80x86 Assembly Language Libraries

© October-November 2002

Dr. William T. Verts

Introduction

This document describes the contents of two assembly language libraries created in October and November 2002 to support the honors section of CMPSCI 201, Architecture and Assembly Language, fall semester at the University of Massachusetts at Amherst. The libraries were created in Turbo Assembler using the `SMALL` model, and are provided along with this document in `.OBJ` format. The `.OBJ` files are free for general use, and may be distributed freely as long as they are kept together with this document.

The two libraries given here are called `STANDARD.OBJ` and `GRAPHICS.OBJ`. The libraries are completely independent; neither depends upon the other. The `STANDARD` library provides a series of procedures for interfacing assembly language programs with the BIOS of the PC, along with routines to provide general-purpose computational services. These routines include keyboard handlers, print routines for integers, random-number generators, etc. The `GRAPHICS` library contains routines that support the VGA/MCGA mode-13 graphics (320×200 pixels, 256 colors).

Any program that uses either or both of these libraries should be designed to use the `SMALL` assembler model (64K code, 64K data). All library procedures use the `NEAR` calling mechanism. A general template for new programs is shown below, which as an example correctly references the `Text80x25` routine from the `STANDARD` library:

```
.MODEL    SMALL
.STACK    100h
.DATA
…                         ; Data declarations here
.CODE
EXTRN     Text80x25:NEAR  ; External subroutine
MOV       AX,@DATA        ; StartUp Code
MOV       DS,AX           ;
…                         ; New code here
CALL      Text80x25       ; Call to external subroutine
…                         ; New code here
MOV       AH,4CH          ; Exit to DOS
INT       21H             ;
END
```

If this sample program is called MAIN.ASM, then the assembly and linking steps are performed as follows:

```
TASM MAIN
TLINK MAIN+STANDARD
```

The TASM step converts the statements from the MAIN.ASM file into a linkable binary file called MAIN.OBJ. The TLINK step is where the EXTRN references in MAIN.OBJ are resolved with the proper addresses from the STANDARD.OBJ file. Linking will create the executable file MAIN.EXE, which can be run by typing its name at an MS-DOS command line or by double-clicking its icon in Windows.

If the new program uses both the STANDARD.OBJ and GRAPHICS.OBJ files, the process is as follows:

```
TASM MAIN
TLINK MAIN+STANDARD+GRAPHICS
```

STANDARD.OBJ

Introduction

The STANDARD library contains a set of routines that are useful to nearly all programs. These routines include keyboard and printing routines, random-number routines, and screen and diagnostic routines. To include links to these routines in a new program, copy the following external references into the code section of the program (or copy the references to just the routines that are required):

```
EXTRN     Start_Stack:NEAR
EXTRN     Check_Stack:NEAR
EXTRN     Text80x25:NEAR
EXTRN     Delay:NEAR
EXTRN     Keyboard_Flush:NEAR
EXTRN     Keyboard_Wait:NEAR
EXTRN     Keypressed:NEAR
EXTRN     ReadKey:NEAR
EXTRN     PrintASCII:NEAR
EXTRN     PrintBEEP:NEAR
EXTRN     PrintBLANK:NEAR
EXTRN     PrintTAB:NEAR
EXTRN     PrintMINUS:NEAR
EXTRN     PrintCRLF:NEAR
EXTRN     PrintHEX:NEAR
EXTRN     PrintBYTE:NEAR
EXTRN     PrintNYBBLE:NEAR
EXTRN     PrintBIN:NEAR
EXTRN     PrintSigned:NEAR
EXTRN     PrintUnsigned:NEAR
EXTRN     PrintFraction:NEAR
EXTRN     PrintDump:NEAR
EXTRN     Random_Initialize:NEAR
EXTRN     Random_Randomize:NEAR
EXTRN     Random_Set_Seed:NEAR
EXTRN     Random_Get_New:NEAR
EXTRN     Upper_Case:NEAR
EXTRN     Lower_Case:NEAR
EXTRN     SQRT_Signed:NEAR
EXTRN     SQRT_Unsigned:NEAR
```

Each of these routines is described in the following sections.

General Routines

Start_Stack and Check_Stack

Purpose: These two routines are diagnostic routines to aid in debugging code that uses the stack. Calling `Start_Stack` simply stores the current value of the stack pointer into a local variable. Calling `Check_Stack` compares the current value of the stack pointer with the value stored in the local variable; if the values are equal then a message stating that there is nothing on the stack is printed, and if the values differ then another message stating the difference in the number of bytes is printed. It is probably a good idea to call `Start_Stack` at the beginning of a new program and `Check_Stack` at the end in order to verify overall stack usage; these calls should be commented out after the program has been debugged.

Calling Sequence:

```
CALL Start_Stack
…                       ; do something with the stack
CALL Check_Stack
```

Registers: Unlike all the other routines in this library, the `Check_Stack` routine does not preserve the registers. This is so because to preserve the registers on the stack would modify the stack value under test.

Text80x25

Purpose: This routine sets the display mode to text at 25 lines by 80 characters per line, and also clears the screen. Calling this routine also exits any graphics mode that may be running. It is a good idea to call this routine at the start and at the end of any program to establish a consistent environment for all the other code.

