The UCR Standard Library Chapter Seven

Most programming languages provide several "built-in" functions to reduce the effort needed to write a program. Traditionally, assembly language programmers have not had access to a standard set of commonly used subroutines for their programs; hence, assembly language programmers' productivity has been quite low because they are constantly "reinventing the wheel" in every program they write. The UCR Standard Library for 80x86 programmers provides such a set of routines. This chapter discusses a small subset of the routines available in the library. After reading this chapter, you should peruse the documentation accompanying the standard library routines.

7.0 Chapter Overview

This chapter provides a basic introduction to the functions available in the UCR Standard Librfary for 80x86 assembly language programmers. This brief introduction covers the following subjects:

- The UCR Standard Library for 80x86 Assembly Language Programmers.
- Memory management routines.
- Input routines.
- Output routines.
- Conversions.
- Predefined constants and macros.

7.1 An Introduction to the UCR Standard Library

The "UCR Standard Library for 80x86 Assembly Language Programmers" is a set of assembly language subroutines patterned after the "C" standard library. Among other things, the standard library includes procedures to handle input, output, conversions, various comparisons and checks, string handling, memory management, character set operators, floating point operations, list handling, serial port I/O, concurrency and coroutines, and pattern matching.

This chapter will not attempt to describe every routine in the library. First of all, the Library is constantly changing so such a description would quickly become outdated. Second, some of the library routines are for advanced programmers only and are beyond the scope of this text. Finally, there are hundreds of routines in the library. Attempting to describe them all here would be a major distraction from the real job at hand– learning assembly language.

Therefore, this chapter will cover the few necessary routines that will get you up and running with the least amount of effort. Note that the full documentation for the library, as well as the source code and several example files are on the companion diskette for this text. A reference guide appears in the appendices of this text. You can also find the latest version of the UCR Standard Library on many on-line services, BBSes, and from many shareware software houses. It is also available via anonymous FTP on the internet.

When using the UCR Standard Library you should always use the SHELL.ASM file provided as the "skeleton" of a new program. This file sets up the necessary segments, provides the proper include directives, and initializes necessary Library routines for you. You should not attempt to create a new program from scratch unless you are very familiar with the internal operation of the Standard Library.

Note that most of the Standard Library routines use macros rather than the call instruction for invocation. You cannot, for example, directly call the putc routine. Instead,

you invoke the putc macro that includes a call to the sl_putc procedure ("SL" stands for "Standard Library").

If you choose not to use the SHELL.ASM file, your program must include several statements to activate the standard library and satisfy certain requirements for the standard library. Please see the documentation accompanying the standard library if you choose to go this route. Until you gain some more experience with assembly language programming, you should always use the SHELL.ASM file as the starting point for your programs.

7.1.1 Memory Management Routines: MEMINIT, MALLOC, and FREE

The Standard Library provides several routines that manage free memory in the *heap*. They give assembly language programmers the ability to dynamically allocate memory during program execution and return this memory to the system when the program no longer needs it. By dynamically allocating and freeing blocks of memory, you can make efficient use of memory on a PC.

The meminit routine initializes the memory manager and you must call it before any routine that uses the memory manager. Since many Standard Library routines use the memory manager, you should call this procedure early in the program. The "SHELL.ASM" file makes this call for you.

The malloc routine allocates storage on the heap and returns a pointer to the block it allocates in the es:di registers. Before calling malloc you need to load the size of the block (in bytes) into the cx register. On return, malloc sets the carry flag if an error occurs (insufficient memory). If the carry is clear, es:di points at a block of bytes the size you've specified:

```
mov       cx, 1024              ;Grab 1024 bytes on the heap
malloc                          ;Call MALLOC
jc        MallocError           ;If memory error.
mov       word ptr PNTR, DI     ;Save away pointer to block.
mov       word ptr PNTR+2, ES
```

When you call malloc, the memory manager promises that the block it gives you is free and clear and it will not reallocate that block until you explicitly free it. To return a block of memory back to the memory manager so you can (possibly) re-use that block of memory in the future, use the free Library routine. free expects you to pass the pointer returned by malloc:

```
les       di, PNTR              ;Get pointer to free
free                            ;Free that block
jc        BadFree
```

As usual for most Standard Library routines, if the free routine has some sort of difficulty it will return the carry flag set to denote an error.

7.1.2 The Standard Input Routines: GETC, GETS, GETSM

While the Standard Library provides several input routines, there are three in particular you will use all the time: getc (get a character), gets (get a string), and getsm (get a malloc'd string).

Getc reads a single character from the keyboard and returns that character in the al register. It returns end of file (EOF) status in the ah register (zero means EOF did not occur, one means EOF did occur). It does not modify any other registers. As usual, the carry flag returns the error status. You do not need to pass getc any values in the registers. Getc does not *echo* the input character to the display screen. You must explicitly print the character if you want it to appear on the output monitor.

The following example program continually loops until the user presses the Enter key:

```
; Note: "CR" is a symbol that appears in the "consts.a"
; header file. It is the value 13 which is the ASCII code
; for the carriage return character

Wait4Enter:     getc
                cmp     al, cr
                jne     Wait4Enter
```

The gets routine reads an entire line of text from the keyboard. It stores each successive character of the input line into a byte array whose base address you pass in the es:di register pair. This array must have room for at least 128 bytes. The gets routine will read each character and place it in the array except for the carriage return character. Gets terminates the input line with a zero byte (which is compatible with the Standard Library string handling routines). Gets echoes each character you type to the display device, it also handles simple line editing functions such as backspace. As usual, gets returns the carry set if an error occurs. The following example reads a line of text from the standard input device and then counts the number of characters typed. This code is tricky, note that it initializes the count and pointer to -1 prior to entering the loop and then immediately increments them by one. This sets the count to zero and adjusts the pointer so that it points at the first character in the string. This simplification produces slightly more efficient code than the straightforward solution would produce:

```
DSEG            segment

MyArray         byte    128 dup (?)

DSEG            ends

CSEG            segment

                .
                .
                .
; Note: LESI is a macro (found in consts.a) that loads
; ES:DI with the address of its operand. It expands to the
; code:
;
;               mov di, seg operand
;               mov es, di
;               mov di, offset operand
;
; You will use the macro quite a bit before many Standard
; Library calls.

                lesi    MyArray             ;Get address of inp buf.
                gets                        ;Read a line of text.
                mov     ah, -1              ;Save count here.
                lea     bx, -1[di]          ;Point just before string.
CountLoop:      inc     ah                  ;Bump count by one.
                inc     bx                  ;Point at next char in str.
                cmp     byte ptr es:[bx], 0
                jne     CoutLoop

; Now AH contains the number of chars in the string.

                .
                .
                .
```

