Strings and Character Sets — Chapter 15

A string is a collection of objects stored in contiguous memory locations. Strings are usually arrays of bytes, words, or (on 80386 and later processors) double words. The 80x86 microprocessor family supports several instructions specifically designed to cope with strings. This chapter explores some of the uses of these string instructions.

The 8088, 8086, 80186, and 80286 can process two types of strings: byte strings and word strings. The 80386 and later processors also handle double word strings. They can move strings, compare strings, search for a specific value within a string, initialize a string to a fixed value, and do other primitive operations on strings. The 80x86's string instructions are also useful for manipulating arrays, tables, and records. You can easily assign or compare such data structures using the string instructions. Using string instructions may speed up your array manipulation code considerably.

15.0 Chapter Overview

This chapter presents a review of the operation of the 80x86 string instructions. Then it discusses how to process character strings using these instructions. Finally, it concludes by discussing the string instruction available in the UCR Standard Library. The sections below that have a "•" prefix are essential. Those sections with a "❏" discuss advanced topics that you may want to put off for a while.

- • The 80x86 string instructions.
- • Character strings.
- • Character string functions.
- • String functions in the UCR Standard Library.
- ❏ Using the string instructions on other data types.

15.1 The 80x86 String Instructions

All members of the 80x86 family support five different string instructions: movs, cmps, scas, lods, and stos[1]. They are the string primitives since you can build most other string operations from these five instructions. How you use these five instructions is the topic of the next several sections.

15.1.1 How the String Instructions Operate

The string instructions operate on blocks (contiguous linear arrays) of memory. For example, the movs instruction moves a sequence of bytes from one memory location to another. The cmps instruction compares two blocks of memory. The scas instruction scans a block of memory for a particular value. These string instructions often require three operands, a destination block address, a source block address, and (optionally) an element count. For example, when using the movs instruction to copy a string, you need a source address, a destination address, and a count (the number of string elements to move).

Unlike other instructions which operate on memory, the string instructions are single-byte instructions which don't have any explicit operands. The operands for the string instructions include

1. The 80186 and later processor support two additional string instructions, INS and OUTS which input strings of data from an input port or output strings of data to an output port. We will not consider these instructions in this chapter.

- the si (source index) register,
- the di (destination index) register,
- the cx (count) register,
- the ax register, and
- the direction flag in the FLAGS register.

For example, one variant of the movs (move string) instruction copies a string from the source address specified by ds:si to the destination address specified by es:di, of length cx. Likewise, the cmps instruction compares the string pointed at by ds:si, of length cx, to the string pointed at by es:di.

Not all instructions have source and destination operands (only movs and cmps support them). For example, the scas instruction (scan a string) compares the value in the accumulator to values in memory. Despite their differences, the 80x86's string instructions all have one thing in common – using them requires that you deal with two segments, the data segment and the extra segment.

15.1.2 The REP/REPE/REPZ and REPNZ/REPNE Prefixes

The string instructions, by themselves, do not operate on strings of data. The movs instruction, for example, will move a single byte, word, or double word. When executed by itself, the movs instruction ignores the value in the cx register. The repeat prefixes tell the 80x86 to do a multi-byte string operation. The syntax for the repeat prefix is:

```
Field:
Label   repeat    mnemonic   operand              ;comment

For MOVS:
        rep       movs       {operands}

For CMPS:
        repe      cmps       {operands}
        repz      cmps       {operands}
        repne     cmps       {operands}
        repnz     cmps       {operands}

For SCAS:
        repe      scas       {operands}
        repz      scas       {operands}
        repne     scas       {operands}
        repnz     scas       {operands}

For STOS:
        rep       stos       {operands}
```

You don't normally use the repeat prefixes with the lods instruction.

As you can see, the presence of the repeat prefixes introduces a new field in the source line – the repeat prefix field. This field appears only on source lines containing string instructions. In your source file:

- the label field should always begin in column one,
- the repeat field should begin at the first tab stop, and
- the mnemonic field should begin at the second tab stop.

When specifying the repeat prefix before a string instruction, the string instruction repeats cx times[2]. Without the repeat prefix, the instruction operates only on a single byte, word, or double word.

2. Except for the cmps instruction which repeats *at most* the number of times specified in the cx register.

You can use repeat prefixes to process entire strings with a single instruction. You can use the string instructions, without the repeat prefix, as string primitive operations to synthesize more powerful string operations.

The operand field is optional. If present, MASM simply uses it to determine the size of the string to operate on. If the operand field is the name of a byte variable, the string instruction operates on bytes. If the operand is a word address, the instruction operates on words. Likewise for double words. If the operand field is not present, you must append a "B", "W", or "D" to the end of the string instruction to denote the size, e.g., movsb, movsw, or movsd.

15.1.3 The Direction Flag

Besides the si, di, si, and ax registers, one other register controls the 80x86's string instructions – the flags register. Specifically, the *direction flag* in the flags register controls how the CPU processes strings.

If the direction flag is clear, the CPU increments si and di after operating upon each string element. For example, if the direction flag is clear, then executing movs will move the byte, word, or double word at ds:si to es:di and will increment si and di by one, two, or four. When specifying the rep prefix before this instruction, the CPU increments si and di for each element in the string. At completion, the si and di registers will be pointing at the first item beyond the string.

If the direction flag is set, then the 80x86 decrements si and di after processing each string element. After a repeated string operation, the si and di registers will be pointing at the first byte or word before the strings if the direction flag was set.

The direction flag may be set or cleared using the cld (clear direction flag) and std (set direction flag) instructions. When using these instructions inside a procedure, keep in mind that they modify the machine state. Therefore, you may need to save the direction flag during the execution of that procedure. The following example exhibits the kinds of problems you might encounter:

```
StringStuff:
                 cld
       <do some operations>
                 call     Str2
       <do some string operations requiring D=0>
                   .
                   .
                   .
Str2             proc     near
                 std
       <Do some string operations>
                 ret
Str2             endp
```

This code will not work properly. The calling code assumes that the direction flag is clear after Str2 returns. However, this isn't true. Therefore, the string operations executed after the call to Str2 will not function properly.

There are a couple of ways to handle this problem. The first, and probably the most obvious, is always to insert the cld or std instructions immediately before executing a string instruction. The other alternative is to save and restore the direction flag using the pushf and popf instructions. Using these two techniques, the code above would look like this:

Always issuing cld or std before a string instruction:

```
StringStuff:
                 cld
       <do some operations>
                 call     Str2
                 cld
       <do some string operations requiring D=0>
```

```
                 .
                 .
                 .
Str2             proc     near
                 std
       <Do some string operations>
                 ret
Str2             endp
```

Saving and restoring the flags register:

```
StringStuff:
                 cld
       <do some operations>
                 call     Str2
       <do some string operations requiring D=0>
                 .
                 .
                 .
Str2             proc     near
                 pushf
                 std
       <Do some string operations>
                 popf
                 ret
Str2             endp
```

If you use the pushf and popf instructions to save and restore the flags register, keep in mind that you're saving and restoring all the flags. Therefore, such subroutines cannot return any information in the flags. For example, you will not be able to return an error condition in the carry flag if you use pushf and popf.

15.1.4 The MOVS Instruction

The movs instruction takes four basic forms. Movs moves bytes, words, or double words, movsb moves byte strings, movsw moves word strings, and movsd moves double word strings (on 80386 and later processors). These four instructions use the following syntax:

```
{REP} MOVSB
{REP} MOVSW
{REP} MOVSD          ;Available only on 80386 and later processors
{REP} MOVS Dest, Source
```

The movsb (move string, bytes) instruction fetches the byte at address ds:si, stores it at address es:di, and then increments or decrements the si and di registers by one. If the rep prefix is present, the CPU checks cx to see if it contains zero. If not, then it moves the byte from ds:si to es:di and decrements the cx register. This process repeats until cx becomes zero.

The movsw (move string, words) instruction fetches the word at address ds:si, stores it at address es:di, and then increments or decrements si and di by two. If there is a rep prefix, then the CPU repeats this procedure as many times as specified in cx.

The movsd instruction operates in a similar fashion on double words. Incrementing or decrementing si and di by four for each data movement.

MASM automatically figures out the size of the movs instruction by looking at the size of the operands specified. If you've defined the two operands with the byte (or comparable) directive, then MASM will emit a movsb instruction. If you've declared the two labels via word (or comparable), MASM will generate a movws instruction. If you've declared the two labels with dword, MASM emits a movsd instruction. The assembler will also check the segments of the two operands to ensure they match the current assumptions (via the assume directive) about the es and ds registers. You should always use the movsb, movsw, and movsd forms and forget about the movs form.

Although, in theory, the movs form appears to be an elegant way to handle the move string instruction, in practice it creates more trouble than it's worth. Furthermore, this form of the move string instruction implies that movs has explicit operands, when, in fact, the si and di registers implicitly specify the operands. For this reason, we'll always use the movsb, movsw, or movsd instructions. When used with the rep prefix, the movsb instruction will move the number of bytes specified in the cx register. The following code segment copies 384 bytes from String1 to String2:

```
                  cld
                  lea      si, String1
                  lea      di, String2
                  mov      cx, 384
        rep       movsb
                   .
                   .
                   .
String1           byte     384 dup (?)
String2           byte     384 dup (?)
```

This code, of course, assumes that String1 and String2 are in the same segment and both the ds and es registers point at this segment. If you substitute movws for movsb, then the code above will move 384 words (768 bytes) rather than 384 bytes:

```
                  cld
                  lea      si, String1
                  lea      di, String2
                  mov      cx, 384
        rep       movsw
                   .
                   .
                   .
String1           word     384 dup (?)
String2           word     384 dup (?)
```

Remember, the cx register contains the element count, not the byte count. When using the movsw instruction, the CPU moves the number of words specified in the cx register.

If you've set the direction flag before executing a movsb/movsw/movsd instruction, the CPU decrements the si and di registers after moving each string element. This means that the si and di registers must point at the end of their respective strings before issuing a movsb, movsw, or movsd instruction. For example,

```
                  std
                  lea      si, String1+383
                  lea      di, String2+383
                  mov      cx, 384
        rep       movsb
                   .
                   .
                   .
String1           byte     384 dup (?)
String2           byte     384 dup (?)
```

Although there are times when processing a string from tail to head is useful (see the cmps description in the next section), generally you'll process strings in the forward direction since it's more straightforward to do so. There is one class of string operations where being able to process strings in both directions is absolutely mandatory: processing strings when the source and destination blocks overlap. Consider what happens in the following code:

```
                  cld
                  lea      si, String1
                  lea      di, String2
                  mov      cx, 384
        rep       movsb
                   .
                   .
                   .
String1           byte     ?
String2           byte     384 dup (?)
```

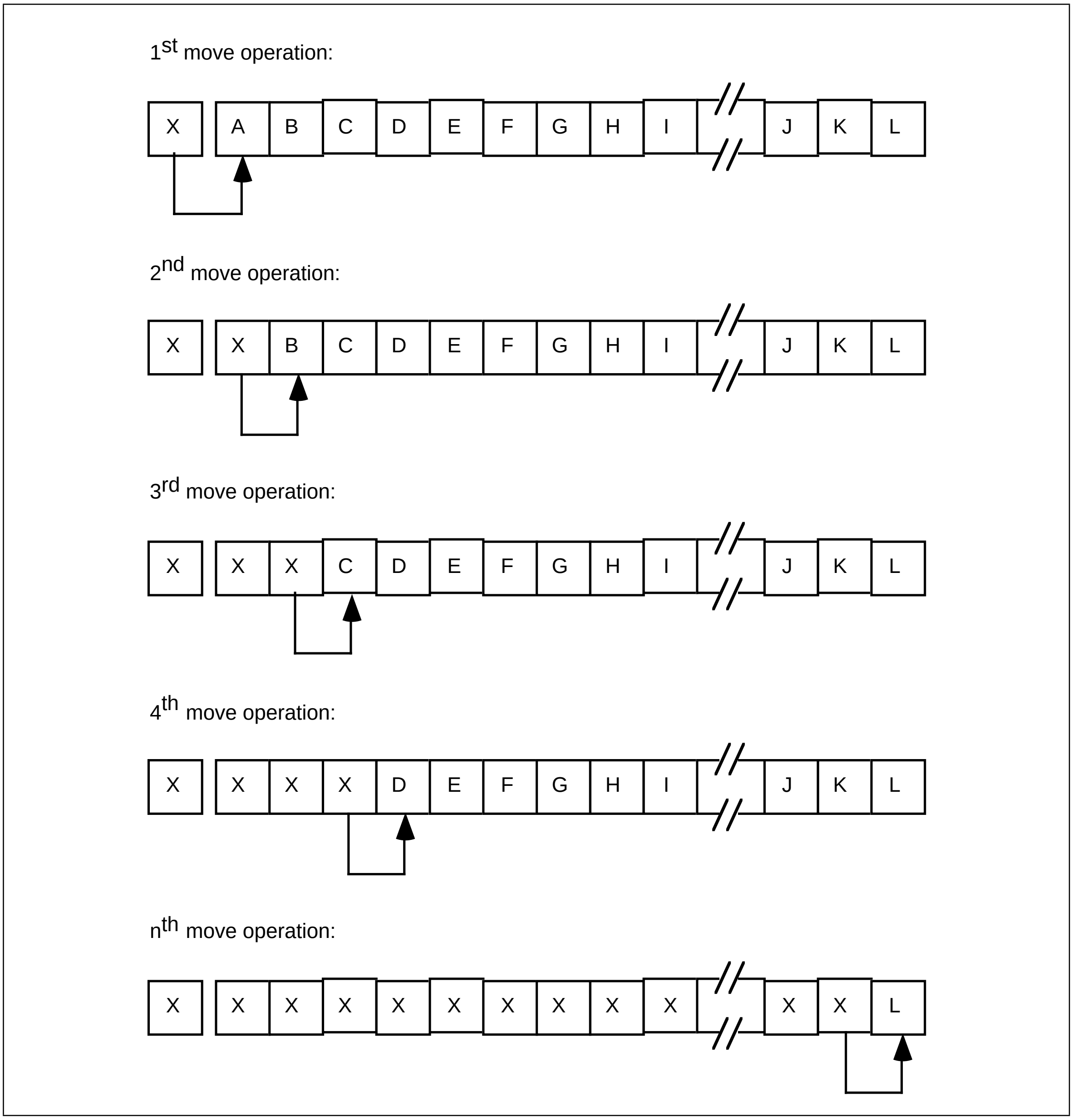

Figure 15.1 Overwriting Data During a Block Move Operation

This sequence of instructions treats String1 and String2 as a pair of 384 byte strings. However, the last 383 bytes in the String1 array overlap the first 383 bytes in the String2 array. Let's trace the operation of this code byte by byte.

