screen content (e.g. pixel state and/or color, sprite visibility and location, etc.)

blockwrite(sourceaddress, vramaddress, count);

blockwrite transfers count bytes from the H8’s system RAM location sourceaddress to the VRAM location vramaddress. It is important to ensure that the destination in VRAM is large enough to hold count bytes, otherwise unexpected behavior is likely to occur.

blockread(destaddress, vramaddress, count);

blockread transfers count bytes from VRAM location vramaddress to the H8’s system RAM location destaddress. It is important to ensure that the destination array or data structure is large enough to hold the data, otherwise unexpected behavior is likely to occur.

wrtvramdirect(address, data);

This routine stores the least significant byte of the int data into the VRAM location specified by address.

rdvramdirect(address);

This routine returns an int with the least significant byte set to the VRAM value contained at address.

## Sprite Routines

The VDP’s ability to display and manipulate sprites provides a powerful tool for smooth, high-speed animation of display objects. It is important to understand some of the basics of sprite usage:

* Up to 32 sprites may be displayed simultaneously on the screen
* Sprites are displayed on overlapping video planes with 0 being the front-most (highest priority) plane and 31 being the rear-most (lowest priority). Higher priority sprites obscure those behind them (unless their color is set to transparent).
* Sprites are not active in Text mode.
* Sprite locations are defined by the location of the top left corner of the sprite. Their position and color are specified in the respective Sprite Attribute Table entry. Changes to the x and y location in the table cause the associated sprite to move instantly to that location. When the x entry in the table is set to the special value 208 (hexadecimal 0xD0) sprite processing is terminated. This allows the Sprite Attribute Table to be shortened and also allows the user to blank all or part of the sprites by changing a single byte in VRAM.
* A maximum of four sprites can be displayed on a horizontal line. If this rule is violated the four highest priority sprites are displayed normally. The fifth and subsequent sprites are not displayed on that line, however the ‘fifth-sprite’ bit in the VDP status register is set and the sprite number for the violating sprite is stored there as well. This lets a user detect this sprite overload condition and take appropriate action (if any).
* The VDP can detect when the patterns of two sprites overlap and sets the ‘coincidence’ bit in the VDP status register to indicate this condition. This lets a user detect “collision” of two sprite objects and take appropriate (if any).
* Standard sprites are 8x8 pixels in size. Large sprites are 16x16 pixels in size. Sprite size is specified by a bit in VDP register 1 and can be set via the vdpoptions() routine.
* Sprites can also be magnified. A magnified sprite is displayed with two pixels for each bit in the pattern. Magnified standard sprites are therefore 16x16 pixels in size. Magnified large sprites are 32x32 pixels in size. Sprite magnification is specified by a bit in the VDP register 1 and can be set via the vdpoptions() routine.
* Each sprite can have an “early” bit set. This allows sprites smoothly enter and exit the screen.
* There can be at most 32 sprites displayed at a time but there can be many more sprite patterns defined in the Sprite Pattern Generator sub-block of VRAM. For a standard configuration with 2,048 bytes of VRAM allocate to sprite patterns, up to 256 8-byte patterns (all possible patterns) could be stored for defining standard 8x8 sprites. Similarly, up to 64 large sprite patterns could be defined.
* The size and magnification aspects of sprites for a given application are usually established when the graphics system is initially configured using the vramallocate(), initmcmode() or initg2mode() routines, but they can also be explicitly set using the vdpoptions() call.

Defining a sprite requires two steps: 1) creating an entry in the appropriate record of the Sprite Attribute Table. The entry contains vertical and horizontal position information, the sprite name (index into the Sprite Generator Table) and a tag byte containing the color and the “early clock” bit. 2) creating an entry in the Sprite Generator Table and connecting it to the name field in the Sprite Attribute Table. The size of the pattern in the Sprite Generator Table should correspond to the sprite size specified in the VDP options (eight bytes for 8x8 sprites; 32 bytes for 16x16 sprites). The sprite position entries are based on a coordinate system with the upper left corner of the screen as (0,0) and the lower right as (255,191).

defsprite(sprnumber, x, y,color, early, pattern);

This routine performs both steps described above. All arguments are of type int except pattern which is a char\* (i.e. a pointer to an array of bytes.) The routine creates an entry in the Sprite Attribute Table using sprnumber as the sprite number (0..31), and x, y, color and early to define the four table entries. The color value is used for pixels that have a bit set (1) in the corresponding pattern definition. Pixels where the bit is 0 are transparent, allowing sprites and content at lower priority levels to show through. The early bit can be set to 1 to shift the horizontal position of the sprite left by 32 pixels, allowing bleed-in from the left edge of the backdrop.