Calling Sequence:

```
CALL Text80x25
```

Registers: All registers preserved except the status register.

Delay

Purpose: This routine pauses for a few milliseconds; the number of milliseconds to wait is passed through `AX`.

Calling Sequence:

```
MOV  AX,milliseconds
CALL Delay
```

Registers: All registers preserved except the status register.

Example:

The following example framework shows the layout of a typical assembly language program using the SMALL programming model. It shows the expected positions of the calls to the first three routines shown in this section.

```
.MODEL    SMALL
.STACK    100h
.DATA

…                          ; Data declarations here

.CODE
EXTRN     Text80x25
EXTRN     Start_Stack
EXTRN     Check_Stack

MOV       AX,@DATA         ; StartUp Code
MOV       DS,AX            ;
CALL      Start_Stack      ;
CALL      Text80x25        ;

…                          ; Program statements here

CALL      Text80x25        ;
CALL      Check_Stack      ;
MOV       AH,4CH           ; Exit to DOS
INT       21H              ;
END
```

Keyboard Routines

These routines all use the MS-DOS keyboard buffer, which is an array of characters used to store characters coming in from the keyboard until they are required by an application program. Striking a key on the keyboard generates an interrupt, which temporarily suspends your program while the new value is inserted into the buffer.

Keyboard_Flush

Purpose: This routine clears the MS-DOS keyboard buffer. As keys are entered at the keyboard, the values are entered automatically into the keyboard buffer. These key values stay in the buffer until they are read (by `ReadKey`, below). If the program is designed to ask a question, then wait for an answer, it is necessary to flush the keyboard buffer first in order to avoid incorrect answers.

Calling Sequence:

```
CALL Keyboard_Flush
```

Registers: All registers preserved except the status register.

Keyboard_Wait

Purpose: This routine waits until a key is pressed at the keyboard. If a key is already waiting in the keyboard buffer, then this routine will return immediately.

Calling Sequence:

```
CALL Keyboard_Wait
```

Registers: All registers preserved except the status register.

Keypressed

Purpose: This routine returns TRUE (Carry bit = 1) or FALSE (Carry bit = 0) depending on whether a key value is waiting in the keyboard buffer. The `Keyboard_Wait` routine (described above) is essentially equivalent to "While Not Keypressed Do nothing".

Calling Sequence:

```
CALL Keypressed
JC   …              ; Jump if key is pressed
…                   ; Handle no key being pressed
```

Registers: All registers preserved except the status register. `C`=1 if a key is pressed, `C`=0 if no key is pressed.

ReadKey

Purpose: This routine waits for a key-press if necessary, and then returns the value of that key in the AX register. It will wait forever if no keys are ever pressed. If the `ReadKey` routine is called when one or more characters are in the keyboard buffer, it will remove and return the first of those key values.

Calling Sequence:

```
CALL ReadKey
```

Registers: All registers preserved except AX and the status register. AX contains the key value read from the keyboard.

Example #1:

In this example, the computer waits until a Y or an N is entered from the keyboard (as in the answer to a yes-or-no question).

```
                CALL Keyboard_Flush ; Eliminate any waiting keys
Char_Loop:      CALL ReadKey
                CMP  AL,'Y'
                JZ   Do_Yes
                CMP  AL,'N'
                JZ   Do_No
                JMP  Char_Loop

Do_Yes:         …

Do_No:          …
```

Example #2:

In this example framework, an event loop is being implemented where something has to happen continuously (like updating a clock or the positions of players in a game), even if no keys are hit. When a key is hit the appropriate handler is called, and afterwards the event loop continues running.

```
                CALL Keyboard_Flush

Event_Loop:     CALL Update_Process ; Do what must be done a lot
                CALL Keypressed     ; Check the keyboard and…
                JNC  Event_Loop     ; …go back if nothing to do

                CALL ReadKey        ; Get the new key value

     Check_A:   CMP  AL,'A'
                JNZ  Check_B
                CALL Do_A
                JMP  Event_Loop

     Check_B:   CMP  AL,'B'
                JNZ  Check_C
                CALL Do_B
                JMP  Event_Loop

     Check_C:   CMP  AL,'C'
                JNZ  Check_D
                CALL Do_C
                JMP  Event_Loop

     Check_D:

                .
                .
                .

     Check_Z:   CMP  AL,'Z'
                JNZ  No_Key
                CALL Do_Z
                JMP  Event_Loop

     No_Key:    CALL Do_Error       ; Not a valid key
                JMP  Event_Loop
```

Printing Routines

All of these routines cause a change to appear on the text mode screen display (with the exception of `PrintBEEP` which "prints" a bell ringing code). These routines should not be called if the video system is running in a graphics mode, such as the mode 13 graphics used in `GRAPHICS.OBJ`, but only in text mode.

PrintASCII

Purpose: This routine prints out to the text screen the character whose ASCII value is passed in through the `AL` (or `AX`) register.

Calling Sequence:

```
MOV  AL,xxx          ; ASCII character value
CALL PrintASCII

-or-

MOV  AX,xxx          ; AH ignored
CALL PrintASCII
```

Registers: All registers preserved except the status register.