When the CPU executes the movsb instruction, it copies the byte at ds:si (String1) to the byte pointed at by es:di (String2). Then it increments si and di, decrements cx by one, and repeats this process. Now the si register points at String1+1 (which is the address of String2) and the di register points at String2+1. The movsb instruction copies the byte pointed at by si to the byte pointed at by di. However, this is the byte originally copied from location String1. So the movsb instruction copies the value originally in location String1 to both locations String2 and String2+1. Again, the CPU increments si and di, decrements cx, and repeats this operation. Now the movsb instruction copies the byte from location String1+2 (String2+1) to location String2+2. But once again, this is the value that originally appeared in location String1. Each repetition of the loop copies the next element in String1 to the next available location in the String2 array. Pictorially, it looks something like that in Figure 15.1.

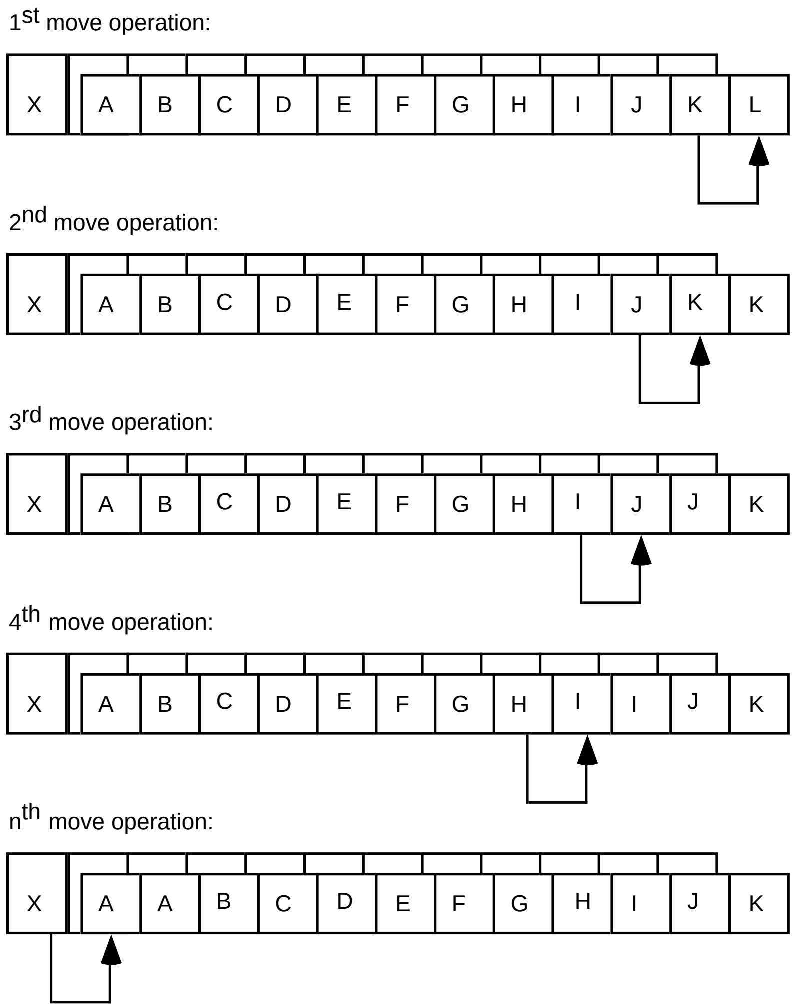

Figure 15.2 Correct Way to Move Data With a Block Move Operation

The end result is that X gets replicated throughout the string. The move instruction copies the source operand into the memory location which will become the source operand for the very next move operation, which causes the replication.

If you really want to move one array into another when they overlap, you should move each element of the source string to the destination string starting at the end of the two strings as shown in Figure 15.2.

Setting the direction flag and pointing si and di at the end of the strings will allow you to (correctly) move one string to another when the two strings overlap and the source string begins at a lower address than the destination string. If the two strings overlap and the source string begins at a higher address than the destination string, then clear the direction flag and point si and di at the beginning of the two strings.

If the two strings do not overlap, then you can use either technique to move the strings around in memory. Generally, operating with the direction flag clear is the easiest, so that makes the most sense in this case.

You shouldn't use the movs instruction to fill an array with a single byte, word, or double word value. Another string instruction, stos, is much better suited for this purpose. However, for arrays whose elements are larger than four bytes, you can use the movs instruction to initialize the entire array to the content of the first element. See the questions for additional information.

15.1.5 The CMPS Instruction

The cmps instruction compares two strings. The CPU compares the string referenced by es:di to the string pointed at by ds:si. Cx contains the length of the two strings (when using the rep prefix). Like the movs instruction, the MASM assembler allows several different forms of this instruction:

```
{REPE}    CMPSB
{REPE}    CMPSW
{REPE}    CMPSD                  ;Available only on 80386 and later
{REPE}    CMPS     dest, source
{REPNE}   CMPSB
{REPNE}   CMPSW
{REPNE}   CMPSD                  ;Available only on 80386 and later
{REPNE}   CMPS     dest, source
```

Like the movs instruction, the operands present in the operand field of the cmps instruction determine the size of the operands. You specify the actual operand addresses in the si and di registers.

Without a repeat prefix, the cmps instruction subtracts the value at location es:di from the value at ds:si and updates the flags. Other than updating the flags, the CPU doesn't use the difference produced by this subtraction. After comparing the two locations, cmps increments or decrements the si and di registers by one, two, or four (for cmpsb/cmpsw/cmpsd, respectively). Cmps increments the si and di registers if the direction flag is clear and decrements them otherwise.

Of course, you will not tap the real power of the cmps instruction using it to compare single bytes or words in memory. This instruction shines when you use it to compare whole strings. With cmps, you can compare consecutive elements in a string until you find a match or until consecutive elements do not match.

To compare two strings to see if they are equal or not equal, you must compare corresponding elements in a string until they don't match. Consider the following strings:

“String1”

“String1”

The only way to determine that these two strings are equal is to compare each character in the first string to the corresponding character in the second. After all, the second string could have been “String2” which definitely is not equal to “String1”. Of course, once you encounter a character in the destination string which doesn't equal the corresponding character in the source string, the comparison can stop. You needn't compare any other characters in the two strings.

The repe prefix accomplishes this operation. It will compare successive elements in a string as long as they are equal and cx is greater than zero. We could compare the two strings above using the following 80x86 assembly language code:

```
; Assume both strings are in the same segment and ES and DS
; both point at this segment.

          cld
          lea      si, AdrsString1
          lea      di, AdrsString2
          mov      cx, 7
    repe  cmpsb
```

After the execution of the cmpsb instruction, you can test the flags using the standard conditional jump instructions. This lets you check for equality, inequality, less than, greater than, etc.

Character strings are usually compared using *lexicographical ordering*. In lexicographical ordering, the least significant element of a string carries the most weight. This is in direct contrast to standard integer comparisons where the most significant portion of the number carries the most weight. Furthermore, the length of a string affects the comparison only if the two strings are identical up to the length of the shorter string. For example, “Zebra” is less than “Zebras”, because it is the shorter of the two strings, however, “Zebra” is greater than “AAAAAAAAAAH!” even though it is shorter. Lexicographical comparisons compare corresponding elements until encountering a character which doesn’t match, or until encountering the end of the shorter string. If a pair of corresponding characters do not match, then this algorithm compares the two strings based on that single character. If the two strings match up to the length of the shorter string, we must compare their length. The two strings are equal if and only if their lengths are equal and each corresponding pair of characters in the two strings is identical. Lexicographical ordering is the standard alphabetical ordering you’ve grown up with.

For character strings, use the cmps instruction in the following manner:

- The direction flag must be cleared before comparing the strings.
- Use the cmpsb instruction to compare the strings on a byte by byte basis. Even if the strings contain an even number of characters, you cannot use the cmpsw instruction. It does not compare strings in lexicographical order.
- The cx register must be loaded with the length of the smaller string.
- Use the repe prefix.
- The ds:si and es:di registers must point at the very first character in the two strings you want to compare.

After the execution of the cmps instruction, if the two strings were equal, their lengths must be compared in order to finish the comparison. The following code compares a couple of character strings:

```
                 lea      si, source
                 lea      di, dest
                 mov      cx, lengthSource
                 mov      ax, lengthDest
                 cmp      cx, ax
                 ja       NoSwap
                 xchg     ax, cx
NoSwap: repe     cmpsb
                 jne      NotEqual
                 mov      ax, lengthSource
                 cmp      ax, lengthDest
NotEqual:
```

If you’re using bytes to hold the string lengths, you should adjust this code appropriately.

You can also use the cmps instruction to compare multi-word integer values (that is, extended precision integer values). Because of the amount of setup required for a string comparison, this isn’t practical for integer values less than three or four words in length, but for large integer values, it’s an excellent way to compare such values. Unlike character strings, we cannot compare integer strings using a lexicographical ordering. When comparing strings, we compare the characters from the least significant byte to the most significant byte. When comparing integers, we must compare the values from the most significant byte (or word / double word) down to the least significant byte, word or double word. So, to compare two eight-word (128-bit) integer values, use the following code on the 80286:

```
                 std
                 lea      si, SourceInteger+14
                 lea      di, DestInteger+14
                 mov      cx, 8
        repe     cmpsw
```

This code compares the integers from their most significant word down to the least significant word. The cmpsw instruction finishes when the two values are unequal or upon decrementing cx to zero (implying that the two values are equal). Once again, the flags provide the result of the comparison.

The repne prefix will instruct the cmps instruction to compare successive string elements as long as they do not match. The 80x86 flags are of little use after the execution of this instruction. Either the cx register is zero (in which case the two strings are totally different), or it contains the number of elements compared in the two strings until a match. While this form of the cmps instruction isn't particularly useful for comparing strings, it is useful for locating the first pair of matching items in a couple of byte or word arrays. In general, though, you'll rarely use the repne prefix with cmps.

One last thing to keep in mind with using the cmps instruction – the value in the cx register determines the number of elements to process, not the number of bytes. Therefore, when using cmpsw, cx specifies the number of words to compare. This, of course, is twice the number of bytes to compare.

15.1.6 The SCAS Instruction

The cmps instruction compares two strings against one another. You cannot use it to search for a particular element within a string. For example, you could not use the cmps instruction to quickly scan for a zero throughout some other string. You can use the scas (scan string) instruction for this task.

Unlike the movs and cmps instructions, the scas instruction only requires a destination string (es:di) rather than both a source and destination string. The source operand is the value in the al (scasb), ax (scasw), or eax (scasd) register.

The scas instruction, by itself, compares the value in the accumulator (al, ax, or eax) against the value pointed at by es:di and then increments (or decrements) di by one, two, or four. The CPU sets the flags according to the result of the comparison. While this might be useful on occasion, scas is a lot more useful when using the repe and repne prefixes.

When the repe prefix (repeat while equal) is present, scas scans the string searching for an element which does not match the value in the accumulator. When using the repne prefix (repeat while not equal), scas scans the string searching for the first string element which is equal to the value in the accumulator.

You're probably wondering "why do these prefixes do exactly the opposite of what they ought to do?" The paragraphs above haven't quite phrased the operation of the scas instruction properly. When using the repe prefix with scas, the 80x86 scans through the string while the value in the accumulator is equal to the string operand. This is equivalent to searching through the string for the first element which does not match the value in the accumulator. The scas instruction with repne scans through the string while the accumulator is not equal to the string operand. Of course, this form searches for the first value in the string which matches the value in the accumulator register. The scas instruction takes the following forms:

```
{REPE}    SCASB
{REPE}    SCASW
{REPE}    SCASD        ;Available only on 80386 and later processors
{REPE}    SCAS      dest
{REPNE}   SCASB
{REPNE}   SCASW
{REPNE}   SCASD        ;Available only on 80386 and later processors
{REPNE}   SCAS      dest
```

Like the cmps and movs instructions, the value in the cx register specifies the number of elements to process, not bytes, when using a repeat prefix.

15.1.7 The STOS Instruction

The stos instruction stores the value in the accumulator at the location specified by es:di. After storing the value, the CPU increments or decrements di depending upon the state of the direction flag. Although the stos instruction has many uses, its primary use is to initialize arrays and strings to a constant value. For example, if you have a 256-byte array you want to clear out with zeros, use the following code:

```
; Presumably, the ES register already points at the segment
; containing DestString

                cld
                lea     di, DestString
                mov     cx, 128                 ;256 bytes is 128 words.
                xor     ax, ax                  ;AX := 0
        rep     stosw
```

This code writes 128 words rather than 256 bytes because a single stosw operation is faster than two stosb operations. On an 80386 or later this code could have written 64 double words to accomplish the same thing even faster.

The stos instruction takes four forms. They are

```
        {REP}   STOSB
        {REP}   STOSW
        {REP}   STOSD
        {REP}   STOS    dest
```

The stosb instruction stores the value in the al register into the specified memory location(s), the stosw instruction stores the ax register into the specified memory location(s) and the stosd instruction stores eax into the specified location(s). The stos instruction is either an stosb, stosw, or stosd instruction depending upon the size of the specified operand.

Keep in mind that the stos instruction is useful only for initializing a byte, word, or dword array to a constant value. If you need to initialize an array to different values, you cannot use the stos instruction. You can use movs in such a situation, see the exercises for additional details.

15.1.8 The LODS Instruction

The lods instruction is unique among the string instructions. You will never use a repeat prefix with this instruction. The lods instruction copies the byte or word pointed at by ds:si into the al, ax, or eax register, after which it increments or decrements the si register by one, two, or four. Repeating this instruction via the repeat prefix would serve no purpose whatsoever since the accumulator register will be overwritten each time the lods instruction repeats. At the end of the repeat operation, the accumulator will contain the last value read from memory.