The getsm routine also reads a string from the keyboard and returns a pointer to that string in es:di. The difference between gets and getsm is that you do not have to pass the address of an input buffer in es:di. Getsm automatically allocates storage on the heap with a call to malloc and returns a pointer to the buffer in es:di. Don't forget that you must call meminit at the beginning of your program if you use this routine. The SHELL.ASM skeleton file calls meminit for you. Also, don't forget to call free to return the storage to the heap when you're done with the input line.

```
                getsm                       ;Returns pointer in ES:DI
                .
                .
                .
                free                        ;Return storage to heap.
```

7.1.3 The Standard Output Routines: PUTC, PUTCR, PUTS, PUTH, PUTI, PRINT, and PRINTF

The Standard Library provides a wide array of output routines, far more than you will see here. The following routines are representative of the routines you'll find in the Library.

Putc outputs a single character to the display device. It outputs the character appearing in the al register. It does not affect any registers unless there is an error on output (the carry flag denotes error/no error, as usual). See the Standard Library documentation for more details.

Putcr outputs a "newline" (carriage return/line feed combination) to the standard output. It is completely equivalent to the code:

```
mov     al, cr              ;CR and LF are constants
putc                        ; appearing in the consts.a
mov     al, lf              ; header file.
putc
```

The puts (put a string) routine prints the zero terminated string at which es:di points[1]. Note that puts does *not* automatically output a newline after printing the string. You must either put the carriage return/line feed characters at the end of the string or call putcr after calling puts if you want to print a newline after the string. Puts does not affect any registers (unless there is an error). In particular, it does not change the value of the es:di registers. The following code sequence uses this fact:

```
getsm                       ;Read a string
puts                        ;Print it
putcr                       ;Print a new line
free                        ;Free the memory for string.
```

Since the routines above preserve es:di (except, of course, getsm), the call to free deallocates the memory allocated by the call to getsm.

The puth routine prints the value in the al register as exactly two hexadecimal digits, including a leading zero byte if the value is in the range 0..Fh. The following loop reads a sequence of keys from the keyboard and prints their ASCII values until the user presses the Enter key:

```
KeyLoop:    getc
            cmp     al, cr
            je      Done
            puth
            putcr
            jmp     KeyLoop
Done:
```

The puti routine prints the value in ax as a signed 16 bit integer. The following code fragment prints the sum of I and J to the display:

```
mov     ax, I
add     ax, J
puti
putcr
```

Putu is similar to puti except it outputs *unsigned* integer values rather than signed integers.

Routines like puti and putu always output numbers using the minimum number of possible print positions. For example, puti uses three print positions on the string to print the value 123. Sometimes, you may want to force these output routines to print their values using a fixed number of print positions, padding any extra positions with spaces. The putisize and putusize routines provide this capability. These routines expect a numeric value in ax and a field width specification in cx. They will print the number in a field

1. A zero terminated string is a sequence of characters ending with a zero byte. This is the standard character string format the Standard Library uses.

width of *at least* cx positions. If the value in cx is larger than the number of print position the value requires, these routines will right justify the number in a field of cx print positions. If the value in cx is less than the number of print positions the value requires, these routines ignore the value in cx and use however many print positions the number requires.

```
; The following loop prints out the values of a 3x3 matrix in matrix form:
; On entry, bx points at element [0,0] of a row column matrix.

                mov     dx, 3                   ;Repeat for each row.
PrtMatrix:      mov     ax, [bx]                ;Get first element in this
row.
                mov     cx, 7                   ;Use seven print positions.
                putisize                        ;Print this value.
                mov     ax, 2[bx]               ;Get the second element.
                putisize                        ;CX is still seven.
                mov     ax, 4[bx]               ;Get the third element.
                putisize
                putcr                           ;Output a new line.
                add     bx, 6                   ;Move on to next row.
                dec     dx                      ;Repeat for each row.
                jne     PrtMatrix
```

The print routine is one of the most-often called procedures in the library. It prints the zero terminated string that immediately follows the call to print:

```
                print
                byte    "Print this string to the display",cr,lf,0
```

The example above prints the string "Print this string to the display" followed by a new line. Note that print will print whatever characters immediately follow the call to print, up to the first zero byte it encounters. In particular, you can print the newline sequence and any other control characters as shown above. Also note that you are not limited to printing one line of text with the print routine:

```
                print
                byte    "This example of the PRINT routine",cr,lf
                byte    "prints several lines of text.",cr,lf
                byte    "Also note,",cr,lf,"that the source lines "
                byte    "do not have to correspond to the output."
                byte    cr,lf
                byte    0
```

The above displays:

```
This example of the PRINT routine
prints several lines of text.
Also note,
that the source lines do not have to correspond to the output.
```

It is very important that you *not* forget about that zero terminating byte. The print routine begins executing the first 80x86 machine language instruction following that zero terminating byte. If you forget to put the zero terminating byte after your string, the print routine will gladly eat up the instruction bytes following your string (printing them) until it finds a zero byte (zero bytes are common in assembly language programs). This will cause your program to misbehave and is a big source of errors beginning programmers have when they use the print routine. Always keep this in mind.