Example:

```
MOV  AX,'H'
CALL PrintASCII
MOV  AX,'e'
CALL PrintASCII
MOV  AX,'l'
CALL PrintASCII
MOV  AX,'l'
CALL PrintASCII
MOV  AX,'o'
CALL PrintASCII
```

PrintBEEP

Purpose: Generates a short tone through the speaker. Works best on pure MS-DOS systems (generates a short click when run under Windows systems). This call is equivalent to `MOV AX,7` followed by `CALL PrintASCII` (character code 7 is the "BELL" code).

Calling Sequence:

```
CALL PrintBEEP
```

Registers: All registers preserved except the status register.

PrintBLANK

Purpose: Prints a blank on the text screen. This call is equivalent to `MOV AX,' '` followed by `CALL PrintASCII`.

Calling Sequence:
```
CALL PrintBLANK
```

Registers: All registers preserved except the status register.

PrintTAB

Purpose: Prints a tab character on the text screen, jumping to the next tab stop. Tab stops are at character positions 1, 9, 17, 25, etc. (every eight characters) This call is equivalent to `MOV AX,9` followed by `CALL PrintASCII`.

Calling Sequence:
```
CALL PrintTAB
```

Registers: All registers preserved except the status register.

PrintMINUS

Purpose: Prints a minus sign on the text screen. This call is equivalent to `MOV AX,'-'` followed by `CALL PrintASCII`.

Calling Sequence:
```
CALL PrintMINUS
```

Registers: All registers preserved except the status register.

PrintCRLF

Purpose: Prints a carriage return and a line feed on the text screen (moves the cursor to the beginning of the next line). This call is equivalent to `MOV AX,13` followed by `CALL PrintASCII`, and then `MOV AX,10` followed by `CALL PrintASCII`, (character code 13 is the carriage-return, and character code 10 is the line-feed).

Calling Sequence:
```
CALL PrintCRLF
```

Registers: All registers preserved except the status register.

PrintHEX

Purpose: Prints the contents of the AX register on the screen as four hexadecimal characters. (This routine calls PrintBYTE twice.)

Calling Sequence:

```
MOV  AX,xxx          ; value to print
CALL PrintHex
```

Registers: All registers preserved except the status register.

PrintBYTE

Purpose: Prints the contents of the AL register on the screen as two hexadecimal characters. (This routine calls PrintNYBBLE twice.)

Calling Sequence:

```
MOV  AL,xxx          ; value to print
CALL PrintBYTE
```

Registers: All registers preserved except the status register.

PrintNYBBLE

Purpose: Prints the lower four bits of the contents of the AL/AX register on the screen as a single hexadecimal character.

Calling Sequence:

```
MOV  AL,xxx          ; value to print
CALL PrintNYBBLE
```

Registers: All registers preserved except the status register.

PrintBIN

Purpose: Prints the contents of the AX register on the screen as a string of 16 bits ('0' or '1' characters).

Calling Sequence:

```
MOV  AX,xxx          ; value to print
CALL PrintBIN
```

Registers: All registers preserved except the status register.

PrintSigned

Purpose: Prints the contents of the AX register on the screen in decimal, treating the value as a 16-bit signed two's-complement integer. The result printed will be a value between –32768 and 32767, and only the minimum necessary number of characters will be printed.

Calling Sequence:

```
MOV  AX,xxx          ; value to print
CALL PrintSigned
```

Registers: All registers preserved except the status register.

PrintUnsigned

Purpose: Prints the contents of the AX register on the screen in decimal, treating the value as a 16-bit unsigned integer. The result printed will be a value between 0 and 65535, and only the minimum necessary number of characters will be printed.

Calling Sequence:

```
MOV  AX,xxx          ; value to print
CALL PrintUnsigned
```

Registers: All registers preserved except the status register.

PrintFraction

Purpose: Prints the contents of the AX register on the screen in decimal, treating the value as a 16-bit unsigned fraction, where the decimal point is assumed to be at the left side of the register. The decimal point will be the first character printed. The decimal result will be a value less than 1, and the entire fraction will be printed. With 16 bits, the smallest non-zero value that can be represented is .0000152587890625 (equivalent to hexadecimal value $0001), and the largest is .9999847412109375 (equivalent to hexadecimal $FFFF), and all of those digits will be printed.

Calling Sequence:

```
MOV  AX,xxx          ; value to print
CALL PrintFraction
```

Registers: All registers preserved except the status register.

PrintDump

Purpose: Prints the contents of all registers on the screen, plus the top four entries on the stack, for debugging purposes. No registers are changed as a result of this call. The top of the stack is dumped as if the call to `PrintDump` was invisible; it prints out the status of the `SP` register and the top four entries of the stack as they are just before (and just after) the call. Each register is printed in hexadecimal (between `0000` and `FFFF`), as an unsigned integer (between 0 and 65535), and as a signed integer (between –32768 and +32767).

Calling Sequence:

```
CALL PrintDump
```

Registers: All registers preserved except the status register.