Instead, use the lods instruction to fetch bytes (lodsb), words (lodsw), or double words (lodsd) from memory for further processing. By using the stos instruction, you can synthesize powerful string operations.

Like the stos instruction, the lods instruction takes four forms:

```
        {REP}   LODSB
        {REP}   LODSW
        {REP}   LODSD                   ;Available only on 80386 and later
        {REP}   LODS    dest
```

As mentioned earlier, you'll rarely, if ever, use the rep prefixes with these instructions[3]. The 80x86 increments or decrements si by one, two, or four depending on the direction flag and whether you're using the lodsb, lodsw, or lodsd instruction.

3. They appear here simply because they are allowed. They're not useful, but they are allowed.

15.1.9 Building Complex String Functions from LODS and STOS

The 80x86 supports only five different string instructions: movs, cmps, scas, lods, and stos[4]. These certainly aren't the only string operations you'll ever want to use. However, you can use the lods and stos instructions to easily generate any particular string operation you like. For example, suppose you wanted a string operation that converts all the upper case characters in a string to lower case. You could use the following code:

```
; Presumably, ES and DS have been set up to point at the same
; segment, the one containing the string to convert.

                 lea      si, String2Convert
                 mov      di, si
                 mov      cx, LengthOfString
Convert2Lower:   lodsb                              ;Get next char in str.
                 cmp      al, 'A'                   ;Is it upper case?
                 jb       NotUpper
                 cmp      al, 'Z'
                 ja       NotUpper
                 or       al, 20h                   ;Convert to lower case.
NotUpper:        stosb                              ;Store into destination.
                 loop     Convert2Lower
```

Assuming you're willing to waste 256 bytes for a table, this conversion operation can be sped up somewhat using the xlat instruction:

```
; Presumably, ES and DS have been set up to point at the same
; segment, the one containing the string to be converted.
                 cld
                 lea      si, String2Convert
                 mov      di, si
                 mov      cx, LengthOfString
                 lea      bx, ConversionTable
Convert2Lower:   lodsb                              ;Get next char in str.
                 xlat                               ;Convert as appropriate.
                 stosb                              ;Store into destination.
                 loop     Convert2Lower
```

The conversion table, of course, would contain the index into the table at each location except at offsets 41h..5Ah. At these locations the conversion table would contain the values 61h..7Ah (i.e., at indexes 'A'..'Z' the table would contain the codes for 'a'..'z').

Since the lods and stos instructions use the accumulator as an intermediary, you can use any accumulator operation to quickly manipulate string elements.

15.1.10 Prefixes and the String Instructions

The string instructions will accept segment prefixes, lock prefixes, and repeat prefixes. In fact, you can specify all three types of instruction prefixes should you so desire. However, due to a bug in the earlier 80x86 chips (pre-80386), you should never use more than a single prefix (repeat, lock, or segment override) on a string instruction unless your code will only run on later processors; a likely event these days. If you absolutely must use two or more prefixes and need to run on an earlier processor, make sure you turn off the interrupts while executing the string instruction.

4. Not counting INS and OUTS which we're ignoring here.

15.2 Character Strings

Since you'll encounter character strings more often than other types of strings, they deserve special attention. The following sections describe character strings and various types of string operations.

15.2.1 Types of Strings

At the most basic level, the 80x86's string instruction only operate upon arrays of characters. However, since most string data types contain an array of characters as a component, the 80x86's string instructions are handy for manipulating that portion of the string.

Probably the biggest difference between a character string and an array of characters is the length attribute. An array of characters contains a fixed number of characters. Never any more, never any less. A character string, however, has a dynamic run-time length, that is, the number of characters contained in the string at some point in the program. Character strings, unlike arrays of characters, have the ability to change their size during execution (within certain limits, of course).

To complicate things even more, there are two generic types of strings: statically allocated strings and dynamically allocated strings. Statically allocated strings are given a fixed, maximum length at program creation time. The length of the string may vary at run-time, but only between zero and this maximum length. Most systems allocate and deallocate dynamically allocated strings in a memory pool when using strings. Such strings may be any length (up to some reasonable maximum value). Accessing such strings is less efficient than accessing statically allocated strings. Furthermore, garbage collection[5] may take additional time. Nevertheless, dynamically allocated strings are much more space efficient than statically allocated strings and, in some instances, accessing dynamically allocated strings is faster as well. Most of the examples in this chapter will use statically allocated strings.

A string with a dynamic length needs some way of keeping track of this length. While there are several possible ways to represent string lengths, the two most popular are length-prefixed strings and zero-terminated strings. A length-prefixed string consists of a single byte or word that contains the length of that string. Immediately following this length value, are the characters that make up the string. Assuming the use of byte prefix lengths, you could define the string "HELLO" as follows:

```
HelloStr        byte    5,"HELLO"
```

Length-prefixed strings are often called Pascal strings since this is the type of string variable supported by most versions of Pascal[6].

Another popular way to specify string lengths is to use zero-terminated strings. A zero-terminated string consists of a string of characters terminated with a zero byte. These types of strings are often called C-strings since they are the type used by the C/C++ programming language. The UCR Standard Library, since it mimics the C standard library, also uses zero-terminated strings.

Pascal strings are much better than C/C++ strings for several reasons. First, computing the length of a Pascal string is trivial. You need only fetch the first byte (or word) of the string and you've got the length of the string. Computing the length of a C/C++ string is considerably less efficient. You must scan the entire string (e.g., using the scasb instruction) for a zero byte. If the C/C++ string is long, this can take a long time. Furthermore, C/C++ strings cannot contain the NULL character. On the other hand, C/C++ strings can be any length, yet require only a single extra byte of overhead. Pascal strings, however,

5. Reclaiming unused storage.
6. At least those versions of Pascal which support strings.

can be no longer than 255 characters when using only a single length byte. For strings longer than 255 bytes, you'll need two bytes to hold the length for a Pascal string. Since most strings are less than 256 characters in length, this isn't much of a disadvantage.

An advantage of zero-terminated strings is that they are easy to use in an assembly language program. This is particularly true of strings that are so long they require multiple source code lines in your assembly language programs. Counting up every character in a string is so tedious that it's not even worth considering. However, you can write a macro which will easily build Pascal strings for you:

```
PString         macro    String
                local    StringLength, StringStart
                byte     StringLength
StringStart     byte     String
StringLength    =        $-StringStart
                endm
                  .
                  .
                  .
                PString    "This string has a length prefix"
```

As long as the string fits entirely on one source line, you can use this macro to generate Pascal style strings.

Common string functions like concatenation, length, substring, index, and others are much easier to write when using length-prefixed strings. So we'll use Pascal strings unless otherwise noted. Furthermore, the UCR Standard library provides a large number of C/C++ string functions, so there is no need to replicate those functions here.

15.2.2 String Assignment

You can easily assign one string to another using the movsb instruction. For example, if you want to assign the length-prefixed string String1 to String2, use the following:

```
; Presumably, ES and DS are set up already

                lea      si, String1
                lea      di, String2
                mov      ch, 0                  ;Extend len to 16 bits.
                mov      cl, String1            ;Get string length.
                inc      cx                     ;Include length byte.
        rep     movsb
```

This code increments cx by one before executing movsb because the length byte contains the length of the string exclusive of the length byte itself.

Generally, string variables can be initialized to constants by using the PString macro described earlier. However, if you need to set a string variable to some constant value, you can write a StrAssign subroutine which assigns the string immediately following the call. The following procedure does exactly that:

```
                include    stdlib.a
                includelib stdlib.lib

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

; String assignment procedure

MainPgm         proc     far
                mov      ax, seg dseg
                mov      ds, ax
                mov      es, ax

                lea      di, ToString
                call     StrAssign
                byte     "This is an example of how the "
```

```
                byte     "StrAssign routine is used",0
                nop
                ExitPgm
MainPgm         endp

StrAssign       proc     near
                push     bp
                mov      bp, sp
                pushf
                push     ds
                push     si
                push     di
                push     cx
                push     ax
                push     di          ;Save again for use later.
                push     es
                cld

; Get the address of the source string

                mov      ax, cs
                mov      es, ax
                mov      di, 2[bp]   ;Get return address.
                mov      cx, 0ffffh  ;Scan for as long as it takes.
                mov      al, 0       ;Scan for a zero.
       repne    scasb                ;Compute the length of string.
                neg      cx          ;Convert length to a positive #.
                dec      cx          ;Because we started with -1, not 0.
                dec      cx          ;skip zero terminating byte.

; Now copy the strings

                pop      es          ;Get destination segment.
                pop      di          ;Get destination address.
                mov      al, cl      ;Store length byte.
                stosb

; Now copy the source string.

                mov      ax, cs
                mov      ds, ax
                mov      si, 2[bp]
       rep      movsb

; Update the return address and leave:

                inc      si          ;Skip over zero byte.
                mov      2[bp], si

                pop      ax
                pop      cx
                pop      di
                pop      si
                pop      ds
                popf
                pop      bp
                ret
StrAssign       endp

cseg            ends

dseg            segment para public 'data'
ToString        byte     255 dup (0)
dseg            ends

sseg            segment para stack 'stack'
                word     256 dup (?)
sseg            ends
                end      MainPgm
```

This code uses the scas instruction to determine the length of the string immediately following the call instruction. Once the code determines the length, it stores this length into the first byte of the destination string and then copies the text following the call to the string variable. After copying the string, this code adjusts the return address so that it points just beyond the zero terminating byte. Then the procedure returns control to the caller.

Of course, this string assignment procedure isn't very efficient, but it's very easy to use. Setting up es:di is all that you need to do to use this procedure. If you need fast string assignment, simply use the movs instruction as follows:

```
; Presumably, DS and ES have already been set up.

                 lea      si, SourceString
                 lea      di, DestString
                 mov      cx, LengthSource
        rep      movsb
                   .
                   .
                   .
SourceString     byte     LengthSource-1
                 byte     "This is an example of how the "
                 byte     "StrAssign routine is used"
LengthSource     =        $-SourceString

DestString       byte     256 dup (?)
```

Using in-line instructions requires considerably more setup (and typing!), but it is much faster than the StrAssign procedure. If you don't like the typing, you can always write a macro to do the string assignment for you.

15.2.3 String Comparison

Comparing two character strings was already beaten to death in the section on the cmps instruction. Other than providing some concrete examples, there is no reason to consider this subject any further.

Note: all the following examples assume that es and ds are pointing at the proper segments containing the destination and source strings.

Comparing Str1 to Str2:

```
                 lea      si, Str1
                 lea      di, Str2

; Get the minimum length of the two strings.

                 mov      al, Str1
                 mov      cl, al
                 cmp      al, Str2
                 jb       CmpStrs
                 mov      cl, Str2

; Compare the two strings.

CmpStrs:         mov      ch, 0
                 cld
        repe     cmpsb
                 jne      StrsNotEqual

; If CMPS thinks they're equal, compare their lengths
; just to be sure.

                 cmp      al, Str2
StrsNotEqual:
```

At label StrsNotEqual, the flags will contain all the pertinent information about the ranking of these two strings. You can use the conditional jump instructions to test the result of this comparison.

15.3 Character String Functions

Most high level languages, like Pascal, BASIC, "C", and PL/I, provide several string functions and procedures (either built into the language or as part of a standard library). Other than the five string operations provided above, the 80x86 doesn't support any string functions. Therefore, if you need a particular string function, you'll have to write it yourself. The following sections describe many of the more popular string functions and how to implement them in assembly language.

15.3.1 Substr

The Substr (substring) function copies a portion of one string to another. In a high level language, this function usually takes the form:

```
DestStr := Substr(SrcStr,Index,Length);
```

where:

- DestStr is the name of the string variable where you want to store the substring,
- SrcStr is the name of the source string (from which the substring is to be taken),
- Index is the starting character position within the string (1..length(SrcStr)), and
- Length is the length of the substring you want to copy into DestStr.

The following examples show how Substr works.

```
SrcStr := 'This is an example of a string';
DestStr := Substr(SrcStr,11,7);
write(DestStr);
```

This prints 'example'. The index value is eleven, so, the Substr function will begin copying data starting at the eleventh character in the string. The eleventh character is the 'e' in 'example'. The length of the string is seven.

This invocation copies the seven characters 'example' to DestStr.

```
SrcStr := 'This is an example of a string';
DestStr := Substr(SrcStr,1,10);
write(DestStr);
```

This prints 'This is an'. Since the index is one, this occurrence of the Substr function starts copying 10 characters starting with the first character in the string.

```
SrcStr := 'This is an example of a string';
DestStr := Substr(SrcStr,20,11);
write(DestStr);
```

This prints 'of a string'. This call to Substr extracts the last eleven characters in the string.

What happens if the index and length values are out of bounds? For example, what happens if Index is zero or is greater than the length of the string? What happens if Index is fine, but the sum of Index and Length is greater than the length of the source string? You can handle these abnormal situations in one of three ways: (1)ignore the possibility of error; (2)abort the program with a run-time error; (3)process some reasonable number of characters in response to the request.

The first solution operates under the assumption that the caller never makes a mistake computing the values for the parameters to the Substr function. It blindly assumes that the values passed to the Substr function are correct and processes the string based on that assumption. This can produce some bizarre effects. Consider the following examples, which use length-prefixed strings:

```
SourceStr :='1234567890ABCDEFGHIJKLMNOPQRSTUVWXYZ';
DestStr := Substr(SourceStr,0,5);
Write('DestStr');
```

prints '$1234'. The reason, of course, is that SourceStr is a length-prefixed string. Therefore the length, 36, appears at offset zero within the string. If Substr uses the illegal index of zero then the length of the string will be returned as the first character. In this particular case, the length of the string, 36, just happened to correspond to the ASCII code for the '$' character.

The situation is considerably worse if the value specified for Index is negative or is greater than the length of the string. In such a case, the Substr function would be returning a substring containing characters appearing before or after the source string. This is not a reasonable result.

Despite the problems with ignoring the possibility of error in the Substr function, there is one big advantage to processing substrings in this manner: the resulting Substr code is more efficient if it doesn't have to perform any run-time checking on the data. If you know that the index and length values are always within an acceptable range, then there is no need to do this checking within Substr function. If you can guarantee that an error will not occur, your programs will run (somewhat) faster by eliminating the run-time check.