Printf, like its "C" namesake, provides formatted output capabilities for the Standard Library package. A typical call to printf always takes the following form:

```
                printf
                byte        "format string",0
                dword       operand1, operand2, ..., operandn
```

The format string is comparable to the one provided in the "C" programming language. For most characters, printf simply prints the characters in the format string up to the terminating zero byte. The two exceptions are characters prefixed by a backslash ("\") and characters prefixed by a percent sign ("%"). Like C's printf, the Standard Library's printf

uses the backslash as an escape character and the percent sign as a lead-in to a format string.

Printf uses the escape character ("\") to print special characters in a fashion similar to, but not identical to C's printf. The Standard Library's printf routine supports the following special characters:

- \r Print a carriage return (but no line feed)
- \n Print a new line character (carriage return/line feed).
- \b Print a backspace character.
- \t Print a tab character.
- \l Print a line feed character (but no carriage return).
- \f Print a form feed character.
- \\ Print the backslash character.
- \% Print the percent sign character.
- \0xhh Print ASCII code hh, represented by two hex digits.

C users should note a couple of differences between Standard Library's escape sequences and C's. First, use "\%" to print a percent sign within a format string, not "%%". C doesn't allow the use of "\%" because the C compiler processes "\%" at compile time (leaving a single "%" in the object code) whereas printf processes the format string at run-time. It would see a single "%" and treat it as a format lead-in character. The Standard Library's printf, on the other hand, processes both the "\" and "%" at run-time, therefore it can distinguish "\%".

Strings of the form "\0xhh" must contain exactly two hex digits. The current printf routine isn't robust enough to handle sequences of the form "\0xh" which contain only a single hex digit. Keep this in mind if you find printf chopping off characters after you print a value.

There is absolutely no reason to use any hexadecimal escape character sequence except "\0x00". Printf grabs all characters following the call to printf up to the terminating zero byte (which is why you'd need to use "\0x00" if you want to print the null character, printf will not print such values). The Standard Library's `printf` routine doesn't care how those characters got there. In particular, you are not limited to using a single string after the printf call. The following is perfectly legal:

```
printf
byte "This is a string",13,10
byte "This is on a new line",13,10
byte "Print a backspace at the end of this line:"
byte 8,13,10,0
```

Your code will run a tiny amount faster if you avoid the use of the escape character sequences. More importantly, the escape character sequences take at least two bytes. You can encode most of them as a single byte by simply embedding the ASCII code for that byte directly into the code stream. Don't forget, you cannot embed a zero byte into the code stream. A zero byte terminates the format string. Instead, use the "\0x00" escape sequence.

Format sequences always begin with "%". For each format sequence, you must provide a far pointer to the associated data immediately following the format string, e.g.,

```
printf
byte        "%i %i",0
dword       i,j
```

Format sequences take the general form "%s\cn^f" where:

- "%" is always the "%" character. Use "\%" if you actually want to print a percent sign.
- s is either nothing or a minus sign ("-").
- "\c" is also optional, it may or may not appear in the format item. "c" represents any printable character.
- "n" represents a string of 1 or more decimal digits.
- "^" is just the caret (up-arrow) character.
- "f" represents one of the format characters: i, d, x, h, u, c, s, ld, li, lx, or lu.

The "s", "\c", "n", and "^" items are optional, the "%" and "f" items must be present. Furthermore, the order of these items in the format item is very important. The "\c" entry, for example, cannot precede the "s" entry. Likewise, the "^" character, if present, must follow everything except the "f" character(s).

The format characters i, d, x, h, u, c, s, ld, li, lx, and lu control the output format for the data. The i and d format characters perform identical functions, they tell printf to print the following value as a 16 bit signed decimal integer. The x and h format characters instruct printf to print the specified value as a 16 bit or 8-bit hexadecimal value (respectively). If you specify u, printf prints the value as a 16-bit unsigned decimal integer. Using c tells printf to print the value as a single character. S tells printf that you're supplying the address of a zero-terminated character string, printf prints that string. The ld, li, lx, and lu entries are long (32-bit) versions of d/i, x, and u. The corresponding address points at a 32-bit value that printf will format and print to the standard output.

The following example demonstrates these format items:

```
printf
byte        "I= %i, U= %u, HexC= %h, HexI= %x, C= %c, "
            "S= %s",13,10
byte        "L= %ld",13,10,0
dword       i,u,c,i,c,s,l
```

The number of far addresses (specified by operands to the "dd" pseudo-opcode) must match the number of "%" format items in the format string. Printf counts the number of "%" format items in the format string and skips over this many far addresses following the format string. If the number of items do not match, the return address for printf will be incorrect and the program will probably hang or otherwise malfunction. Likewise (as for the print routine), the format string must end with a zero byte. The addresses of the items following the format string must point directly at the memory locations where the specified data lies.

When used in the format above, printf always prints the values using the minimum number of print positions for each operand. If you want to specify a minimum field width, you can do so using the "n" format option. A format item of the format "%10d" prints a decimal integer using at least ten print positions. Likewise, "%16s" prints a string using at least 16 print positions. If the value to print requires more than the specified number of print positions, printf will use however many are necessary. If the value to print requires fewer, printf will always print the specified number, padding the value with blanks. Printf will print the value right justified in the print field (regardless of the data's type). If you want to print the value left justified in the output file, use the "-" format character as a prefix to the field width, e.g.,

```
printf
byte        "%-17s",0
dword       string
```

In this example, printf prints the string using a 17 character long field with the string left justified in the output field.

By default, printf blank fills the output field if the value to print requires fewer print positions than specified by the format item. The "\c" format item allows you to change the padding character. For example, to print a value, right justified, using "*" as the padding character you would use the format item "%*10d". To print it left justified you would use the format item "%-*10d". Note that the "-" must precede the "*". This is a limitation of the current version of the software. The operands must appear in this order.

Normally, the address(es) following the printf format string must be far pointers to the actual data to print.

On occasion, especially when allocating storage on the heap (using malloc), you may not know (at assembly time) the address of the object you want to print. You may have only a pointer to the data you want to print. The "^" format option tells printf that the far pointer following the format string is the address of a pointer to the data rather than the address of the data itself. This option lets you access the data indirectly.