Sample Screen Output:

```
REG     HEX     UnSign  Signed
----    ----    ------  ------
AX      0C59    3161    3161
BX      0000    0       0
CX      00FF    255     255
DX      0C13    3091    3091
SI      0000    0       0
DI      FF00    65280   -256
BP      0912    2322    2322
SP      FF00    65280   -256

SP-->   0001    1       1
        0000    0       0
        E738    59192   -6344
        DB07    56071   -9465
```

Random Number Routines

These routines use a 16-bit shift register as a random number generator. Every new random number is generated from the previous number by shifting it to the right, shifting in to the most significant bit the exclusive-OR of bits 0, 1, 7, and 12. This process will generate all possible 16-bit patterns except 0 (if zero were ever allowed in the shift register it would remain at zero thereafter). Note that this is a simple technique that is easily implemented, but the sequence does eventually repeat. This routine is OK for casual usage, but should ***not*** be used for serious work requiring extremely long sequences of random numbers.

Random_Initialize

Purpose: This routine initializes the random-number generator to a fixed starting seed. Calling this routine is equivalent to the instruction `MOV AX,1` followed by `CALL Random_Set_Seed`.

Calling Sequence:

```
CALL Random_Initialize
```

Registers: All registers preserved.

Random_Randomize

Purpose: This routine initializes the random-number generator to a seed generated by the computer's real-clock. The value is a 16-bit number formed by using the current time as follows: seed := hour×256 + minute + seconds×256 + hundredths. (Exactly midnight gives a seed of zero, which is illegal and is changed by `Random_Set_Seed` to 1.)

Calling Sequence:

```
CALL Random_Randomize
```

Registers: All registers preserved except the status register.

Random_Set_Seed

Purpose: This routine allows the programmer to initialize the random-number generator to a known starting seed. The new seed value is passed to the routine through the `AX` register, and may be any 16-bit value except zero (which is automatically changed to 1).

Calling Sequence:

```
MOV  AX,xxx
CALL Random_Set_Seed
```

Registers: All registers preserved except the status register.

Random_Get_New

Purpose: This routine updates the random number seed to a new value, and returns that new value through the AX register. The value returned may be any 16-bit number except zero. This value may be scaled to any range by treating it as an integer and using the remainder of dividing it by another integer, or by treating the value as a fraction and multiplying it by an integer, then truncating the result.

Calling Sequence:

```
CALL Random_Get_New
{ use value in AX }
```

Registers: All registers preserved except AX and the status register. Register AX contains the new random number, as an integer between 1 and 65535 (inclusive).

Example:

Here is some sample code that returns a random integer between 1 and 10. Note that the scaling part of the code gets a random number between 0 and 9, to which 1 must be added to shift the range upwards to between 1 and 10. The first version extracts the remainder of dividing the random integer by 10, and the second multiplies the random fraction by 10 and truncating the result.

```
CALL Random_Get_New
XOR  DX,DX
MOV  BX,10
DIV  BX         ; AX := DX:AX÷10, DX=remainder 0..9
ADD  DX,1
{ random number is in DX }


CALL Random_Get_New
MOV  BX,10
MUL  BX         ; DX:AX := 0.AX×10, DX=whole part 0..9
ADD  DX,1
{ random number is in DX }
```

To generate a number in any range of integers, replace the 1 in the code above with the smallest expected value, and replace the 10 with the total number of random values in the sequence (not the largest expected value, but instead the largest value minus the smallest value plus one).

Character Routines

These routines perform some simple processing on ASCII characters.

Upper_Case

Purpose: This routine capitalizes the ASCII character in the AL register. If the character is not in the range 'a'.. 'z' then no change is made.

Calling Sequence:

```
CALL Upper_Case
```

Registers: All registers preserved except AL.

Lower_Case

Purpose: This routine changes the ASCII character in the AL register into lower case. If the character is not in the range 'A'.. 'Z' then no change is made.

Calling Sequence:

```
CALL Lower_Case
```

Registers: All registers preserved except AL.

Square-Root Routines

These routines extract the integer square root of the number in AX and return the value back in the AX register. The number returned is the floor of what would be the true, real square root of the argument; the largest integer less than or equal to the true answer. Thus, the square root of 16 is 4, but so are the square roots of all numbers between 16 and 24. The square root of all numbers between 25 and 35 is 5, the square root of all numbers between 36 and 48 is 6, etc.

SQRT_Signed and SQRT_Unsigned

Purpose: The SQRT_Unsigned routine expects an unsigned integer argument between 0 and 65535 in AX, and returns the square root in AX (between 0 and 255). The SQRT_Signed routine expects a signed integer in AX between –32768 and +32767. It returns the proper square root in AX (between 0 and 181) for argument values greater than or equal to zero and clears the Carry bit, but for illegal values less than zero the value returned in AX is zero and the Carry bit is set.

Calling Sequence:

```
MOV  AX,xxx                MOV  AX,xxx
CALL SQRT_Signed           CALL SQRT_Unsigned
JCS  Error
```

Registers: All registers preserved except AX.