Since most programs are rarely error-free, you're taking a big gamble if you assume that all calls to the Substr routine are passing reasonable values. Therefore, some sort of run-time check is often necessary to catch errors in your program. An error occurs under the following conditions:

- The index parameter (Index) is less than one.
- Index is greater than the length of the string.
- The Substr length parameter (Length) is greater than the length of the string.
- The sum of Index and Length is greater than the length of the string.

An alternative to ignoring any of these errors is to abort with an error message. This is probably fine during the program development phase, but once your program is in the hands of users it could be a real disaster. Your customers wouldn't be very happy if they'd spent all day entering data into a program and it aborted, causing them to lose the data they've entered. An alternative to aborting when an error occurs is to have the Substr function return an error condition. Then leave it up to the calling code to determine if an error has occurred. This technique works well with the third alternative to handling errors: processing the substring as best you can.

The third alternative, handling the error as best you can, is probably the best alternative. Handle the error conditions in the following manner:

- The index parameter (Index) is less than one. There are two ways to handle this error condition. One way is to automatically set the Index parameter to one and return the substring beginning with the first character of the source string. The other alternative is to return the *empty string*, a string of length zero, as the substring. Variations on this theme are also possible. You might return the substring beginning with the first character if the index is zero and an empty string if the index is negative. Another alternative is to use unsigned numbers. Then you've only got to worry about the case where Index is zero. A negative number, should the calling code accidentally generate one, would look like a large positive number.

- The index is greater than the length of the string. If this is the case, then the Substr function should return an empty string. Intuitively, this is the proper response in this situation.
- The Substr length parameter (Length) is greater than the length of the string. -or-
- The sum of Index and Length is greater than the length of the string. Points three and four are the same problem, the length of the desired substring extends beyond the end of the source string. In this event, Substr should return the substring consisting of those characters starting at Index through the end of the source string.

The following code for the Substr function expects four parameters: the addresses of the source and destination strings, the starting index, and the length of the desired substring. Substr expects the parameters in the following registers:

ds:si-	The address of the source string.
es:di-	The address of the destination string.
ch-	The starting index.
cl-	The length of the substring.

Substr returns the following values:

- The substring, at location es:di.
- Substr clears the carry flag if there were no errors. Substr sets the carry flag if there was an error.
- Substr preserves all the registers.

If an error occurs, then the calling code must examine the values in si, di and cx to determine the exact cause of the error (if this is necessary). In the event of an error, the Substr function returns the following substrings:

- If the Index parameter (ch) is zero, Substr uses one instead.
- The Index and Length parameters are both unsigned byte values, therefore they are never negative.
- If the Index parameter is greater than the length of the source string, Substr returns an empty string.
- If the sum of the Index and Length parameters is greater than the length of the source string, Substr returns only those characters from Index through the end of the source string. The following code realizes the substring function.

```
; Substring function.
;
; HLL form:
;
;procedure substring(var Src:string;
;                          Index, Length:integer;
;                          var Dest:string);
;
; Src- Address of a source string.
; Index- Index into the source string.
; Length- Length of the substring to extract.
; Dest- Address of a destination string.
;
; Copies the source string from address [Src+index] of length
; Length to the destination string.
;
; If an error occurs, the carry flag is returned set, otherwise
; clear.
;
; Parameters are passed as follows:
;
; DS:SI- Source string address.
; ES:DI- Destination string address.
```

```
; CH- Index into source string.
; CL- Length of source string.
;
; Note: the strings pointed at by the SI and DI registers are
; length-prefixed strings. That is, the first byte of each
; string contains the length of that string.

Substring       proc    near
                push    ax
                push    cx
                push    di
                push    si
                clc                     ;Assume no error.
                pushf                   ;Save direction flag status.

; Check the validity of the parameters.

                cmp     ch, [si]        ;Is index beyond the length of
                ja      ReturnEmpty     ; the source string?
                mov     al, ch          ;See if the sum of index and
                dec     al              ; length is beyond the end of the
                add     al, cl          ; string.
                jc      TooLong         ;Error if > 255.
                cmp     al, [si]        ;Beyond the length of the source?
                jbe     OkaySoFar

; If the substring isn't completely contained within the source
; string, truncate it:

TooLong:        popf
                stc                     ;Return an error flag.
                pushf
                mov     al, [si]        ;Get maximum length.
                sub     al, ch          ;Subtract index value.
                inc     al              ;Adjust as appropriate.
                mov     cl, al          ;Save as new length.

OkaySoFar:      mov     es:[di], cl     ;Save destination string length.
                inc     di
                mov     al, ch          ;Get index into source.
                mov     ch, 0           ;Zero extend length value into CX.
                mov     ah, 0           ;Zero extend index into AX.
                add     si, ax          ;Compute address of substring.
                cld
        rep     movsb                   ;Copy the substring.

                popf
SubStrDone:     pop     si
                pop     di
                pop     cx
                pop     ax
                ret

; Return an empty string here:

ReturnEmpty:    mov     byte ptr es:[di], 0
                popf
                stc
                jmp     SubStrDone
SubString       endp
```

15.3.2 Index

The Index string function searches for the first occurrence of one string within another and returns the offset to that occurrence. Consider the following HLL form:

```
SourceStr := 'Hello world';
TestStr := 'world';
I := INDEX(SourceStr, TestStr);
```

The Index function scans through the source string looking for the first occurrence of the test string. If found, it returns the index into the source string where the test string begins. In the example above, the Index function would return seven since the substring 'world' starts at the seventh character position in the source string.

The only possible error occurs if Index cannot find the test string in the source string. In such a situation, most implementations return zero. Our version will do likewise. The Index function which follows operates in the following fashion:

1) It compares the length of the test string to the length of the source string. If the test string is longer, Index immediately returns zero since there is no way the test string will be found in the source string in this situation.

2) The index function operates as follows:

```
i := 1;
while (i < (length(source)-length(test)) and
           test <> substr(source, i, length(test)) do
                     i := i+1;
```

When this loop terminates, if (i < length(source)-length(test)) then it contains the index into source where test begins. Otherwise test is not a substring of source. Using the previous example, this loop compares test to source in the following manner:

```
i=1
test:           world                            No match
source:         Hello world

i=2
test:            world                           No match
source:         Hello world

i=3
test:             world                          No match
source:         Hello world

i=4
test:              world                         No match
source:         Hello world

i=5
test:               world                        No match
source:         Hello world

i=6
test:                world                       No match
source:         Hello world

i=7
test:                 world                      Match
source:         Hello world
```

There are (algorithmically) better ways to do this comparison[7], however, the algorithm above lends itself to the use of 80x86 string instructions and is very easy to understand. Index's code follows:

```
; INDEX- computes the offset of one string within another.
;
; On entry:
;
```

7. The interested reader should look up the Knuth-Morris-Pratt algorithm in "Data Structure Techniques" by Thomas A. Standish. The Boyer-Moore algorithm is another fast string search routine, although somewhat more complex.

```
; ES:DI-                   Points at the test string that INDEX will search for
;                          in the source string.
; DS:SI-                   Points at the source string which (presumably)
;                           contains the string INDEX is searching for.
;
; On exit:
;
; AX-                      Contains the offset into the source string where the
;                          test string was found.

INDEX           proc     near
                push     si
                push     di
                push     bx
                push     cx
                pushf                  ;Save direction flag value.
                cld

                mov      al, es:[di]  ;Get the length of the test string.
                cmp      al, [si]     ;See if it is longer than the length
                ja       NotThere     ; of the source string.

; Compute the index of the last character we need to compare the
; test string against in the source string.

                mov      al, es:[di]  ;Length of test string.
                mov      cl, al       ;Save for later.
                mov      ch, 0
                sub      al, [si]     ;Length of source string.
                mov      bl, al       ;# of times to repeat loop.
                inc      di           ;Skip over length byte.
                xor      ax, ax       ;Init index to zero.
CmpLoop:        inc      ax           ;Bump index by one.
                inc      si           ;Move on to the next char in source.
                push     si           ;Save string pointers and the
                push     di           ; length of the test string.
                push     cx
        rep     cmpsb                 ;Compare the strings.
                pop      cx           ;Restore string pointers
                pop      di           ; and length.
                pop      si
                je       Foundindex   ;If we found the substring.
                dec      bl
                jnz      CmpLoop      ;Try next entry in source string.

; If we fall down here, the test string doesn't appear inside the
; source string.

NotThere:       xor      ax, ax       ;Return INDEX = 0

; If the substring was found in the loop above, remove the
; garbage left on the stack

FoundIndex:     popf
                pop      cx
                pop      bx
                pop      di
                pop      si
                ret
INDEX           endp
```

15.3.3 Repeat

The Repeat string function expects three parameters– the address of a string, a length, and a character. It constructs a string of the specified length containing “length” copies of the specified character. For example, Repeat(STR,5,'*') stores the string '*****' into the STR string variable. This is a very easy string function to write, thanks to the stosb instruction:

```
; REPEAT-                 Constructs a string of length CX where each element
;                         is initialized to the character passed in AL.
;
; On entry:
;
; ES:DI-                  Points at the string to be constructed.
; CX-                     Contains the length of the string.
; AL-                     Contains the character with which each element of
;                         the string is to be initialized.

REPEAT          proc      near
                push      di
                push      ax
                push      cx
                pushf                  ;Save direction flag value.
                cld
                mov       es:[di], cl  ;Save string length.
                mov       ch, 0        ;Just in case.
                inc       di           ;Start string at next location.
       rep      stosb
                popf
                pop       cx
                pop       ax
                pop       di
                ret
REPEAT          endp
```

15.3.4 Insert

The Insert string function inserts one string into another. It expects three parameters, a source string, a destination string, and an index. Insert inserts the source string into the destination string starting at the offset specified by the index parameter. HLLs usually call the Insert procedure as follows:

```
source := ' there';
dest := 'Hello world';
INSERT(source,dest,6);
```

The call to Insert above would change source to contain the string 'Hello there world'. It does this by inserting the string ' there' before the sixth character in 'Hello world'.

The insert procedure using the following algorithm:

Insert(Src,dest,index);

1) Move the characters from location dest+index through the end of the destination string length (Src) bytes up in memory.
2) Copy the characters from the Src string to location dest+index.
3) Adjust the length of the destination string so that it is the sum of the destination and source lengths. The following code implements this algorithm:

```
; INSERT- Inserts one string into another.
;
; On entry:
;
; DS:SI Points at the source string to be inserted
;
; ES:DI Points at the destination string into which the source
; string will be inserted.
;
; DX Contains the offset into the destination string where the
```

```
; source string is to be inserted.
;
;
; All registers are preserved.
;
; Error condition-
;
; If the length of the newly created string is greater than 255,
; the insert operation will not be performed and the carry flag
; will be returned set.
;
; If the index is greater than the length of the destination
; string,
; then the source string will be appended to the end of the destin- ; ation
string.

INSERT          proc     near
                push     si
                push     di
                push     dx
                push     cx
                push     bx
                push     ax
                clc                      ;Assume no error.
                pushf
                mov      dh, 0           ;Just to be safe.

; First, see if the new string will be too long.

                mov      ch, 0
                mov      ah, ch
                mov      bh, ch
                mov      al, es:[di]     ;AX = length of dest string.
                mov      cl, [si]        ;CX = length of source string.
                mov      bl, al          ;BX = length of new string.
                add      bl, cl
                jc       TooLong         ;Abort if too long.
                mov      es:[di], bl     ;Update length.

; See if the index value is too large:

                cmp      dl, al
                jbe      IndexIsOK
                mov      dl, al
IndexIsOK:

; Now, make room for the string that’s about to be inserted.

                push     si              ;Save for later.
                push     cx

                mov      si, di          ;Point SI at the end of current
                add      si, ax          ; destination string.
                add      di, bx          ;Point DI at the end of new str.
                std
        rep     movsb                    ;Open up space for new string.

; Now, copy the source string into the space opened up.

                pop      cx
                pop      si
                add      si, cx          ;Point at end of source string.
        rep     movsb
                jmp      INSERTDone

TooLong:        popf
                stc
                pushf

INSERTDone:     popf
```

```
                pop     ax
                pop     bx
                pop     cx
                pop     dx
                pop     di
                pop     si
                ret
INSERT          endp
```

15.3.5 Delete

The Delete string removes characters from a string. It expects three parameters – the address of a string, an index into that string, and the number of characters to remove from that string. A HLL call to Delete usually takes the form:

```
Delete(Str,index,length);
```

For example,

```
Str := 'Hello there world';
Delete(str,7,6);
```

This call to Delete will leave str containing 'Hello world'. The algorithm for the delete operation is the following:

1) Subtract the length parameter value from the length of the destination string and update the length of the destination string with this new value.

2) Copy any characters following the deleted substring over the top of the deleted substring.