The defsprite() routine also creates an entry in the Sprite Generator Table using the supplied pattern data. If the small sprites are in effect 8 bytes are used; for large sprites 32 bytes are required. It is the user’s responsibility to ensure that the correct number of bytes are provided in pattern, corresponding to the sprite size setting in effect.

defsprite() may often be all that’s needed to define a set of sprites, however there are also routines to define or redefine just the sprite patterns (in the Sprite Generator Table), define or redefine just the Sprite Attribute Table entries, and to connect a sprite to one of the patterns in the Sprite Generator Table. These routines are described below:

crsprite(sprnumber, x, y, color, early);

This routine is similar to defspite() but only creates (or updates) the entry in the Sprite Attribute Table. The calling program must separately connect the sprite to the appropriate pattern using asgsprpattern() (described below).

crpattern(patnumber, pattern);

This routine is similar to defsprite() but only creates (or updates) the pattern entry in the Sprite Generator Table. The calling program must separately connect the sprite to the appropriate pattern using asgsprpattern() (described below). patnumber is simply the index of the pattern in the table (0..255 for small sprites, 0..63 for large ones).

asgsprpattern(patnumber, sprnumber);

This routine connects a pattern in the Sprite Generator Table to an entry in the Sprite Attribute Table, effecting an instantaneous change in the sprite display. patnumber and sprnumber are as defined above.

Here is a sample code segment from a dice game that creates 8x8 sprites representing the six possible rolls of a die. crpattern() is used to store the six patterns (specified in the spat[][] array) in the first six entries of the Sprite Generator Table. The program (only partially shown) then uses crsprite() to create 32 sprites with randomized locations, colors and die values, and then uses asgsprpattern() to connect each sprite to the appropriate sprite pattern for the face value of the die:

/\* patterns defining 8x8 dice sprites \*/

char spatt[6][8] = {

{0x00, 0x00, 0x00, 0x18, 0x18, 0x00, 0x00, 0x00},

{0x03, 0x03, 0x00, 0x00, 0x00, 0x00, 0xC0, 0xC0},

{0x03, 0x03, 0x00, 0x18, 0x18, 0x00, 0xC0, 0xC0},

{0xC3, 0xC3, 0x00, 0x00, 0x00, 0x00, 0xC3, 0xC3},

{0xC3, 0xC3, 0x00, 0x18, 0x18, 0x00, 0xC3, 0xC3},

{0xC3, 0xC3, 0x00, 0xC3, 0xC3, 0x00, 0xC3, 0xC3}

};

/\* randdie - used with the main screen sprites \*/

randdie(i)

int i;

{

sdice[i].x = rndrange(0,31)\*8;

sdice[i].y = rndrange(0,23)\*8;

sdice[i].dy = rndrange(2,4);

sdice[i].value = rndrange(0,5);

sdice[i].color = rndrange(c\_midgrn, c\_white);

}

…

/\* set up small sprites \*/

/\* create the six sprite patterns \*/

for (i=0; i<6; i++)

crpattern(i, &spatt[i][0]);

/\* create 32 sprites at random locations with random

\*\* sprite patterns

\*/

for (i=0; i<32; i++) {

/\* select random die characteristics and create

\*\* the associated sprite, then assign it the

\*\* corresponding sprite pattern...

\*/

randdie(i);

crsprite(i, sdice[i].x, sdice[i].y, sdice[i].spcolor, 0);

asgsprpattern(sdice[i].value, i);

}

There are three utility routines that can be used to manipulate the entries in the Sprite Attribute Table for a given sprite. They let you change the location, color and “early bit” status of any of the 32 sprites:

positsprite(sprnumber, x, y);

Changes the position of the sprite specified by sprnumber. The display is updated instantly showing the sprite in the new location. x indicates the horizontal (column) position in the range of 0..255. y indicates the vertical (row) position in the range of 0..191.

chgsprcolor(sprnumber, color);

Changes the foreground color of the specified sprite to color, where color is one of the 16 color constants defined in ha83.h. The display is updated instantly showing the sprite in the specified color. Any pixel that is defined as a 1 in the associated Sprite Generator Table pattern is displayed in the specified color. Any pixel that is defined as a 0 is transparent.

wrtearlybit(sprnumber, early);

Sets the “early bit” for the associated sprite number to the value specified in early (1=set, 0=cleared). This reduces the horizontal location by 32 pixels. One way to hide a sprite, for example, is to set the early bit to 1 and the horizontal position to 0. This will cause the sprite to be hidden off the left side of the screen.

## Graphics Modes and Routines