GRAPHICS.OBJ

Introduction

The `GRAPHICS` library contains a set of routines which exploit mode 13 graphics on the VGA/MCGA video adapter. Mode 13 graphics is low resolution (only 320×200), but supports up to 256 simultaneous colors, and is a particularly easy graphics mode to access from assembly language. These routines include getting into and out of graphics mode, drawing lines, pixels, boxes (filled and outline), circles (filled and outline), and text, and scrolling the screen up or down. To include links to these routines in a new program, copy the following external references into the code section of the program (or copy the references to just the routines that are required):

```
EXTRN     Graphics_Mode:NEAR
EXTRN     Text_Mode:NEAR
EXTRN     Set_Color:NEAR
EXTRN     Clear_Screen:NEAR
EXTRN     Set_Pixel:NEAR
EXTRN     HLine:NEAR
EXTRN     VLine:NEAR
EXTRN     Line:NEAR
EXTRN     Block:NEAR
EXTRN     Box:NEAR
EXTRN     Circle:NEAR
EXTRN     Spot:NEAR
EXTRN     Scroll_Up:NEAR
EXTRN     Scroll_Down:NEAR
EXTRN     OutChar:NEAR
EXTRN     OutString:NEAR
EXTRN     Justify_Left:NEAR
EXTRN     Justify_Center:NEAR
EXTRN     Justify_Right:NEAR
EXTRN     Plot_Unsigned:NEAR
EXTRN     Plot_Signed:NEAR
EXTRN     Save_BMP:NEAR
EXTRN     Save_Default:NEAR
EXTRN     Palette_Save:NEAR
EXTRN     Palette_Restore:NEAR
EXTRN     VGA_Set_Segment:NEAR
EXTRN     VGA_Get_Segment:NEAR
EXTRN     VGA_Reset_Segment:NEAR
EXTRN     VGA_Copy_To_Screen:NEAR
EXTRN     VGA_Copy_To_Buffer:NEAR
```

Screen Routines

These routines are used to get into and out of graphics mode. Note that the Text_Mode routine is functionally identical to the `Text80x25` routine in the `STANDARD.OBJ` library, but because it is included here, the `GRAPHICS.OBJ` library is independent of the `STANDARD.OBJ` library, and the two libraries need not be loaded together unless functions from both are truly required.

Graphics_Mode

Purpose: This routine switches the video mode into mode 13 graphics mode. The screen is cleared by the action of switching modes, but this routine does not explicitly set the screen to any particular color (to do that, call the `Clear_Screen` routine instead). Before exiting any program, get out of graphics mode by calling the `Text_Mode` routine. This routine is provided as a convenience, as the actual code is very simple (`MOV AX,0013H` followed by `INT 10H`).

Calling Sequence:

```
CALL Graphics_Mode
```

Registers: All registers preserved except the status register.

Text_Mode

Purpose: This routine switches the video mode into text mode with 25 lines by 80 characters per line, and should always be called immediately before exiting to MS-DOS in any routine using graphics. It is provided as a convenience, as the actual code is very simple (`MOV AX,0003H` followed by `INT 10H`). This routine is identical to `Text80x25` from the `STANDARD.OBJ` library.

Calling Sequence:

```
CALL Text_Mode
```

Registers: All registers preserved except the status register.

Pixel Routines

These routines are the simplest ways to put color on the screen. Mode 13 graphics supports 256 colors on screen in a palette. By default, upon program start the first 16 entries in the palette are predefined to contain the standard 16 VGA colors. Those colors are as follows:

```
0    Black          8    Dark Gray
1    Blue           9    Light Blue
2    Green          10   Light Green
3    Cyan           11   Light Cyan
4    Red            12   Light Red
5    Magenta        13   Light Magenta
6    Brown          14   Yellow
7    Light Gray     15   White
```

The screen is a two-dimensional array of pixels, where the coordinates of the upper-left pixel are <0,0> and the coordinates of the lower-right pixel are <319,199>. There are 64000 separate pixels in the screen.

Set_Color

Purpose: This routine defines to the graphics system the color to be used in *all* drawing routines. The color value is passed to the routine through the `AL` register (or through `AX`, where only the lower 8 bits are used).

Calling Sequence:

```
MOV  AX,xxx
CALL Set_Color
```

Registers: All registers preserved except the status register.

Clear_Screen

Purpose: This routine nearly instantly sets every pixel in the screen to the last color defined by `Set_Color`.

Calling Sequence:

```
MOV  AX,xxx
CALL Set_Color
CALL Clear_Screen
```

Registers: All registers preserved except the status register.

Set_Pixel

Purpose: This routine puts one pixel on the screen. Coordinates are passed in through the stack, and the pixel is set to the last color defined by `Set_Color`. Full clipping is supported, so that nothing is painted on the screen if the X value is outside the range of 0 through 319 or if the Y value is outside the range of 0 through 199.

Calling Sequence:

```
{Push X}
{Push Y}
CALL Set_Pixel
```

Registers: All registers preserved except the status register.

Example:

```
MOV  AX,319
PUSH AX
MOV  AX,199
PUSH AX
CALL Set_Pixel      ; Set_Pixel(319,199)
```

Line Routines

These routines all draw straight lines on the screen using the last color defined by `Set_Color`. Drawing horizontal lines is slightly faster than drawing vertical lines, and both are far faster than drawing a general line between any two arbitrary points.

HLine

Purpose: This routine draws a horizontal line between coordinates <X1,Y> and <X2,Y> using the last color defined by `Set_Color`. The values of X1 and X2 can appear in either order. This is an extremely fast routine; once the coordinates have been clipped to the screen, the remaining visible piece of line (if any) is painted by a single x86 instruction.