There are a couple of errors that may occur when using the delete procedure. The index value could be zero or larger than the size of the specified string. In this case, the Delete procedure shouldn't do anything to the string. If the sum of the index and length parameters is greater than the length of the string, then the Delete procedure should delete all the characters to the end of the string. The following code implements the Delete procedure:

```
; DELETE - removes some substring from a string.
;
; On entry:
;
; DS:SI                    Points at the source string.
; DX                       Index into the string of the start of the substring
;                          to delete.
; CX                       Length of the substring to be deleted.
;
; Error conditions-
;
; If DX is greater than the length of the string, then the
; operation is aborted.
;
; If DX+CX is greater than the length of the string, DELETE only
; deletes those characters from DX through the end of the string.

DELETE          proc    near
                push    es
                push    si
                push    di
                push    ax
                push    cx
                push    dx
                pushf                   ;Save direction flag.
                mov     ax, ds          ;Source and destination strings
                mov     es, ax          ; are the same.
                mov     ah, 0
```

```
                mov     dh, ah          ;Just to be safe.
                mov     ch, ah

; See if any error conditions exist.

                mov     al, [si]        ;Get the string length
                cmp     dl, al          ;Is the index too big?
                ja      TooBig
                mov     al, dl          ;Now see if INDEX+LENGTH
                add     al, cl          ;is too large
                jc      Truncate
                cmp     al, [si]
                jbe     LengthIsOK

; If the substring is too big, truncate it to fit.

Truncate:       mov     cl, [si]        ;Compute maximum length
                sub     cl, dl
                inc     cl

; Compute the length of the new string.

LengthIsOK:     mov     al, [si]
                sub     al, cl
                mov     [si], al

; Okay, now delete the specified substring.

                add     si, dx          ;Compute address of the substring
                mov     di, si          ; to be deleted, and the address of
                add     di, cx          ; the first character following it.
                cld
        rep     movsb                   ;Delete the string.

TooBig:         popf
                pop     dx
                pop     cx
                pop     ax
                pop     di
                pop     si
                pop     es
                ret
DELETE          endp
```

15.3.6 Concatenation

The concatenation operation takes two strings and appends one to the end of the other. For example, Concat('Hello ','world') produces the string 'Hello world'. Some high level languages treat concatenation as a function call, others as a procedure call. Since in assembly language everything is a procedure call anyway, we'll adopt the procedural syntax. Our Concat procedure will take the following form:

```
Concat(source1,source2,dest);
```

This procedure will copy source1 to dest, then it will concatenate source2 to the end of dest. Concat follows:

```
; Concat-               Copies the string pointed at by SI to the string
;                       rointed at byDI and then concatenates the string;
;                       pointed at by BX to the destination string.
;
; On entry-
;
; DS:SI-                Points at the first source string
; DS:BX-                Points at the second source string
; ES:DI-                Points at the destination string.
```

```
;
; Error condition-
;
; The sum of the lengths of the two strings is greater than 255.
; In this event, the second string will be truncated so that the
; entire string is less than 256 characters in length.

CONCAT          proc     near
                push     si
                push     di
                push     cx
                push     ax
                pushf

; Copy the first string to the destination string:

                mov      al, [si]
                mov      cl, al
                mov      ch, 0
                mov      ah, ch
                add      al, [bx]     ;Compute the sum of the string’s
                adc      ah, 0        ; lengths.
                cmp      ax, 256
                jb       SetNewLength
                mov      ah, [si]     ;Save original string length.
                mov      al, 255      ;Fix string length at 255.
SetNewLength:   mov      es:[di], al  ;Save new string length.
                inc      di           ;Skip over length bytes.
                inc      si
       rep      movsb                 ;Copy source1 to dest string.

; If the sum of the two strings is too long, the second string
; must be truncated.

                mov      cl, [bx]     ;Get length of second string.
                cmp      ax, 256
                jb       LengthsAreOK
                mov      cl, ah       ;Compute truncated length.
                neg      cl           ;CL := 256-Length(Str1).

LengthsAreOK:   lea      si, 1[bx]    ;Point at second string and
;                                     ; skip the string length.
                cld
                rep      movsb                  ;Perform the concatenation.

                popf
                pop      ax
                pop      cx
                pop      di
                pop      si
                ret
CONCAT          endp
```

15.4 String Functions in the UCR Standard Library

The UCR Standard Library for 80x86 Assembly Language Programmers provides a very rich set of string functions you may use. These routines, for the most part, are quite similar to the string functions provided in the C Standard Library. As such, these functions support zero terminated strings rather than the length prefixed strings supported by the functions in the previous sections.

Because there are so many different UCR StdLib string routines and the sources for all these routines are in the public domain (and are present on the companion CD-ROM for this text), the following sections will not discuss the implementation of each routine. Instead, the following sections will concentrate on how to *use* these library routines.

The UCR library often provides several variants of the same routine. Generally a suffix of “l”, “m”, or “ml” appears at the end of the name of these variant routines. The “l” suffix stands for “literal constant”. Routines with the “l” (or “ml”) suffix require two string operands. The first is generally pointed at by es:di and the second immediate follows the call in the code stream.

Most StdLib string routines operate on the specified string (or one of the strings if the function has two operands). The “m” (or “ml”) suffix instructs the string function to allocate storage on the heap (using malloc, hence the “m” suffix) for the new string and store the modified result there rather than changing the source string(s). These routines always return a pointer to the newly created string in the es:di registers. In the event of a memory allocation error (insufficient memory), these routines with the “m” or “ml” suffix return the carry flag set. They return the carry clear if the operation was successful.

15.4.1 StrBDel, StrBDelm

These two routines delete leading spaces from a string. StrBDel removes any leading spaces from the string pointed at by es:di. It actually modifies the source string. StrBDelm makes a copy of the string on the heap with any leading spaces removed. If there are no leading spaces, then the StrBDel routines return the original string without modification. Note that these routines only affect *leading* spaces (those appearing at the beginning of the string). They do not remove trailing spaces and spaces in the middle of the string. See Strtrim if you want to remove trailing spaces. Examples:

```
MyString        byte     “    Hello there, this is my string”,0
MyStrPtr        dword    MyString
                  .
                  .
                  .
                les      di, MyStrPtr
                strbdelm       ;Creates a new string w/o leading spaces,
                jc       error ; pointer to string is in ES:DI on return.
                puts           ;Print the string pointed at by ES:DI.
                free           ;Deallocate storage allocated by strbdelm.
                  .
                  .
                  .
; Note that “MyString” still contains the leading spaces.
; The following printf call will print the string along with
; those leading spaces. “strbdelm” above did not change MyString.

                printf
                byte     “MyString = ‘%s’\n”,0
                dword    MyString
                  .
                  .
                  .
                les      di, MyStrPtr
                strbdel

; Now, we really have removed the leading spaces from “MyString”

                printf
                byte     “MyString = ‘%s’\n”,0
                dword    MyString
                  .
                  .
                  .
```

Output from this code fragment:

```
Hello there, this is my string
MyString = ‘    Hello there, this is my string’
MyString = ‘Hello there, this is my string’
```

15.4.2 Strcat, Strcatl, Strcatm, Strcatml

The strcat(xx) routines perform string concatenation. On entry, es:di points at the first string, and for strcat/strcatm dx:si points at the second string. For strcatl and strcatlm the second string follows the call in the code stream. These routines create a new string by appending the second string to the end of the first. In the case of strcat and strcatl, the second string is directly appended to the end of the first string (es:di) in memory. You must make sure there is sufficient memory at the end of the first string to hold the appended characters. Strcatm and strcatml create a new string on the heap (using malloc) holding the concatenated result. Examples:

```
String1         byte     "Hello ",0
                byte     16 dup (0)              ;Room for concatenation.

String2         byte     "world",0

; The following macro loads ES:DI with the address of the
; specified operand.

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

; The following macro loads DX:SI with the address of the
; specified operand.

ldxi            macro    operand
                mov      dx, seg operand
                mov      si, offset operand
                endm
                  .
                  .
                  .
                lesi     String1
                ldxi     String2
                strcatm                ;Create "Hello world"
                jc       error         ;If insufficient memory.
                print
                byte     "strcatm: ",0
                puts                   ;Print "Hello world"
                putcr
                free                   ;Deallocate string storage.
                  .
                  .
                  .
                lesi     String1       ;Create the string
                strcatml               ; "Hello there"
                jc       error         ;If insufficient memory.
                byte     "there",0
                print
                byte     "strcatml: ",0
                puts                   ;Print "Hello there"
                putcr
                free
                  .
                  .
                  .
                lesi     String1
                ldxi     String2
                strcat                 ;Create "Hello world"
                printf
                byte     "strcat: %s\n",0
                  .
                  .
                  .
; Note: since strcat above has actually modified String1,
; the following call to strcatl appends "there" to the end
; of the string "Hello world".

                lesi     String1
```

```
                  strcatl
                  byte     "there",0
                  printf
                  byte     "strcatl: %s\n",0
                   .
                   .
                   .
```

The code above produces the following output:

```
strcatm: Hello world
strcatml: Hello there
strcat: Hello world
strcatl: Hello world there
```

15.4.3 Strchr

Strchr searches for the first occurrence of a single character within a string. In operation it is quite similar to the scasb instruction. However, you do not have to specify an explicit length when using this function as you would for scasb.

On entry, es:di points at the string you want to search through, al contains the value to search for. On return, the carry flag denotes success (C=1 means the character was *not* present in the string, C=0 means the character was present). If the character was found in the string, cx contains the index into the string where strchr located the character. Note that the first character of the string is at index zero. So strchr will return zero if al matches the first character of the string. If the carry flag is set, then the value in cx has no meaning. Example:

```
; Note that the following string has a period at location
; "HasPeriod+24".

HasPeriod         byte     "This string has a period.",0
                   .
                   .
                   .
                  lesi     HasPeriod    ;See strcat for lesi definition.
                  mov      al, "."      ;Search for a period.
                  strchr
                  jnc      GotPeriod
                  print
                  byte     "No period in string",cr,lf,0
                  jmp      Done

; If we found the period, output the offset into the string:

GotPeriod:        print
                  byte     "Found period at offset ",0
                  mov      ax, cx
                  puti
                  putcr
Done:
```

This code fragment produces the output:

```
Found period at offset 24
```

15.4.4 Strcmp, Strcmpl, Stricmp, Stricmpl

These routines compare strings using a lexicographical ordering. On entry to strcmp or stricmp, es:di points at the first string and dx:si points at the second string. Strcmp compares the first string to the second and returns the result of the comparison in the flags register. Strcmpl operates in a similar fashion, except the second string follows the call in the code stream. The stricmp and stricmpl routines differ from their counterparts in that they ignore case during the comparison. Whereas strcmp would return 'not equal' when comparing "Strcmp" with "strcmp", the stricmp (and stricmpl) routines would return "equal" since the only differences are upper vs. lower case. The “i” in stricmp and stricmpl stands for “ignore case.” Examples:

```
String1         byte     "Hello world", 0
String2         byte     "hello world", 0
String3         byte     "Hello there", 0
                  .
                  .
                  .
                lesi     String1       ;See strcat for lesi definition.
                ldxi     String2       ;See strcat for ldxi definition.
                strcmp
                jae      IsGtrEql
                printf
                byte     "%s is less than %s\n",0
                dword    String1, String2
                jmp      Tryl

IsGtrEql:       printf
                byte     "%s is greater or equal to %s\n",0
                dword    String1, String2

Tryl:           lesi     String2
                strcmpl
                byte     "hi world!",0
                jne      NotEql
                printf
                byte     "Hmmm..., %s is equal to 'hi world!'\n",0
                dword    String2
                jmp      Tryi

NotEql:         printf
                byte     "%s is not equal to 'hi world!'\n",0
                dword    String2

Tryi:           lesi     String1
                ldxi     String2
                stricmp
                jne      BadCmp
                printf
                byte     "Ignoring case, %s equals %s\n",0
                dword    String1, String2
                jmp      Tryil

BadCmp:         printf
                byte     "Wow, stricmp doesn't work! %s <> %s\n",0
                dword    String1, String2

Tryil:          lesi     String2
                stricmpl
                byte     "hELLO THERE",0
                jne      BadCmp2
                print
                byte     "Stricmpl worked",cr,lf,0
                jmp      Done

BadCmp2:        print
                byte     "Stricmp did not work",cr,lf,0

Done:
```

15.4.5 Strcpy, Strcpyl, Strdup, Strdupl

The strcpy and strdup routines copy one string to another. There is no strcpym or strcpyml routines. Strdup and strdupl correspond to those operations. The UCR Standard Library uses the names strdup and strdupl rather than strcpym and strcpyml so it will use the same names as the C standard library.

Strcpy copies the string pointed at by es:di to the memory locations beginning at the address in dx:si. There is no error checking; you must ensure that there is sufficient free space at location dx:si before calling strcpy. Strcpy returns with es:di pointing at the destination string (that is, the original dx:si value). Strcpyl works in a similar fashion, except the source string follows the call.

Strdup duplicates the string which es:di points at and returns a pointer to the new string on the heap. Strdupl works in a similar fashion, except the string follows the call. As usual, the carry flag is set if there is a memory allocation error when using strdup or strdupl. Examples:

```
String1         byte      "Copy this string",0
String2         byte      32 dup (0)
String3         byte      32 dup (0)
StrVar1         dword     0
StrVar2         dword     0
                  .
                  .
                  .
                lesi     String1      ;See strcat for lesi definition.
                ldxi     String2      ;See strcat for ldxi definition.
                strcpy

                ldxi     String3
                strcpyl
                byte     "This string, too!",0

                lesi     String1
                strdup
                jc       error                      ;If insufficient mem.
                mov      word ptr StrVar1, di       ;Save away ptr to
                mov      word ptr StrVar1+2, es     ; string.

                strdupl
                jc       error
                byte     "Also, this string",0
                mov      word ptr StrVar2, di
                mov      word ptr StrVar2+2, es

                printf
                byte     "strcpy: %s\n"
                byte     "strcpyl: %s\n"
                byte     "strdup: %^s\n"
                byte     "strdupl: %^s\n",0
                dword    String2, String3, StrVar1, StrVar2
```

15.4.6 Strdel, Strdelm

Strdel and strdelm delete characters from a string. Strdel deletes the specified characters within the string, strdelm creates a new copy of the source string without the specified characters. On entry, es:di points at the string to manipulate, cx contains the index into the string where the deletion is to start, and ax contains the number of characters to delete from the string. On return, es:di points at the new string (which is on the heap if you call strdelm). For strdelm only, if the carry flag is set on return, there was a memory allocation error. As with all UCR StdLib string routines, the index values for the string are zero-based. That is, zero is the index of the first character in the source string. Example:

```
String1         byte     "Hello there, how are you?",0
                  .
                  .
                  .
                lesi     String1      ;See strcat for lesi definition.
                mov      cx, 5        ;Start at position five (" there")
                mov      ax, 6        ;Delete six characters.
                strdelm               ;Create a new string.
                jc       error        ;If insufficient memory.
                print
                byte     "New string:",0
                puts
```

```
                putcr

                lesi     String1
                mov      ax, 11
                mov      cx, 13
                strdel
                printf
                byte     "Modified string: %s\n",0
                dword    String1
```

This code prints the following:

```
New string: Hello, how are you?
Modified string: Hello there
```