Note: unlike C, Standard Library's printf routine does not support floating point output. Putting floating point into printf would increase the size of this routine a tremendous amount. Since most people don't need the floating point output facilities, it doesn't appear here. There is a separate routine, printff, that includes floating point output.

The Standard Library printf routine is a complex beast. However, it is very flexible and extremely useful. You should spend the time to master its major functions. You will be using this routine quite a bit in your assembly language programs.

The standard output package provides many additional routines besides those mentioned here. There simply isn't enough room to go into all of them in this chapter. For more details, please consult the Standard Library documentation.

7.1.4 Formatted Output Routines: Putisize, Putusize, Putlsize, and Putulsize

The puti, putu, and putl routines output the numeric strings using the minimum number of print positions necessary. For example, puti uses three character positions to print the value -12. On occasion, you may need to specify a different field width so you can line up columns of numbers or achieve other formatting tasks. Although you can use printf to accomplish this goal, printf has two major drawbacks – it only prints values in memory (i.e., it cannot print values in registers) and the field width you specify for printf must be a constant[2]. The putisize, putusize, and putlsize routines overcome these limitations.

Like their puti, putu, and putl counterparts, these routines print signed integer, unsigned integer, and 32-bitsigned integer values. They expect the value to print in the ax register (putisize and putusize) or the dx:ax register pair (putlsize). They also expect a minimum field width in the cx register. As with printf, if the value in the cx register is smaller than the number of print positions that the number actually needs to print, putisize, putusize, and putlsize will ignore the value in cx and print the value using the minimum necessary number of print positions.

7.1.5 Output Field Size Routines: Isize, Usize, and Lsize

Once in a while you may want to know the number of print positions a value will require before actually printing that value. For example, you might want to compute the maximum print width of a set of numbers so you can print them in columnar format automatically adjusting the field width for the largest number in the set. The isize, usize, and lsize routines do this for you.

The isize routine expects a signed integer in the ax register. It returns the minimum field width of that value (including a position for the minus sign, if necessary) in the ax register. Usize computes the size of the unsigned integer in ax and returns the minimum field width in the ax register. Lsize computes the minimum width of the signed integer in dx:ax (including a position for the minus sign, if necessary) and returns this width in the ax register.

2. Unless you are willing to resort to self-modifying code.

7.1.6 Conversion Routines: ATOx, and xTOA

The Standard Library provides several routines to convert between string and numeric values. These include atoi, atoh, atou, itoa, htoa, wtoa, and utoa (plus others). The ATOx routines convert an ASCII string in the appropriate format to a numeric value and leave that value in ax or al. The ITOx routines convert the value in al/ax to a string of digits and store this string in the buffer whose address is in es:di[3]. There are several variations on each routine that handle different cases. The following paragraphs describe each routine.

The atoi routine assumes that es:di points at a string containing integer digits (and, perhaps, a leading minus sign). They convert this string to an integer value and return the integer in ax. On return, es:di still points at the beginning of the string. If es:di does not point at a string of digits upon entry or if an overflow occurs, atoi returns the carry flag set. Atoi preserves the value of the es:di register pair. A variant of atoi, atoi2, also converts an ASCII string to an integer except it does *not* preserve the value in the di register. The atoi2 routine is particularly useful if you need to convert a sequence of numbers appearing in the same string. Each call to atoi2 leaves the di register pointing at the first character beyond the string of digits. You can easily skip over any spaces, commas, or other delimiter characters until you reach the next number in the string; then you can call atoi2 to convert that string to a number. You can repeat this process for each number on the line.

Atoh works like the atoi routine, except it expects the string to contain hexadecimal digits (no leading minus sign). On return, ax contains the converted 16 bit value and the carry flag denotes error/no error. Like atoi, the atoh routine preserves the values in the es:di register pair. You can call atoh2 if you want the routine to leave the di register pointing at the first character beyond the end of the string of hexadecimal digits.

Atou converts an ASCII string of decimal digits in the range 0..65535 to an integer value and returns this value in ax. Except that the minus sign is not allowed, this routine behaves just like atoi. There is also an atou2 routine that does not preserve the value of the di register; it leaves di pointing at the first character beyond the string of decimal digits.

Since there is no geti, geth, or getu routines available in the Standard Library, you will have to construct these yourself. The following code demonstrates how to read an integer from the keyboard:

```
print
byte        "Enter an integer value:",0
getsm
atoi        ;Convert string to an integer in AX
free        ;Return storage allocated by getsm
print
byte        "You entered ",0
puti        ;Print value returned by ATOI.
putcr
```

The itoa, utoa, htoa, and wtoa routines are the logical inverse to the ato*x* routines. They convert numeric values to their integer, unsigned, and hexadecimal string representations. There are several variations of these routines depending upon whether you want them to automatically allocate storage for the string or if you want them to preserve the di register.

Itoa converts the 16 bit signed integer in ax to a string and stores the characters of this string starting at location es:di. When you call itoa, you must ensure that es:di points at a character array large enough to hold the resulting string. Itoa requires a maximum of seven bytes for the conversion: five numeric digits, a sign, and a zero terminating byte. Itoa preserves the values in the es:di register pair, so upon return es:di points at the beginning of the string produced by itoa.