Calling Sequence:

```
{Push X1}
{Push X2}
{Push Y}
CALL HLine
```

Registers: All registers preserved except the status register.

Example:

```
MOV  AX,0      ; X1
PUSH AX
MOV  AX,319    ; X2
PUSH AX
MOV  AX,199    ; Y
PUSH AX
CALL HLine     ; HLine (0,319,199)  { bottom raster }
```

VLine

Purpose: This routine draws a vertical line between coordinates <X,Y1> and <X,Y2> using the last color defined by `Set_Color`. The values of Y1 and Y2 can appear in either order. Once the coordinates have been clipped to the screen, the remaining visible piece of line (if any) is painted by a small tight loop, but this routine is not quite as fast as `HLine`.

Calling Sequence:

```
{Push X}
{Push Y1}
{Push Y2}
CALL VLine
```

Registers: All registers preserved except the status register.

Example:

```
MOV  AX,319     ; X
PUSH AX
MOV  AX,0       ; Y1
PUSH AX
MOV  AX,199     ; Y2
PUSH AX
CALL VLine      ; VLine (319,0,199)  { right side }
```

Line

Purpose: This routine draws a line between two arbitrary coordinates <X1,Y1> and <X2,Y2>, using the last color defined by `Set_Color`. The algorithm uses Bresenham's digital line stepping algorithm to generate the coordinates of individual pixels along the line. Those coordinates are passed to `Set_Pixel`, which performs the necessary clipping to the screen to eliminate off-screen pixels. If Y1 equals Y2, this routine automatically calls `HLine` to speed up line drawing. Similarly, if X1 equals X2, the `VLine` routine is automatically called.

Calling Sequence:

```
{Push X1}
{Push Y1}
{Push X2}
{Push Y2}
CALL VLine
```

Registers: All registers preserved except the status register.

Example:

```
MOV  AX,0       ; X1
PUSH AX
MOV  AX,0       ; Y1
PUSH AX
MOV  AX,199     ; X2
PUSH AX
MOV  AX,319     ; Y2
PUSH AX
CALL Line       ; Line (0,0,319,199)  { corner-corner }
```

Drawing Routines

These routines all draw two-dimensional figures on the screen using the last color defined by `Set_Color`. `Block` and `Box` paint rectangular figures on the screen, the first a solid rectangle and the second just the outline of a box. `Spot` and `Circle` paint circular figures on the screen, the first a solid spot and the second just the circle outline.

Block

Purpose: This routine draws a rectangular block defined by corner coordinates <X1,Y1> and <X2,Y2> using the last color defined by `Set_Color`. The values X1 and X2 can appear in either order, as can the values Y1 and Y2 (i.e., it does not matter in which order the opposing corner coordinates are specified). The coordinates are clipped to the screen, and the remaining lines of the block (if any) are painted on screen using the same technique as in `HLine` for maximum drawing speed.

Calling Sequence:

```
{Push X1}
{Push Y1}
{Push X2}
{Push Y2}
CALL Block
```

Registers: All registers preserved except the status register.

Example:

```
MOV  AX,10     ; X1
PUSH AX
MOV  AX,10     ; Y1
PUSH AX
MOV  AX,50     ; X2
PUSH AX
MOV  AX,30     ; Y2
PUSH AX
CALL Block     ; Block (10,10,50,30)
```

Box

Purpose: This routine is identical to the `Block` routine except that it draws the outline of the rectangle defined by corner coordinates <X1,Y1> and <X2,Y2>, instead of its interior, using the last color defined by `Set_Color`. The values X1 and X2 can appear in either order, as can the values Y1 and Y2. The body of this routine calls `HLine` twice and `VLine` twice with the appropriate sets of coordinates.

Calling Sequence:

```
{Push X1}
{Push Y1}
{Push X2}
{Push Y2}
CALL Box
```

Registers: All registers preserved except the status register.

Example:

```
MOV  AX,10      ; X1
PUSH AX
MOV  AX,10      ; Y1
PUSH AX
MOV  AX,50      ; X2
PUSH AX
MOV  AX,30      ; Y2
PUSH AX
CALL Box        ; Box (10,10,50,30)
```

Spot

Purpose: This routine paints a solid circular spot on screen, at center coordinate <X,Y> and with radius R, using the last color defined by `Set_Color`. The routine clips the bounding box of the circle against the screen coordinates to see if any part needs to be drawn at all. The body of this routine calls the `HLine` routine for each raster line in the circle, which performs any needed clipping for partially visible circles. Spots of radius 0 still appear as a single pixel at the center coordinate. Negative radii are OK; the absolute value is used.

Calling Sequence:

```
{Push X}
{Push Y}
{Push R}
CALL Spot
```

Registers: All registers preserved except the status register.

Example:

```
MOV  AX,160    ; X
PUSH AX
MOV  AX,100    ; Y
PUSH AX
MOV  AX,50     ; R
PUSH AX
CALL Spot      ; Spot (160,100,50)  { screen center }
```

Circle

Purpose: This routine is essentially the same as the Spot routine except that it draws the outline ring of a circle at center <X,Y> and radius R, instead of its interior, using the last color defined by `Set_Color`. The routine clips the bounding box of the circle against the screen coordinates to see if any part needs to be drawn at all. The body of this routine calls the `Set_Pixel` routine for each pixel in the circle, which performs any needed clipping for partially visible circles. Circles of radius 0 still appear as a single pixel at the center coordinate. Negative radii are OK; the absolute value is used.