15.4.7 Strins, Strinsl, Strinsm, Strinsml

The strins(xx) functions insert one string within another. For all four routines es:di points at the source string into you want to insert another string. Cx contains the insertion point (0..length of source string). For strins and strinsm, dx:si points at the string you wish to insert. For strinsl and strinsml, the string to insert appears as a literal constant in the code stream. Strins and strinsl insert the second string directly into the string pointed at by es:di. Strinsm and strinsml make a copy of the source string and insert the second string into that copy. They return a pointer to the new string in es:di. If there is a memory allocation error then strinsm/strinsml sets the carry flag on return. For strins and strinsl, the first string must have sufficient storage allocated to hold the new string. Examples:

```
InsertInMe      byte     "Insert >< Here",0
                byte     16 dup (0)
InsertStr       byte     "insert this",0
StrPtr1         dword    0
StrPtr2         dword    0
                  .
                  .
                  .
                lesi     InsertInMe   ;See strcat for lesi definition.
                ldxi     InsertStr    ;See strcat for ldxi definition.
                mov      cx, 8        ;Însert before "<"
                strinsm
                mov      word ptr StrPtr1, di
                mov      word ptr StrPtr1+2, es

                lesi     InsertInMe
                mov      cx, 8
                strinsml
                byte     "insert that",0
                mov      word ptr StrPtr2, di
                mov      word ptr StrPtr2+2, es

                lesi     InsertInMe
                mov      cx, 8
                strinsl
                byte     " ",0        ;Two spaces

                lesi     InsertInMe
                ldxi     InsertStr
                mov      cx, 9        ;In front of first space from above.
                strins

                printf
                byte     "First string: %^s\n"
                byte     "Second string: %^s\n"
                byte     "Third string: %s\n",0
                dword    StrPtr1, StrPtr2, InsertInMe
```

Note that the strins and strinsl operations above both insert strings into the same destination string. The output from the above code is

```
First string: Insert >insert this< here
Second string: Insert >insert that< here
Third string: Insert > insert this < here
```

15.4.8 Strlen

Strlen computes the length of the string pointed at by es:di. It returns the number of characters up to, but not including, the zero terminating byte. It returns this length in the cx register. Example:

```
GetLen          byte    "This string is 33 characters long",0
                  .
                  .
                  .
                lesi    GetLen       ;See strcat for lesi definition.
                strlen
                print
                byte    "The string is ",0
                mov     ax, cx       ;Puti needs the length in AX!
                puti
                print
                byte    " characters long",cr,lf,0
```

15.4.9 Strlwr, Strlwrm, Strupr, Struprm

Strlwr and Strlwrm convert any upper case characters in a string to lower case. Strupr and Struprm convert any lower case characters in a string to upper case. These routines do not affect any other characters present in the string. For all four routines, es:di points at the source string to convert. Strlwr and strupr modify the characters directly in that string. Strlwrm and struprm make a copy of the string to the heap and then convert the characters in the new string. They also return a pointer to this new string in es:di. As usual for UCR StdLib routines, strlwrm and struprm return the carry flag set if there is a memory allocation error. Examples:

```
String1         byte    "This string has lower case.",0
String2         byte    "THIS STRING has Upper Case.",0
StrPtr1         dword   0
StrPtr2         dword   0
                  .
                  .
                  .
                lesi    String1      ;See strcat for lesi definition.
                struprm              ;Convert lower case to upper case.
                jc      error
                mov     word ptr StrPtr1, di
                mov     word ptr StrPtr1+2, es

                lesi    String2
                strlwrm              ;Convert upper case to lower case.
                jc      error
                mov     word ptr StrPtr2, di
                mov     word ptr StrPtr2+2, es

                lesi    String1
                strlwr               ;Convert to lower case, in place.

                lesi    String2
                strupr               ;Convert to upper case, in place.

                printf
                byte    "struprm: %^s\n"
                byte    "strlwrm: %^s\n"
                byte    "strlwr: %s\n"
                byte    "strupr: %s\n",0
                dword   StrPtr1, StrPtr2, String1, String2
```

The above code fragment prints the following:

```
struprm: THIS STRING HAS LOWER CASE
strlwrm: this string has upper case
strlwr: this string has lower case
strupr: THIS STRING HAS UPPER CASE
```

15.4.10 Strrev, Strrevm

These two routines reverse the characters in a string. For example, if you pass strrev the string "ABCDEF" it will convert that string to "FEDCBA". As you'd expect by now, the strrev routine reverse the string whose address you pass in es:di; strrevm first makes a copy of the string on the heap and reverses those characters leaving the original string unchanged. Of course strrevm will return the carry flag set if there was a memory allocation error. Example:

```
Palindrome       byte     "radar",0
NotPaldrm        byte     "x + y - z",0
StrPtr1          dword    0
                   .
                   .
                   .
                 lesi     Palindrome   ;See strcat for lesi definition.
                 strrevm
                 jc       error
                 mov      word ptr StrPtr1, di
                 mov      word ptr StrPtr1+2, es

                 lesi     NotPaldrm
                 strrev

                 printf
                 byte     "First string: %^s\n"
                 byte     "Second string: %s\n",0
                 dword    StrPtr1, NotPaldrm
```

The above code produces the following output:

```
First string: radar
Second string: z - y + x
```

15.4.11 Strset, Strsetm

Strset and strsetm replicate a single character through a string. Their behavior, however, is not quite the same. In particular, while strsetm is quite similar to the *repeat* function (see "Repeat" on page 840), strset is not. Both routines expect a single character value in the al register. They will replicate this character throughout some string. Strsetm also requires a count in the cx register. It creates a string on the heap consisting of cx characters and returns a pointer to this string in es:di (assuming no memory allocation error). Strset, on the other hand, expects you to pass it the address of an existing string in es:di. It will replace each character in that string with the character in al. Note that you do not specify a length when using the strset function, strset uses the length of the existing string. Example:

```
String1          byte     "Hello there",0
                   .
                   .
                   .
                 lesi     String1      ;See strcat for lesi definition.
                 mov      al, '*'
                 strset

                 mov      cx, 8
                 mov      al, '#'
                 strsetm

                 print
```

```
            byte    "String2: ",0
            puts
            printf
            byte    "\nString1: %s\n",0
            dword   String1
```

The above code produces the output:

```
String2: ########
String1: ***********
```

15.4.12 Strspan, Strspanl, Strcspan, Strcspanl

These four routines search through a string for a character which is either in some specified character set (strspan, strspanl) or not a member of some character set (strcspan, strcspanl). These routines appear in the UCR Standard Library only because of their appearance in the C standard library. You should rarely use these routines. The UCR Standard Library includes some other routines for manipulating character sets and performing character matching operations. Nonetheless, these routines are somewhat useful on occasion and are worth a mention here.

These routines expect you to pass them the addresses of two strings: a source string and a character set string. They expect the address of the source string in es:di. Strspan and strcspan want the address of the character set string in dx:si; the character set string follows the call with strspanl and strcspanl. On return, cx contains an index into the string, defined as follows:

strspan, strspanl: Index of first character in source found in the character set.

strcspan, strcspanl: Index of first character in source *not* found in the character set.

If all the characters are in the set (or are not in the set) then cx contains the index into the string of the zero terminating byte.

Example:

```
Source      byte    "ABCDEFG 0123456",0
Set1        byte    "ABCDEFGHIJKLMNOPQRSTUVWXYZ",0
Set2        byte    "0123456789",0
Index1      word    ?
Index2      word    ?
Index3      word    ?
Index4      word    ?
              .
              .
              .
            lesi    Source          ;See strcat for lesi definition.
            ldxi    Set1            ;See strcat for ldxi definition.
            strspan                 ;Search for first ALPHA char.
            mov     Index1, cx      ;Index of first alphabetic char.

            lesi    Source
            lesi    Set2
            strspan                 ;Search for first numeric char.
            mov     Index2, cx

            lesi    Source
            strcspanl
            byte    "ABCDEFGHIJKLMNOPQRSTUVWXYZ",0
            mov     Index3, cx

            lesi    Set2
            strcspnl
            byte    "0123456789",0
            mov     Index4, cx

            printf
            byte    "First alpha char in Source is at offset %d\n"
            byte    "First numeric char is at offset %d\n"
```

```
                byte     "First non-alpha in Source is at offset %d\n"
                byte     "First non-numeric in Set2 is at offset %d\n",0
                dword    Index1, Index2, Index3, Index4
```

This code outputs the following:

```
First alpha char in Source is at offset 0
First numeric char is at offset 8
First non-alpha in Source is at offset 7
First non-numeric in Set2 is at offset 10
```

15.4.13 Strstr, Strstrl

Strstr searches for the first occurrence of one string within another. es:di contains the address of the string in which you want to search for a second string. dx:si contains the address of the second string for the strstr routine; for strstrl the search second string immediately follows the call in the code stream.

On return from strstr or strstrl, the carry flag will be set if the second string is not present in the source string. If the carry flag is clear, then the second string is present in the source string and cx will contain the (zero-based) index where the second string was found. Example:

```
SourceStr       byte     "Search for 'this' in this string",0
SearchStr       byte     "this",0
                  .
                  .
                  .
                lesi     SourceStr    ;See strcat for lesi definition.
                ldxi     SearchStr    ;See strcat for ldxi definition.
                strstr
                jc       NotPresent
                print
                byte     "Found string at offset ",0
                mov      ax, cx       ;Need offset in AX for puti
                puti
                putcr

                lesi     SourceStr
                strstrl
                byte     "for",0
                jc       NotPresent
                print
                byte     "Found 'for' at offset ",0
                mov      ax, cx
                puti
                putcr
NotPresent:
```

The above code prints the following:

```
Found string at offset 12
Found 'for' at offset 7
```

15.4.14 Strtrim, Strtrimm

These two routines are quite similar to strbdel and strbdelm. Rather than removing leading spaces, however, they trim off any trailing spaces from a string. Strtrim trims off any trailing spaces directly on the specified string in memory. Strtrimm first copies the source string and then trims and space off the copy. Both routines expect you to pass the address of the source string in es:di. Strtrimm returns a pointer to the new string (if it could allocate it) in es:di. It also returns the carry set or clear to denote error/no error. Example:

```
String1         byte     "Spaces at the end      ",0
String2         byte     "    Spaces on both sides     ",0
StrPtr1         dword    0
StrPtr2         dword    0
                  .
                  .
                  .

; TrimSpcs trims the spaces off both ends of a string.
; Note that it is a little more efficient to perform the
; strbdel first, then the strtrim. This routine creates
; the new string on the heap and returns a pointer to this
; string in ES:DI.

TrimSpcs        proc
                strbdelm
                jc       BadAlloc     ;Just return if error.
                strtrim
                clc
BadAlloc:       ret
TrimSpcs        endp
                          .
                          .
                          .
                lesi     String1      ;See strcat for lesi definition.
                strtrimm
                jc       error
                mov      word ptr StrPtr1, di
                mov      word ptr StrPtr1+2, es

                lesi     String2
                call     TrimSpcs
                jc       error
                mov      word ptr StrPtr2, di
                mov      word ptr StrPtr2+2, es

                printf
                byte     "First string: '%s'\n"
                byte     "Second string: '%s'\n",0
                dword    StrPtr1, StrPtr2
```

This code fragment outputs the following:

```
First string: 'Spaces at the end'
Second string: 'Spaces on both sides'
```

15.4.15 Other String Routines in the UCR Standard Library

In addition to the "strxxx" routines listed in this section, there are many additional string routines available in the UCR Standard Library. Routines to convert from numeric types (integer, hex, real, etc.) to a string or vice versa, pattern matching and character set routines, and many other conversion and string utilities. The routines described in this chapter are those whose definitions appear in the "strings.a" header file and are specifically targeted towards generic string manipulation. For more details on the other string routines, consult the UCR Standard Library reference section in the appendices.

15.5 The Character Set Routines in the UCR Standard Library

The UCR Standard Library provides an extensive collection of character set routines. These routines let you create sets, clear sets (set them to the empty set), add and remove one or more items, test for set membership, copy sets, compute the union, intersection, or difference, and extract items from a set. Although intended to manipulate sets of characters, you can use the StdLib character set routines to manipulate any set with 256 or fewer possible items.

The first unusual thing to note about the StdLib's sets is their storage format. A 256-bit array would normally consumes 32 consecutive bytes. For performance reasons, the UCR Standard Library's set format packs eight separate sets into 272 bytes (256 bytes for the eight sets plus 16 bytes overhead). To declare set variables in your data segment you should use the set macro. This macro takes the form:

```
                set     SetName1, SetName2, ..., SetName8
```

SetName1..SetName8 represent the names of up to eight set variables. You may have fewer than eight names in the operand field, but doing so will waste some bits in the set array.

The CreateSets routine provides another mechanism for creating set variables. Unlike the set macro, which you would use to create set variables in your data segment, the CreateSets routine allocates storage for up to eight sets dynamically at run time. It returns a pointer to the first set variable in es:di. The remaining seven sets follow at locations es:di+1, es:di+2, ..., es:di+7. A typical program that allocates set variables dynamically might use the following code:

```
Set0            dword   ?
Set1            dword   ?
Set2            dword   ?
Set3            dword   ?
Set4            dword   ?
Set5            dword   ?
Set6            dword   ?
Set7            dword   ?
                  .
                  .
                  .
                CreateSets
                mov     word ptr Set0+2, es
                mov     word ptr Set1+2, es
                mov     word ptr Set2+2, es
                mov     word ptr Set3+2, es
                mov     word ptr Set4+2, es
                mov     word ptr Set5+2, es
                mov     word ptr Set6+2, es
                mov     word ptr Set7+2, es

                mov     word ptr Set0, di
                inc     di
                mov     word ptr Set1, di
                inc     di
                mov     word ptr Set2, di
                inc     di
                mov     word ptr Set3, di
                inc     di
                mov     word ptr Set4, di
                inc     di
                mov     word ptr Set5, di
                inc     di
                mov     word ptr Set6, di
                inc     di
                mov     word ptr Set7, di
                inc     di
```

This code segment creates eight different sets on the heap, all empty, and stores pointers to them in the appropriate pointer variables.

The SHELL.ASM file provides a commented-out line of code in the data segment that includes the file STDSETS.A. This include file provides the bit definitions for eight commonly used character sets. They are alpha (upper and lower case alphabetics), lower (lower case alphabetics), upper (upper case alphabetics), digits ("0".."9"), xdigits ("0".."9", "A".."F", and "a".."f"), alphanum (upper and lower case alphabetics plus the digits), whitespace (space, tab, carriage return, and line feed), and delimiters (whitespace plus commas, semicolons, less than, greater than, and vertical bar). If you would like to use these standard character sets in your program, you need to remove the semicolon from the beginning of the include statement in the SHELL.ASM file.