Occasionally, you may not want to preserve the value in the di register when calling the itoa routine. For example, if you want to create a single string containing several con-

3. There are also a set of xTOAM routines that automatically allocate storage on the heap for you.

verted values, it would be nice if itoa would leave di pointing at the end of the string rather than at the beginning of the string. The itoa2 routine does this for you; it will leave the di register pointing at the zero terminating byte at the end of the string. Consider the following code segment that will produce a string containing the ASCII representations for three integer variables, Int1, Int2, and Int3:

```
; Assume es:di already points at the starting location to store the converted
; integer values

                mov     ax, Int1
                itoa2                           ;Convert Int1 to a string.

; Okay, output a space between the numbers and bump di so that it points
; at the next available position in the string.

                mov     byte ptr es:[di], ' '
                inc     di

; Convert the second value.

                mov     ax, Int2
                itoa2
                mov     byte ptr es:[di], ' '
                inc     di

; Convert the third value.

                mov     ax, Int3
                itoa2

; At this point, di points at the end of the string containing the
; converted values. Hopefully you still know where the start of the
; string is so you can manipulate it!
```

Another variant of the itoa routine, itoam, does not require you to initialize the es:di register pair. This routine calls malloc to automatically allocate the storage for you. It returns a pointer to the converted string on the heap in the es:di register pair. When you are done with the string, you should call free to return its storage to the heap.

```
; The following code fragment converts the integer in AX to a string and prints
; this string. Of course, you could do this same operation with PUTI, but this
; code does demonstrate how to call itoam.

                itoam                           ;Convert integer to string.
                puts                            ;Print the string.
                free                            ;Return storage to the heap.
```

The utoa, utoa2, and utoam routines work just like itoa, itoa2, and itoam, except they convert the unsigned integer value in ax to a string. Note that utoa and utoa2 require, at most, six bytes since they never output a sign character.

Wtoa, wtoa2, and wtoam convert the 16 bit value in ax to a string of exactly four hexadecimal characters plus a zero terminating byte. Otherwise, they behave exactly like itoa, itoa2, and itoam. Note that these routines output leading zeros so the value is always four digits long.

The htoa, htoa2, and htoam routines are similar to the wtoa, wtoa2, and wtoam routines. However, the hto*x* routines convert the eight bit value in al to a string of two hexadecimal characters plus a zero terminating byte.

The Standard Library provides several other conversion routines as well as the ones mentioned in this section. See the Standard Library documentation in the appendices for more details.

7.1.7 Routines that Test Characters for Set Membership

The UCR Standard Library provides many routines that test the character in the al register to see if it falls within a certain set of characters. These routines all return the status in the zero flag. If the condition is true, they return the zero flag set (so you can test the con-

dition with a je instruction). If the condition is false, they clear the zero flag (test this condition with jne). These routines are

- IsAlNum- Checks to see if al contains an alphanumeric character.
- IsXDigit- Checks al to see if it contains a hexadecimal digit character.
- IsDigit- Checks al to see if it contains a decimal digit character.
- IsAlpha- Checks al to see if it contains an alphabetic character.
- IsLower- Checks al to see if it contains a lower case alpha character.
- IsUpper- Checks al to see if it contains an upper case alpha character.

7.1.8 Character Conversion Routines: ToUpper, ToLower

The ToUpper and ToLower routines check the character in the al register. They will convert the character in al to the appropriate alphabetic case.

If al contains a lower case alphabetic character, ToUpper will convert it to the equivalent upper case character. If al contains any other character, ToUpper will return it unchanged.

If al contains an upper case alphabetic character, ToLower will convert it to the equivalent lower case character. If the value is not an upper case alphabetic character ToLower will return it unchanged.

7.1.9 Random Number Generation: Random, Randomize

The Standard Library Random routine generates a sequence of pseudo-random numbers. It returns a random value in the ax register on each call. You can treat this value as a signed or unsigned value since Random manipulates all 16 bits of the ax register.

You can use the div and idiv instructions to force the output of random to a specific range. Just divide the value random returns by some number *n* and the remainder of this division will be a value in the range 0..*n*-1. For example, to compute a random number in the range 1..10, you could use code like the following:

```
random                  ;Get a random number in range 0..65535.
sub       dx, dx        ;Zero extend to 16 bits.
mov       bx, 10        ;Want value in the range 1..10.
div       bx            ;Remainder goes to dx!
inc       dx            ;Convert 0..9 to 1..10.

; At this point, a random number in the range 1..10 is in the dx register.
```

The random routine always returns the same sequence of values when a program loads from disk and executes. Random uses an internal table of *seed* values that it stores as part of its code. Since these values are fixed, and always load into memory with the program, the algorithm that random uses will always produce the same sequence of values when a program containing it loads from the disk and begins running. This might not seem very "random" but, in fact, this is a nice feature since it is very difficult to test a program that uses truly random values. If a random number generator always produces the same sequence of numbers, any tests you run on that program will be repeatable.

Unfortunately, there are many examples of programs that you may want to write (e.g., games) where having repeatable results is not acceptable. For these applications you can call the randomize routine. Randomize uses the current value of the time of day clock to generate a nearly random starting sequence. So if you need a (nearly) unique sequence of random numbers each time your program begins execution, call the randomize routine once before ever calling the random routine. Note that there is little benefit to calling the randomize routine more than once in your program. Once random establishes a random starting point, further calls to randomize will not improve the quality (randomness) of the numbers it generates.

7.1.10 Constants, Macros, and other Miscellany

When you include the "stdlib.a" header file, you are also defining certain macros (see Chapter Eight for a discussion of macros) and commonly used constants. These include the following:

```
NULL            =       0           ;Some common ASCII codes
BELL            =       07          ;Bell character
bs              =       08          ;Backspace character
tab             =       09          ;Tab character
lf              =       0ah         ;Line feed character
cr              =       0dh         ;Carriage return
```

In addition to the constants above, "stdlib.a" also defines some useful macros including ExitPgm, lesi, and ldxi. These macros contain the following instructions:

```
; ExitPgm- Returns control to MS-DOS

ExitPgm         macro
                mov     ah, 4ch     ;DOS terminate program opcode
                int     21h         ;DOS call.
                endm

; LESI ADRS-
;       Loads ES:DI with the address of the specified operand.

lesi            macro   adrs
                mov     di, seg adrs
                mov     es, di
                mov     di, offset adrs
                endm

; LDXI ADRS-
;       Loads DX:SI with the address of the specified operand.

ldxi            macro   adrs
                mov     dx, seg adrs
                mov     si, offset adrs
                endm
```

The lesi and ldxi macros are especially useful for load addresses into es:di or dx:si before calling various standard library routines (see Chapter Seven for details about macros).