Calling Sequence:

```
{Push X}
{Push Y}
{Push R}
CALL Circle
```

Registers: All registers preserved except the status register.

Example:

```
MOV  AX,160    ; X
PUSH AX
MOV  AX,100    ; Y
PUSH AX
MOV  AX,50     ; R
PUSH AX
CALL Circle    ; Circle (160,100,50) { screen center }
```

Scrolling Routines

These routines scroll the entire screen up or down by one raster line. The raster moving off the edge of the screen is lost, and the raster freed up by the scroll is painted with the last color defined by `Set_Color`. Bear in mind that these routines are moving 64000 bytes (pixels) at once; even though the routines use the fastest instructions available in the x86 these routines are quite slow, even on fast hardware.

Scroll_Up

Purpose: This routine scrolls the entire screen up by one raster. The raster at the top of the screen is lost, and the raster at the bottom of the screen is painted with the last color defined by `Set_Color`.

Calling Sequence:

```
CALL Scroll_Up
```

Registers: All registers preserved except the status register.

Scroll_Down

Purpose: This routine scrolls the entire screen down by one raster. The raster at the bottom of the screen is lost, and the raster at the top of the screen is painted with the last color defined by `Set_Color`.

Calling Sequence:

```
CALL Scroll_Down
```

Registers: All registers preserved except the status register.

Character Routines

These routines plot, or help plot, characters on the screen using the last color defined by `Set_Color`. The characters are defined by an 8×8 grid of pixels, and have the same definitions as in the original IBM-PC graphics character set. The 256 complete character definitions are included as part of the `GRAPHICS.OBJ` file, which adds only 2K to the overall size of the executable file.

OutChar

Purpose: This routine plots a single character on the screen, where the upper-left coordinate of the character grid is at location <X,Y>. The shape of the character will be painted using the last color defined by `Set_Color`. Character values follow the IBM-PC extended ASCII set; any value between 0 and 255 is legal. Full clipping insures all parts of the character that *should* appear on screen *will* be painted, and as fast as possible regardless of how much of the character is visible.

Calling Sequence:

```
{Push X}
{Push Y}
{Push Char}
CALL Circle
```

Registers: All registers preserved except the status register.

Example:

```
MOV  AX,160    ; X
PUSH AX
MOV  AX,100    ; Y
PUSH AX
MOV  AX,'Q'    ; R
PUSH AX
CALL OutChar   ; Print 'Q' at <160,100>
```

Justify_Left, Justify_Center, and Justify_Right

Purpose: These routines affect how the `OutString` routine (described next) plots strings on screen. In left justification (the default) the X coordinate of the string position is at the left end of the plotted string, in right justification the X coordinate is at the right end of the plotted string, and in center justification the X coordinate is in the middle of the plotted string. Justification does not use or modify the Y coordinate.

Calling Sequence:

```
CALL Justify_Left   or
CALL Justify_Center or
CALL Justify_Right
```

Registers: All registers preserved.

OutString

Purpose: This routine plots a zero-terminated string one character at a time, starting at location <X,Y> (according to the selected justification), using the last color defined by `Set_Color`. Each character is passed to `OutChar`, until the end of the string is reached or the X coordinate of the next character to plot is off the right edge of the screen. The string address must be an offset into the `DATA` segment.

Calling Sequence:

```
CALL Justify_Left   ; Optional justification
{Push X}
{Push Y}
{Push OFFSET String}
CALL OutString
```

Registers: All registers preserved except the status register.

Example:

```
  String1 DB   'Hello',0
          …
          MOV  AX,160              ; X
          PUSH AX
          MOV  AX,100              ; Y
          PUSH AX
          MOV  AX,OFFSET String1   ; R
          PUSH AX
          CALL OutString           ; 'Hello' at <160,100>
```

Plot_Unsigned and Plot_Signed

Purpose: `Plot_Unsigned` converts an unsigned 16-bit integer into decimal and plots the resulting string on the display at <X,Y>, using the last color defined by `Set_Color`. Legal 16-bit unsigned values are between 0 and 65535, and so the resulting string will be between one and five characters in length. Plot_Signed does exactly the same task, but treats the 16-bit value as a two's-complement signed integer. Legal 16-bit signed values are between –32768 and 32767, and so the resulting string will be between one and six characters in length. Characters are passed to `OutChar` for plotting, which performs any necessary clipping.

NOTE: At this time the `Plot_Unsigned` and `Plot_Signed` routines *do not* use the `OutString` routines, so they are not affected by center or right justification. Consider them to be hard-coded as always left-justified. (This may change in a future release.)

Calling Sequences:

```
{Push X}                    {Push X}
{Push Y}                    {Push Y}
{Push N}                    {Push N}
CALL Plot_Unsigned          CALL Plot_Signed
```

Registers: All registers preserved except the status register.