The UCR Standard Library provides 16 character set routines: CreateSets, EmptySet, RangeSet, AddStr, AddStrl, RmvStr, RmvStrl, AddChar, RmvChar, Member, CopySet, SetUnion, SetIntersect, SetDifference, NextItem, and RmvItem. All of these routines except CreateSets require a pointer to a character set variable in the es:di registers. Specific routines may require other parameters as well.

The EmptySet routine clears all the bits in a set producing the empty set. This routine requires the address of the set variable in the es:di. The following example clears the set pointed at by Set1:

```
les      di, Set1
EmptySet
```

RangeSet unions in a range of values into the set variable pointed at by es:di. The al register contains the lower bound of the range of items, ah contains the upper bound. Note that al must be less than or equal to ah. The following example constructs the set of all control characters (ASCII codes one through 31, the null character [ASCII code zero] is not allowed in sets):

```
les      di, CtrlCharSet         ;Ptr to ctrl char set.
mov      al, 1
mov      ah, 31
RangeSet
```

AddStr and AddStrl add all the characters in a zero terminated string to a character set. For AddStr, the dx:si register pair points at the zero terminated string. For AddStrl, the zero terminated string follows the call to AddStrl in the code stream. These routines union each character of the specified string into the set. The following examples add the digits and some special characters into the FPDigits set:

```
Digits           byte     "0123456789",0
                 set      FPDigitsSet
FPDigits         dword    FPDigitsSet
                   .
                   .
                   .
                 ldxi     Digits       ;Loads DX:SI with adrs of Digits.
                 les      di, FPDigits
                 AddStr
                   .
                   .
                   .
                 les      di, FPDigits
                 AddStrL
                 byte     "Ee.+-",0
```

RmvStr and RmvStrl *remove* characters from a set. You supply the characters in a zero terminated string. For RmvStr, dx:si points at the string of characters to remove from the string. For RmvStrl, the zero terminated string follows the call. The following example uses RmvStrl to remove the special symbols from FPDigits above:

```
les      di, FPDigits
RmvStrl
byte     "Ee.+-",0
```

The AddChar and RmvChar routines let you add or remove individual characters. As usual, es:di points at the set; the al register contains the character you wish to add to the set or remove from the set. The following example adds a space to the set FPDigits and removes the "," character (if present):

```
les      di, FPDigits
mov      al, ' '
AddChar
  .
  .
  .
les      di, FPDigits
mov      al, ','
RmvChar
```

The Member function checks to see if a character is in a set. On entry, es:di must point at the set and al must contain the character to check. On exit, the zero flag is set if the character is a member of the set, the zero flag will be clear if the character is not in the set. The following example reads characters from the keyboard until the user presses a key that is not a whitespace character:

```
SkipWS:         get                     ;Read char from user into AL.
                lesi     WhiteSpace     ;Address of WS set into es:di.
                member
                je       SkipWS
```

The CopySet, SetUnion, SetIntersect, and SetDifference routines all operate on two sets of characters. The es:di register points at the destination character set, the dx:si register pair points at a source character set. CopySet copies the bits from the source set to the destination set, replacing the original bits in the destination set. SetUnion computes the union of the two sets and stores the result into the destination set. SetIntersect computes the set intersection and stores the result into the destination set. Finally, the SetDifference routine computes DestSet := DestSet - SrcSet.

The NextItem and RmvItem routines let you extract elements from a set. NextItem returns in al the ASCII code of the first character it finds in a set. RmvItem does the same thing except it also removes the character from the set. These routines return zero in al if the set is empty (StdLib sets cannot contain the NULL character). You can use the RmvItem routine to build a rudimentary iterator for a character set.

The UCR Standard Library's character set routines are very powerful. With them, you can easily manipulate character string data, especially when searching for different patterns within a string. We will consider this routines again when we study pattern matching later in this text (see "Pattern Matching" on page 883).

15.6 Using the String Instructions on Other Data Types

The string instructions work with other data types besides character strings. You can use the string instructions to copy whole arrays from one variable to another, to initialize large data structures to a single value, or to compare entire data structures for equality or inequality. Anytime you're dealing with data structures containing several bytes, you may be able to use the string instructions.

15.6.1 Multi-precision Integer Strings

The cmps instruction is useful for comparing (very) large integer values. Unlike character strings, we cannot compare integers with cmps from the L.O. byte through the H.O. byte. Instead, we must compare them from the H.O. byte down to the L.O. byte. The following code compares two 12-byte integers:

```
                lea      di, integer1+10
                lea      si, integer2+10
                mov      cx, 6
                std
        repe    cmpsw
```

After the execution of the cmpsw instruction, the flags will contain the result of the comparison.

You can easily assign one long integer string to another using the movs instruction. Nothing tricky here, just load up the si, di, and cx registers and have at it. You must do other operations, including arithmetic and logical operations, using the extended precision methods described in the chapter on arithmetic operations.

15.6.2 Dealing with Whole Arrays and Records

The only operations that apply, in general, to all array and record structures are assignment and comparison (for equality/inequality only). You can use the movs and cmps instructions for these operations.

Operations such as scalar addition, transposition, etc., may be easily synthesized using the lods and stos instructions. The following code shows how you can easily add the value 20 to each element of the integer array A:

```
                lea     si, A
                mov     di, si
                mov     cx, SizeOfA
                cld
AddLoop:        lodsw
                add     ax, 20
                stosw
                loop    AddLoop
```

You can implement other operations in a similar fashion.

15.7 Sample Programs

In this section there are three sample programs. The first searches through a file for a particular string and displays the line numbers of any lines containing that string. This program demonstrates the use of the strstr function (among other things). The second program is a demo program that uses several of the string functions available in the UCR Standard Library's string package. The third program demonstrates how to use the 80x86 cmps instruction to compare the data in two files. These programs (find.asm, strdemo.asm, and fcmp.asm) are available on the companion CD-ROM.

15.7.1 Find.asm

```
; Find.asm
;
; This program opens a file specified on the command line and searches for
; a string (also specified on the command line).
;
; Program Usage:
;
;       find "string" filename


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

wp              textequ   <word ptr>

dseg            segment   para public 'data'

StrPtr          dword     ?
FileName        dword     ?
LineCnt         dword     ?

FVar            filevar   {}

InputLine       byte      1024 dup (?)
dseg            ends
```

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


; Readln-       This procedure reads a line of text from the input
;               file and buffers it up in the "InputLine" array.

ReadLn          proc
                push     es
                push     ax
                push     di
                push     bx

                lesi     FVar          ;Read from our file.
                mov      bx, 0         ;Index into InputLine.
ReadLp:         fgetc                  ;Get next char from file.
                jc       EndRead       ;Quit on EOF

                cmp      al, cr        ;Ignore carriage returns.
                je       ReadLp
                cmp      al, lf        ;End of line on line feed.
                je       EndRead

                mov      InputLine[bx], al
                inc      bx
                jmp      ReadLp

; If we hit the end of a line or the end of the file,
; zero-terminate the string.

EndRead:        mov      InputLine[bx], 0
                pop      bx
                pop      di
                pop      ax
                pop      es
                ret
ReadLn          endp


; The following main program extracts the search string and the
; filename from the command line, opens the file, and then searches
; for the string in that file.

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

                argc
                cmp      cx, 2
                je       GoodArgs
                print
                byte     "Usage: find 'string' filename",cr,lf,0
                jmp      Quit

GoodArgs:       mov      ax, 1         ;Get the string to search for
                argv                   ; off the command line.
                mov      wp StrPtr, di
                mov      wp StrPtr+2, es

                mov      ax, 2         ;Get the filename from the
                argv                   ; command line.
                mov      wp Filename, di
                mov      wp Filename+2, es

; Open the input file for reading

                mov      ax, 0         ;Open for read.
                mov      si, wp FileName
```

```
                mov     dx, wp FileName+2
                lesi    Fvar
                fopen
                jc      BadOpen

; Okay, start searching for the string in the file.

                mov     wp LineCnt, 0
                mov     wp LineCnt+2, 0
SearchLp:       call    ReadLn
                jc      AtEOF


; Bump the line number up by one.  Note that this is 8086 code
; so we have to use extended precision arithmetic to do a 32-bit
; add.  LineCnt is a 32-bit variable because some files have more
; that 65,536 lines.

                add     wp LineCnt, 1
                adc     wp LineCnt+2, 0

; Search for the user-specified string on the current line.

                lesi    InputLine
                mov     dx, wp StrPtr+2
                mov     si, wp StrPtr
                strstr
                jc      SearchLp;Jump if not found.

; Print an appropriate message if we found the string.

                printf
                byte    "Found '%^s' at line %ld\n",0
                dword   StrPtr, LineCnt
                jmp     SearchLp

; Close the file when we're done.

AtEOF:          lesi    FVar
                fclose
                jmp     Quit

BadOpen:        printf
                byte    "Error attempting to open %^s\n",cr,lf,0
                dword   FileName


Quit:           ExitPgm                 ;DOS macro to quit program.
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
```

15.7.2 StrDemo.asm

This short demo program just shows off how to use several of the string routines found in the UCR Standard Library strings package.

```
; StrDemo.asm-  Demonstration of some of the various UCR Standard Library
;               string routines.
```

```
                include    stdlib.a
                includelib stdlib.lib

dseg            segment  para public 'data'

MemAvail        word     ?
String          byte     256 dup (0)

dseg            ends



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


Main            proc
                mov      ax, seg dseg ;Set up the segment registers
                mov      ds, ax
                mov      es, ax

                MemInit
                mov      MemAvail, cx
                printf
                byte     "There are %x paragraphs of memory available."
                byte     cr,lf,lf,0
                dword    MemAvail

; Demonstration of StrTrim:

                print
                byte     "Testing strtrim on 'Hello there   '",cr,lf,0
                strdupl
HelloThere1     byte     "Hello there   ",0
                strtrim
                mov      al, "'"
                putc
                puts
                putc
                putcr
                free

;Demonstration of StrTrimm:

                print
                byte     "Testing strtrimm on 'Hello there   '",cr,lf,0
                lesi     HelloThere1
                strtrimm
                mov      al, "'"
                putc
                puts
                putc
                putcr
                free

; Demonstration of StrBdel

                print
                byte     "Testing strbdel on '  Hello there   '",cr,lf,0
                strdupl
HelloThere3     byte     "  Hello there   ",0
                strbdel
                mov      al, "'"
                putc
                puts
                putc
                putcr
                free
```

```
; Demonstration of StrBdelm

                print
                byte     "Testing strbdelm on '  Hello there   '",cr,lf,0
                lesi     HelloThere3
                strbdelm
                mov      al, "'"
                putc
                puts
                putc
                putcr
                free


; Demonstrate StrCpyl:

                ldxi     string
                strcpyl
                byte     "Copy this string to the 'String' variable",0

                printf
                byte     "STRING = '%s'",cr,lf,0
                dword    String

; Demonstrate StrCatl:

                lesi     String
                strcatl
                byte     ". Put at end of 'String'",0

                printf
                byte     "STRING = ",'"%s"',cr,lf,0
                dword    String

; Demonstrate StrChr:

                lesi     String
                mov      al, "'"
                strchr

                print
                byte     "StrChr: First occurrence of ", '"', "'"
                byte     '" found at position ',0
                mov      ax, cx
                puti
                putcr

; Demonstrate StrStrl:

                lesi     String
                strstrl
                byte     "String",0

                print
                byte     'StrStr: First occurrence of "String" found at ‘
                byte     ‘position ',0

                mov      ax, cx
                puti
                putcr

; Demo of StrSet

                lesi     String
                mov      al, '*'
                strset

                printf
                byte     "Strset:  '%s'",cr,lf,0
                dword    String
```

```
; Demo of strlen

                lesi     String
                strlen

                print
                byte     "String length = ",0
                puti
                putcr

Quit:           mov     ah, 4ch
                int     21h
Main            endp

cseg            ends

sseg            segment  para stack 'stack'
stk             db       256 dup ("stack   ")
sseg            ends