7.1.11 Plus more!

The Standard Library contains many, many, routines that this chapter doesn't even mention. As you get time, you should read through the documentation for the Standard Library and find out what's available. The routines mentioned in this chapter are the ones you will use right away. This text will introduce new Standard Library routines as they are needed.

7.2 Sample Programs

The following programs demonstrate some common operations that use the Standard Library.

7.2.1 Stripped SHELL.ASM File

```
; Sample Starting SHELL.ASM file
;
; Randall Hyde
; Version 1.0
; 2/6/96
;
; This file shows what the SHELL.ASM file looks like without
; the superfluous comments that explain where to place objects
; in the source file.  Your programs should likewise begin
; with a stripped version of the SHELL.ASM file.  After all,
; the comments in the original SHELL.ASM file are four *your*
; consumption, not to be read by someone who sees the program
; you wind up writing.

                .xlist
                include     stdlib.a
                includelib  stdlib.lib
                .list

dseg            segment  para public 'data'

dseg            ends

cseg            segment  para public 'code'
                assume   cs:cseg, ds:dseg

Main            proc
                mov      ax, dseg
                mov      ds, ax
                mov      es, ax
                meminit




Quit:           ExitPgm
Main            endp

cseg            ends

sseg            segment  para stack 'stack'
stk             db       1024 dup ("stack   ")
sseg            ends

zzzzzzseg       segment  para public 'zzzzzz'
LastBytes       db       16 dup (?)
zzzzzzseg       ends
                end      Main
```

7.2.2 Numeric I/O

```
; Pgm7_2.asm - Numeric I/O.
;
; Randall Hyde
; 2/6/96
;
; The standard library routines do not provide simple to use numeric input
; routines.  This code demonstrates how to read decimal and hexadecimal values
; from the user using the Getsm, ATOI, ATOU, ATOH, IsDigit, and IsXDigit
routines.
```

```
                .xlist
                include     stdlib.a
                includelib  stdlib.lib
                .list

dseg            segment  para public 'data'

inputLine       byte     128 dup (0)

SignedInteger   sword    ?
UnsignedInt     word     ?
HexValue        word     ?

dseg            ends

cseg            segment  para public 'code'
                assume   cs:cseg, ds:dseg

Main            proc
                mov      ax, dseg
                mov      ds, ax
                mov      es, ax
                meminit


; Read a signed integer value from the user.

InputInteger:   print
                byte     "Input a signed integer value: ",0

                lesi     inputLine    ;Point es:di at inputLine buffer
                gets                  ;Read a line of text from the user.

                mov      bx, -1
SkipSpcs1:      inc      bx
                cmp      inputLine[bx], ' '      ;Skip over any spaces.
                je       SkipSpcs1

                cmp      inputLine[bx], '-'      ;See if it's got a minus sign
                jne      NoSign
                inc      bx                      ;Skip if a negative number

NoSign:         dec      bx                      ;Back up one place.
TestDigs:       inc      bx                      ;Move on to next char
                mov      al, inputLine[bx]
                IsDigit                          ;See if it's a decimal digit.
                je       TestDigs                ;Repeat process if it is.

                cmp      inputLine[bx], ' '      ;See if we end with a
                je       GoodDec                 ; reasonable character.
                cmp      inputLine[bx], ','
                je       GoodDec
                cmp      inputLine[bx], 0        ;Input line ends with a zero.
                je       GoodDec
                printf
                byte     "'%s' is an illegal signed integer.  "
                byte     "Please reenter.",cr,lf,0
                dword    inputLine
                jmp      InputInteger

; Okay, all the characters are cool, let's do the conversion here.  Note that
; ES:DI is still pointing at inputLine.

GoodDec:        ATOI                             ;Do the conversion
                mov      SignedInteger, ax       ;Save the value away.

; Read an unsigned integer value from the user.

InputUnsigned:  print
                byte     "Input an unsigned integer value: ",0

                lesi     inputLine    ;Point es:di at inputLine buffer
                gets                  ;Read a line of text from the user.

; Note the sneakiness in the following code.  It starts with an index of -2
; and then increments it by one.  When accessing data in this loop it compares
```

```
; against locatoin inputLine[bx+1] which effectively starts bx at zero.  In the
; "TestUnsigned" loop below, this code increments bx again so that bx then
; contains the index into the string when the action is occuring.

                mov     bx, -2
SkipSpcs2:      inc     bx
                cmp     inputLine[bx+1], ' '    ;Skip over any spaces.
                je      SkipSpcs2

TestUnsigned:   inc     bx                      ;Move on to next char
                mov     al, inputLine[bx]
                IsDigit                         ;See if it's a decimal digit.
                je      TestUnsigned            ;Repeat process if it is.

                cmp     inputLine[bx], ' '      ;See if we end with a
                je      GoodUnSigned            ; reasonable character.
                cmp     inputLine[bx], ','
                je      GoodUnsigned
                cmp     inputLine[bx], 0        ;Input line ends with a zero.
                je      GoodUnsigned
                printf
                byte    "'%s' is an illegal unsigned integer.  "
                byte    "Please reenter.",cr,lf,0
                dword   inputLine
                jmp     InputUnsigned

; Okay, all the characters are cool, let's do the conversion here.  Note that
; ES:DI is still pointing at inputLine.

GoodUnsigned:   ATOU                            ;Do the conversion
                mov     UnsignedInt, ax         ;Save the value away.


; Read a hexadecimal value from the user.

InputHex:       print
                byte    "Input a hexadecimal value: ",0

                lesi    inputLine    ;Point es:di at inputLine buffer
                gets                 ;Read a line of text from the user.

; The following code uses the same sneaky trick as the code above.

                mov     bx, -2
SkipSpcs3:      inc     bx
                cmp     inputLine[bx+1], ' '    ;Skip over any spaces.
                je      SkipSpcs3

TestHex:        inc     bx                      ;Move on to next char
                mov     al, inputLine[bx]
                IsXDigit                        ;See if it's a hex digit.
                je      TestHex                 ;Repeat process if it is.

                cmp     inputLine[bx], ' '      ;See if we end with a
                je      GoodHex                 ; reasonable character.
                cmp     inputLine[bx], ','
                je      GoodHex
                cmp     inputLine[bx], 0        ;Input line ends with a zero.
                je      GoodHex
                printf
                byte    "'%s' is an illegal hexadecimal value.  "
                byte    "Please reenter.",cr,lf,0
                dword   inputLine
                jmp     InputHex

; Okay, all the characters are cool, let's do the conversion here.  Note that
; ES:DI is still pointing at inputLine.

GoodHex:        ATOH                            ;Do the conversion
                mov     HexValue, ax            ;Save the value away.


; Display the results:

                printf
```

```
                        byte    "Values input:",cr,lf
                        byte    "Signed:   %4d",cr,lf
                        byte    "Unsigned: %4d",cr,lf
                        byte    "Hex:      %4x",cr,lf,0
                        dword   SignedInteger, UnsignedInt, HexValue

Quit:                   ExitPgm
Main                    endp

cseg                    ends

sseg                    segment para stack 'stack'
stk                     db      1024 dup ("stack   ")
sseg                    ends

zzzzzzseg               segment para public 'zzzzzz'
LastBytes               db      16 dup (?)
zzzzzzseg               ends
                        end     Main
```