Example:

```
MOV  AX,160                 ; X
PUSH AX
MOV  AX,100                 ; Y
PUSH AX
MOV  AX,65535               ; R
PUSH AX
CALL Plot_Unsigned          ; Plot '65535' at <160,100>
```

File Routines

These routines save the image on screen to a Windows-compatible bitmap (`.BMP`) file. The files so created are always exactly 65,078 bytes in length. At the current time the palette that is saved with the file has the first 16 entries defined to be the standard VGA colors, and the remaining 240 palette entries are zero. This may change in the near future.

Save_BMP

Purpose: `Save_BMP` writes the entire 64,000 pixel graphic image on screen to a `.BMP` file. The offset to the filename is passed into the routine via the `DX` register, and the string must be zero-terminated. The `AX` register returns any error code from the file service routines called by this subroutine.

Calling Sequences:

```
MOV  DX, OFFSET Filename
CALL Save_BMP
CMP  AX,0
JNZ  Handle_Error
```

Registers: All registers preserved except the status register and `AX`. Any error in creating, writing to, or closing the file will return an error code in the `AX` register. The `AX` register will return zero if no error has occurred.

Example:

```
Filename  DB   'JUNK.BMP',0
          …
          MOV  DX, OFFSET Filename
          CALL Save_BMP
```

Save_Default

Purpose: `Save_Default` calls `Save_BMP` with the fixed name `DEFAULT.BMP` just to have a simple way of saving the graphics screen without worrying about setting up the filename buffer correctly.

Calling Sequences:

```
CALL Save_Default
```

Registers: All registers preserved except the status register and `AX`. Any error in creating, writing to, or closing the file will return an error code in the `AX` register. The `AX` register will return zero if no error has occurred.

Palette Routines

Palette_Save and Palette_Restore

Purpose: These routines save into and restore from a hidden memory area the default VGA palette of 256 colors. The intent is that the palette be saved just after the start of graphics mode processing and restored just before exiting to text mode; in between the palette can be modified at will.

NOTE: At the current time routines to modify individual palette entries are not provided. These may be added in a future release.

Calling Sequences:

```
CALL Palette_Save
…
CALL Palette_Restore
```

Registers: All registers preserved except the status register.

Example:

```
CALL Graphics_Mode
CALL Palette_Save
…
… do something that modifies the palette
…
CALL Palette_Restore
CALL Text_Mode
```

Video Buffer Routines

These routines all deal with the segment of the video screen, to be used by all plotting routines. By default, the video segment is A000 (the start of the VGA video buffer, visible on screen). These routines allow programmers to establish separate video buffers in memory, create images there, and then copy those updated images into the visible video buffer.

VGA_Set_Segment

Purpose: Sets the video segment to the value in the AX register. This segment number must point to the start of a 64,000-byte buffer somewhere in memory. It is possible with this call to set the video segment to a memory buffer, and then create and save graphics images off-screen without ever switching the video card into graphics mode (i.e., have a text-mode-only program create and save graphics files which are never shown on screen).

Calling Sequences:

```
MOV  AX,xxx
CALL VGA_Set_Segment
```

Registers: All registers preserved.

VGA_Get_Segment

Purpose: Gets the current video segment number into the AX register.

Calling Sequences:

```
CALL VGA_Get_Segment
```

Registers: All registers preserved except AX.

VGA_Reset_Segment

Purpose: Sets the video segment to the default value of A000, the start of the default VGA screen buffer. This call is functionally equivalent to MOV AX,0A000H followed by CALL VGA_Set_Segment. Note that there is no need to call VGA_Reset_Segment at the start of a program because the video segment value has already been properly initialized to A000.

Calling Sequences:

```
CALL VGA_Reset_Segment
```

Registers: All registers preserved.

VGA_Copy_To_Screen

Purpose: Copies the 64,000 bytes of an off-screen VGA video buffer into the visible VGA screen. `VGA_Set_Segment` must have been used previously to set the current video segment to point at a 64,000-byte buffer somewhere in memory. If the current segment happens to be the default value of `A000` (the visible screen), then no copy occurs. This function is used to create off-screen graphics, then quickly dump the newly created image into the visible VGA buffer.

Calling Sequences:

```
CALL VGA_Copy_To_Screen
```

Registers: All registers preserved except the status register.

VGA_Copy_To_Buffer

Purpose: Copies the 64,000 bytes of the visible VGA screen into an off-screen VGA video buffer. `VGA_Set_Segment` must have been used previously to set the current video segment to point at a 64,000-byte buffer somewhere in memory. If the current segment happens to be the default value of `A000` (the visible screen), then no copy occurs.

Calling Sequences:

```
CALL VGA_Copy_To_Buffer
```

Registers: All registers preserved except the status register.

Example:

In this example, the off-screen video buffer is stored in the `FARDATA` segment, and all new graphics are created there. Final images are then copied to the VGA screen.

```
            .DATA

            .FARDATA
Buffer      DB   64000 DUP(0)   ; Buffer starts at offset 0

            .CODE
            MOV  AX,@DATA
            MOV  DS,AX
            MOV  AX,@FARDATA
            CALL VGA_Set_Segment
            CALL Graphics_Mode
            ...                 ; Do graphics in off-screen buffer
            CALL VGA_Copy_To_Screen
            ...
```