7.3 Laboratory Exercises

The UCR Standard Library for 80x86 Assembly Language Programmers is available, nearly ready to use, on the companion CD-ROM. In this set of laboratory exercises you will learn how to install the Standard Library on a local hard disk and access the library within your programs.

7.3.1 Obtaining the UCR Standard Library

A recent version of the UCR Standard Library for 80x86 Assembly language programmers appears on the companion CD-ROM. There are, however, periodic updates to the library, so it is quite possible that the version on the CD-ROM is out of date. For most of the projects and examples in this textbook, the version appearing on the CD-ROM is probably sufficient[4]. However, if you want to use the Standard Library to develop your own assembly language software you'll probably want to have the latest version of the library.

The official repository for the UCR Standard library is the ftp.cs.ucr.edu ftp site at the University of California, Riverside. If you have Internet/ftp access, you can download the latest copy of the standard library directly from UCR using an *anonymous ftp account*. To obtain the software over the internet, follow these steps:

- Running your ftp program, connect to ftp.cs.ucr.edu.
- When the system asks for your login name, use *anonymous*.
- When the system asks for your password, use your full login name (e.g., something that looks like *name@machine.domain*).
- At this point, you should be logged onto the system. Switch to the \pub\pc\ibmpcdir using a "cd pub\pc\ibmpcdir" UNIX command.
- The Standard Library files are compressed binary files. Therefore, you must switch ftp to its *binary* (vs. ASCII) mode before downloading the files. On a standard ftp program you would enter a "binary" command to accomplish this. Check the documentation for your ftp program to see how to do this. **The default for download is usually ASCII. If you download the standard library files in ASCII mode, they will probably fail to uncompress properly.**
- In the \pub\pc\ibmpcdir subdirectory you should find several files (generally five but there may be more). Using the appropriate ftp commands (e.g., get or mget), copy these files to your local system.
- Log off the UCR ftp computer and quit your ftp program.

4. Indeed, the only reason to get an update for this text would be to obtain bug fixes.

- If you have been running ftp on a UNIX system, you will need to transfer the files you've downloaded to a PC running DOS or Windows. Consult your instructor or local UNIX system administrator for details.
- That's it! You've now downloaded the latest version of the Standard Library.

If you do not have Internet access, or there is some problem accessing the ftp site at UCR, you can probably locate a copy of the Standard Library at other ftp sites, on other BBSes, or from a shareware vendor. Keep in mind, however, that software you find at other sites may be quite old (indeed, they may have older versions than that appearing on the companion CD-ROM).

For your lab report: If you successfully downloaded the latest version of the library, describe the process you went through. Also, describe the files that you downloaded from the ftp site. If there were any "readme" files you downloaded, read them and describe their content in your lab report.

7.3.2 Unpacking the Standard Library

To reduce disk storage requirements and download time, the UCR Standard Library is compressed. Once you download the files from an ftp site or some other service, you will have to uncompress the files in order to use them. Note: there is a compressed version of the Standard Library on the companion CD-ROM in the event you do not have Internet access and could not download the files in the previous exercises. See the Chapter Seven subdirectory on the companion CD-ROM. Decompressing the Standard Library is nearly an automatic process. Just follow these steps:

- Create a directory on your local hard disk (usually C:) named "STDLIB".[5] Switch to this subdirectory using the command "CD C:\STDLIB".
- Copy the files you downloaded (or the files off the companion CD-ROM in the STDLIB\DIST subdirectory) into the STDLIB subdirectory you've just created.
- Execute the DOS command "PATH=C:\STDLIB".
- Execute the "UNPACK.BAT" batch file by typing "UNPACK" at the DOS command line prompt.
- Sit back and watch the show. Everything else is automatic.
- You should reboot after unpacking the standard library or reset the path to its original value.

If you did not set the path to include the STDLIB directory, the UNPACK.BAT file will report several errors and it will not properly unpack the files. It *will* delete the compressed files from the disk. Therefore, make sure you save a copy of the files you downloaded on a floppy disk or in a different directory when unpacking the Standard Library. Doing so will save you from having to download the STDLIB files again if something goes wrong during the decompression phase.

For your lab report: Describe the directory structure that unpacking the standard library produces.

7.3.3 Using the Standard Library

When you unpack the Standard Library files, the UNPACK.BAT program leaves a (full) copy of the SHELL.ASM file sitting in the STDLIB subdirectory. This should be a familiar file since you've been using SHELL.ASM as a skeletal assembly language program in past projects. This particular version of SHELL.ASM is a "full" version since it

5. If you are doing this on computer systems in your school's laboratories, they may ask you to use a different subdirectory since the Standard Library may already be installed on the machines.

contains several comments that explain where user-written code and variables should go in the file. As a general rule, it is very bad programming style to leave these comments in your SHELL.ASM file. Once you've read these comments and figured out the layout of the SHELL.ASM file, you should delete those comments from any program you write based on the SHELL.ASM file.

For your lab report: include a modified version of the SHELL.ASM file with the superfluous comments removed.

At the beginning of the SHELL.ASM file, you will find the following two statements:

```
include     stdlib.a
includelib  stdlib.lib
```

The first statement tells MASM to read the definitions for the standard library routines from the STDLIB.A *include file* (see Chapter Eight for a description of include files). The second statement tells MASM to pass the name of the STDLIB.LIB object code file on to the linker so it can link your program with the code in the Standard Library. The exact nature of these two statements is unimportant at this time; however, to use the Standard Library routines, MASM needs to be able to *find* these two files at assembly and link time. By default, MASM assumes that these two files are in the current subdirectory whenever you assemble a program based on SHELL.ASM. Since this is not the case, you will have to execute two special DOS commands to tell MASM where it can find these files. The two commands are

```
set include=c:\stdlib\include
set lib=c:\stdlib\lib
```

If you do not execute these commands at least once prior to using MASM with SHELL.ASM for the first time, MASM will report an error (that it cannot find the STDLIB.A file) and abort the assembly.

For your lab report: Execute the DOS commands "SET INCLUDE=C:\" and "SET LIB=C:\"[6] and then attempt to assemble SHELL.ASM using the DOS command:

```
ml shell.asm
```

Report the error in your lab report. Now execute

```
SET INCLUDE=C:\STDLIB\INCLUDE
```

Assemble SHELL.ASM again and report any errors. Finally, execute LIB set command and assemble your program (hopefully) without error.

If you want to avoid having to execute the SET commands every time you sit down to program in assembly language, you can always add these set commands to your autoexec.bat file. If you do this, the system will automatically execute these commands whenever you turn it on.

Other programs (like MASM and Microsoft C++) may also be using SET LIB and SET INCLUDE commands. If there are already SET INCLUDE or SET LIB commands in your autoexec.bat file, you should append the Standard Library entrys to the end of the existing command like the following:

```
set include=c:\MASM611\include;c:\STDLIB\INCLUDE
set lib=c:\msvc\lib;C:\STDLIB\LIB
```

7.3.4 The Standard Library Documentation Files

There are several hundred routines in the UCR Standard Library; far more than this chapter can reasonably document. The "official" source of documentation for the UCR Standard Library is a set of text files appearing in the C:\STDLIB\DOC directory. These files are text files (that you can read with any text editor) that describe the use of each of

6. These command deactivate any current LIB or INCLUDE strings in the environment variables.

the Standard Library routines. If you have any questions about a subroutine or you want to find out what routines are available, you should read the files in this subdirectory.

The documentation consists of several text files organized by routine classification. For example, one file describes output routines, another describes input routines, and yet another describes the string routines. The SHORTREF.TXT file provides a quick synopsis of the entire library. This is a good starting point for information about the routines in the library.

For your lab report: include the names of the text files appearing in the documentation directory. Provide the names of several routines that are docuemented within each file.

7.4 Programming Projects

1) Write any program of your choice that uses at least fifteen different UCR Standard Library routines. Consult the appendix in your textbook and the STDLIB\DOC directory for details on the various StdLib routines. At least five of the routines you choose should *not* appear in this chapter. Learn those routines yourself by studying the UCR StdLib documentation.

2) Write a program that demonstrates the use of each of the format options in the PRINTF StdLib routine.

3) Write a program that reads 16 signed integers from a user and stores these values into a 4x4 matrix. The program should then print the 4x4 matrix in matrix form (i.e., four rows of four numbers with each column nicely aligned).

4) Modify the program in problem (3) above so that figures out which number requires the largest number of print positions and then it outputs the matrix using this value plus one as the field width for all the numbers in the matrix. For example, if the largest number in the matrix is 1234, then the program would print the numbers in the matrix using a minimum field width of five.

7.5 Summary

This chapter introduced several assembler directives and pseudo-opcodes supported by MASM. It also briefly discussed some routines in the UCR Standard Library for 80x86 Assembly Language Programmers. This chapter, by no means, is a complete description of what MASM or the Standard Library has to offer. It does provide enough information to get you going.

To help you write assembly language programs with a minimum of fuss, this text makes extensive use of various routines from the UCR Standard Library for 80x86 Assembly Language Programmers. Although this chapter could not possibly begin to cover all the Standard Library routines, it does discuss many of the routines that you'll use right away. This text will discuss other routines as necessary.

- See "An Introduction to the UCR Standard Library" on page 333.
- See "Memory Management Routines: MEMINIT, MALLOC, and FREE" on page 334.
- See "The Standard Input Routines: GETC, GETS, GETSM" on page 334.
- See "The Standard Output Routines: PUTC, PUTCR, PUTS, PUTH, PUTI, PRINT, and PRINTF" on page 336.
- See "Conversion Routines: ATOx, and xTOA" on page 341.
- "Formatted Output Routines: Putisize, Putusize, Putlsize, and Putulsize" on page 340
- "Output Field Size Routines: Isize, Usize, and Lsize" on page 340
- "Routines that Test Characters for Set Membership" on page 342

7.6 Questions

1. What file should you use to begin your programs when writing code that uses the UCR Standard Library?
2. What routine allocates storage on the heap?
3. What routine would you use to print a single character?
4. What routines allow you to print a literal string of characters to the display?
5. The Standard Library does not provide a routine to read an integer from the user. Describe how to use the GETS and ATOI routines to accomplish this task.
6. What is the difference between the GETS and GETSM routines?
7. What is the difference between the ATOI and ATOI2 routines?
8. What does the ITOA routine do? Describe input and output values.