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

15.7.3 Fcmp.asm

This is a file comparison program. It demonstrates the use of the 80x86 cmps instruction (as well as blocked I/O under DOS).

```
; FCMP.ASM-     A file comparison program that demonstrates the use
;               of the 80x86 string instructions.

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

dseg            segment    para public 'data'

Name1           dword    ?            ;Ptr to filename #1
Name2           dword    ?            ;Ptr to filename #2
Handle1         word     ?            ;File handle for file #1
Handle2         word     ?            ;File handle for file #2
LineCnt         word     0            ;# of lines in the file.

Buffer1         byte     256 dup (0)  ;Block of data from file 1
Buffer2         byte     256 dup (0)  ;Block of data from file 2

dseg            ends

wp              equ      <word ptr>

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

; Error- Prints a DOS error message depending upon the error type.

Error           proc     near
                cmp      ax, 2
                jne      NotFNF
                print
                byte     "File not found",0
                jmp      ErrorDone

NotFNF:         cmp      ax, 4
                jne      NotTMF
```

```
                print
                byte    "Too many open files",0
                jmp     ErrorDone

NotTMF:         cmp     ax, 5
                jne     NotAD
                print
                byte    "Access denied",0
                jmp     ErrorDone

NotAD:          cmp     ax, 12
                jne     NotIA
                print
                byte    "Invalid access",0
                jmp     ErrorDone

NotIA:
ErrorDone:      putcr
                ret
Error           endp

; Okay, here's the main program.  It opens two files, compares them, and
; complains if they're different.

Main            proc
                mov     ax, seg dseg            ;Set up the segment registers
                mov     ds, ax
                mov     es, ax
                meminit

; File comparison routine.  First, open the two source files.

                argc
                cmp     cx, 2                   ;Do we have two filenames?
                je      GotTwoNames
                print
                byte    "Usage: fcmp file1 file2",cr,lf,0
                jmp     Quit

GotTwoNames:    mov     ax, 1                   ;Get first file name
                argv
                mov     wp Name1, di
                mov     wp Name1+2, es

; Open the files by calling DOS.

                mov     ax, 3d00h               ;Open for reading
                lds     dx, Name1
                int     21h
                jnc     GoodOpen1
                printf
                byte    "Error opening %^s:",0
                dword   Name1
                call    Error
                jmp     Quit

GoodOpen1:      mov     dx, dseg
                mov     ds, dx
                mov     Handle1, ax

                mov     ax, 2                   ;Get second file name
                argv
                mov     wp Name2, di
                mov     wp Name2+2, es

                mov     ax, 3d00h               ;Open for reading
                lds     dx, Name2
                int     21h
                jnc     GoodOpen2
                printf
```

```
                byte    "Error opening %^s:",0
                dword   Name2
                call    Error
                jmp     Quit

GoodOpen2:      mov     dx, dseg
                mov     ds, dx
                mov     Handle2, ax

; Read the data from the files using blocked I/O
; and compare it.

                mov     LineCnt, 1
CmpLoop:        mov     bx, Handle1             ;Read 256 bytes from
                mov     cx, 256                 ; the first file into
                lea     dx, Buffer1             ; Buffer1.
                mov     ah, 3fh
                int     21h
                jc      FileError
                cmp     ax, 256                 ;Leave if at EOF.
                jne     EndOfFile

                mov     bx, Handle2             ;Read 256 bytes from
                mov     cx, 256                 ; the second file into
                lea     dx, Buffer2             ; Buffer2
                mov     ah, 3fh
                int     21h
                jc      FileError
                cmp     ax, 256                 ;If we didn't read 256 bytes,
                jne     BadLen                  ; the files are different.

; Okay, we've just read 256 bytes from each file, compare the buffers
; to see if the data is the same in both files.

                mov     ax, dseg
                mov     ds, ax
                mov     es, ax
                mov     cx, 256
                lea     di, Buffer1
                lea     si, Buffer2
                cld
        repe    cmpsb
                jne     BadCmp
                jmp     CmpLoop


FileError:      print
                byte    "Error reading files: ",0
                call    Error
                jmp     Quit


BadLen:         print
                byte    "File lengths were different",cr,lf,0

BadCmp:         print
                byte    7,"Files were not equal",cr,lf,0

                mov     ax, 4c01h    ;Exit with error.
                int     21h


; If we reach the end of the first file, compare any remaining bytes
; in that first file against the remaining bytes in the second file.

EndOfFile:      push    ax           ;Save final length.
                mov     bx, Handle2
                mov     cx, 256
                lea     dx, Buffer2
                mov     ah, 3fh
```

```
                int     21h
                jc      BadCmp

                pop     bx          ;Retrieve file1's length.
                cmp     ax, bx      ;See if file2 matches it.
                jne     BadLen

                mov     cx, ax      ;Compare the remaining
                mov     ax, dseg    ; bytes down here.
                mov     ds, ax
                mov     es, ax
                lea     di, Buffer2
                lea     si, Buffer1
        repe    cmpsb
                jne     BadCmp

Quit:           mov     ax, 4c00h   ;Set Exit code to okay.
                int     21h
Main            endp
cseg            ends

; Allocate a reasonable amount of space for the stack (2k).

sseg            segment para stack 'stack'
stk             byte    256 dup ("stack   ")
sseg            ends

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

15.8 Laboratory Exercises

These exercises use the Ex15_1.asm, Ex15_2.asm, Ex15_3.asm, and Ex15_4.asm files found on the companion CD-ROM. In this set of laboratory exercises you will be measuring the performance of the 80x86 movs instructions and the (hopefully) minor performance differences between length prefixed string operations and zero terminated string operations.

15.8.1 MOVS Performance Exercise #1

The movsb, movsw, and movsd instructions operate at different speeds, even when moving around the same number of bytes. In general, the movsw instruction is twice as fast as movsb when moving the same number of bytes. Likewise, movsd is about twice as fast as movsw (and about four times as fast as movsb) when moving the same number of bytes. Ex15_1.asm is a short program that demonstrates this fact. This program consists of three sections that copy 2048 bytes from one buffer to another 100,000 times. The three sections repeat this operation using the movsb, movsw, and movsd instructions. Run this program and time each phase. **For your lab report:** present the timings on your machine. Be sure to list processor type and clock frequency in your lab report. Discuss why the timings are different between the three phases of this program. Explain the difficulty with using the movsd (versus movsw or movsb) instruction in any program on an 80386 or later processor. Why is it not a general replacement for movsb, for example? How can you get around this problem?

```
; EX15_1.asm
;
; This program demonstrates the proper use of the 80x86 string instructions.

                .386
                option    segment:use16
```

```
                include     stdlib.a
                includelib  stdlib.lib


dseg            segment   para public 'data'

Buffer1         byte      2048 dup (0)
Buffer2         byte      2048 dup (0)

dseg            ends

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

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

; Demo of the movsb, movsw, and movsd instructions

                print
                byte      "The following code moves a block of 2,048 bytes "
                byte      "around 100,000 times.",cr,lf
                byte      "The first phase does this using the movsb "
                byte      "instruction; the second",cr,lf
                byte      "phase does this using the movsw instruction; "
                byte      "the third phase does",cr,lf
                byte      "this using the movsd instruction.",cr,lf,lf,lf
                byte      "Press any key to begin phase one:",0

                getc
                putcr

                mov       edx, 100000

movsbLp:        lea       si, Buffer1
                lea       di, Buffer2
                cld
                mov       cx, 2048
        rep     movsb
                dec       edx
                jnz       movsbLp

                print
                byte      cr,lf
                byte      "Phase one complete",cr,lf,lf
                byte      "Press any key to begin phase two:",0

                getc
                putcr

                mov       edx, 100000

movswLp:        lea       si, Buffer1
                lea       di, Buffer2
                cld
                mov       cx, 1024
        rep     movsw
                dec       edx
                jnz       movswLp

                print
                byte      cr,lf
                byte      "Phase two complete",cr,lf,lf
                byte      "Press any key to begin phase three:",0

                getc
```

```
                putcr

                mov      edx, 100000

movsdLp:        lea      si, Buffer1
                lea      di, Buffer2
                cld
                mov      cx, 512
        rep     movsd
                dec      edx
                jnz      movsdLp


Quit:           ExitPgm             ;DOS macro to quit program.
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
```

15.8.2 MOVS Performance Exercise #2

In this exercise you will once again time the computer moving around blocks of 2,048 bytes. Like Ex15_1.asm in the previous exercise, Ex15_2.asm contains three phases; the first phase moves data using the movsb instruction; the second phase moves the data around using the lodsb and stosb instructions; the third phase uses a loop with simple mov instructions. Run this program and time the three phases. **For your lab report:** include the timings and a description of your machine (CPU, clock speed, etc.). Discuss the timings and explain the results (consult Appendix D as necessary).

```
; EX15_2.asm
;
; This program compares the performance of the MOVS instruction against
; a manual block move operation.  It also compares MOVS against a LODS/STOS
; loop.

                .386
                option     segment:use16

                include    stdlib.a
                includelib stdlib.lib


dseg            segment  para public 'data'

Buffer1         byte     2048 dup (0)
Buffer2         byte     2048 dup (0)

dseg            ends


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

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

```
; MOVSB version done here:

                print
                byte    "The following code moves a block of 2,048 bytes "
                byte    "around 100,000 times.",cr,lf
                byte    "The first phase does this using the movsb "
                byte    "instruction; the second",cr,lf
                byte    "phase does this using the lods/stos instructions; "
                byte    "the third phase does",cr,lf
                byte    "this using a loop with MOV “
                byte    “instructions.",cr,lf,lf,lf
                byte    "Press any key to begin phase one:",0

                getc
                putcr

                mov     edx, 100000

movsbLp:        lea     si, Buffer1
                lea     di, Buffer2
                cld
                mov     cx, 2048
        rep     movsb
                dec     edx
                jnz     movsbLp

                print
                byte    cr,lf
                byte    "Phase one complete",cr,lf,lf
                byte    "Press any key to begin phase two:",0

                getc
                putcr

                mov     edx, 100000

LodsStosLp:     lea     si, Buffer1
                lea     di, Buffer2
                cld
                mov     cx, 2048
lodsstoslp2:    lodsb
                stosb
                loop    LodsStosLp2
                dec     edx
                jnz     LodsStosLp

                print
                byte    cr,lf
                byte    "Phase two complete",cr,lf,lf
                byte    "Press any key to begin phase three:",0

                getc
                putcr

                mov     edx, 100000

MovLp:          lea     si, Buffer1
                lea     di, Buffer2
                cld
                mov     cx, 2048
MovLp2:         mov     al, ds:[si]
                mov     es:[di], al
                inc     si
                inc     di
                loop    MovLp2
                dec     edx
                jnz     MovLp
```

```
Quit:           ExitPgm             ;DOS macro to quit program.
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
```

15.8.3 Memory Performance Exercise

In the previous two exercises, the programs accessed a maximum of 4K of data. Since most modern on-chip CPU caches are at least this big, most of the activity took place directly on the CPU (which is very fast). The following exercise is a slight modification that moves the array data in such a way as to destroy cache performance. Run this program and time the results. **For your lab report:** based on what you learned about the 80x86's cache mechanism in Chapter Three, explain the performance differences.

```
; EX15_3.asm
;
; This program compares the performance of the MOVS instruction against
; a manual block move operation.  It also compares MOVS against a LODS/STOS
; loop.  This version does so in such a way as to wipe out the on-chip CPU
; cache.

                .386
                option     segment:use16

                include    stdlib.a
                includelib stdlib.lib

dseg            segment  para public 'data'

Buffer1         byte     16384 dup (0)
Buffer2         byte     16384 dup (0)

dseg            ends

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

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


; MOVSB version done here:

                print
                byte     "The following code moves a block of 16,384 bytes "
                byte     "around 12,500 times.",cr,lf
                byte     "The first phase does this using the movsb "
                byte     "instruction; the second",cr,lf
                byte     "phase does this using the lods/stos instructions; "
                byte     "the third phase does",cr,lf
                byte     "this using a loop with MOV instructions."
                byte     cr,lf,lf,lf
                byte     "Press any key to begin phase one:",0

                getc
```

```
                putcr

                mov     edx, 12500

movsbLp:        lea     si, Buffer1
                lea     di, Buffer2
                cld
                mov     cx, 16384
        rep     movsb
                dec     edx
                jnz     movsbLp

                print
                byte    cr,lf
                byte    "Phase one complete",cr,lf,lf
                byte    "Press any key to begin phase two:",0

                getc
                putcr

                mov     edx, 12500

LodsStosLp:     lea     si, Buffer1
                lea     di, Buffer2
                cld
                mov     cx, 16384
lodsstoslp2:    lodsb
                stosb
                loop    LodsStosLp2
                dec     edx
                jnz     LodsStosLp

                print
                byte    cr,lf
                byte    "Phase two complete",cr,lf,lf
                byte    "Press any key to begin phase three:",0

                getc
                putcr

                mov     edx, 12500

MovLp:          lea     si, Buffer1
                lea     di, Buffer2
                cld
                mov     cx, 16384
MovLp2:         mov     al, ds:[si]
                mov     es:[di], al
                inc     si
                inc     di
                loop    MovLp2
                dec     edx
                jnz     MovLp

Quit:           ExitPgm                 ;DOS macro to quit program.
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
```

15.8.4 The Performance of Length-Prefixed vs. Zero-Terminated Strings

The following program (Ex15_4.asm on the companion CD-ROM) executes two million string operations. During the first phase of execution, this code executes a sequence of length-prefixed string operations 1,000,000 times. During the second phase it does a comparable set of operation on zero terminated strings. Measure the execution time of each phase. **For your lab report:** report the differences in execution times and comment on the relative efficiency of length prefixed vs. zero terminated strings. Note that the relative performances of these sequences will vary depending upon the processor you use. Based on what you learned in Chapter Three and the cycle timings in Appendix D, explain some possible reasons for relative performance differences between these sequences among different processors.

```
; EX15_4.asm
;
; This program compares the performance of length prefixed strings versus
; zero terminated strings using some simple examples.
;
; Note: these routines all assume that the strings are in the data segment
;       and both ds and es already point into the data segment.

                 .386
                 option     segment:use16

                 include    stdlib.a
                 includelib stdlib.lib

dseg             segment  para public 'data'

LStr1            byte     17,"This is a string."
LResult          byte     256 dup (?)

ZStr1            byte     "This is a string",0
ZResult          byte     256 dup (?)

dseg             ends


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

; LStrCpy: Copies a length prefixed string pointed at by SI to
;          the length prefixed string pointed at by DI.

LStrCpy          proc
                 push     si
                 push     di
                 push     cx

                 cld

                 mov      cl, [si]     ;Get length of string.
                 mov      ch, 0
                 inc      cx           ;Include length byte.
       rep       movsb

                 pop      cx
                 pop      di
                 pop      si
                 ret
LStrCpy          endp

; LStrCat-       Concatenates the string pointed at by SI to the end
;                of the string pointed at by DI using length
;                prefixed strings.

LStrCat          proc
```

```
                push    si
                push    di
                push    cx

                cld

; Compute the final length of the concatenated string

                mov     cl, [di]                ;Get orig length.
                mov     ch, [si]                ;Get 2nd Length.
                add     [di], ch                ;Compute new length.

; Move SI to the first byte beyond the end of the first string.

                mov     ch, 0                   ;Zero extend orig len.
                add     di, cx                  ;Skip past str.
                inc     di                      ;Skip past length byte.

; Concatenate the second string (SI) to the end of the first string (DI)

        rep     movsb                           ;Copy 2nd to end of orig.

                pop     cx
                pop     di
                pop     si
                ret
LStrCat         endp

; LStrCmp-      String comparison using two length prefixed strings.
;               SI points at the first string, DI points at the
;               string to compare it against.

LStrCmp         proc
                push    si
                push    di
                push    cx

                cld

; When comparing the strings, we need to compare the strings
; up to the length of the shorter string.  The following code
; computes the minimum length of the two strings.

                mov     cl, [si]        ;Get the minimum of the two lengths
                mov     ch, [di]
                cmp     cl, ch
                jb      HasMin
                mov     cl, ch
HasMin:         mov     ch, 0

        repe    cmpsb                   ;Compare the two strings.
                je      CmpLen
                pop     cx
                pop     di
                pop     si
                ret

; If the strings are equal through the length of the shorter string,
; we need to compare their lengths

CmpLen:         pop     cx
                pop     di
                pop     si

                mov     cl, [si]
                cmp     cl, [di]
                ret
LStrCmp         endp

; ZStrCpy- Copies the zero terminated string pointed at by SI
```

```
;          to the zero terminated string pointed at by DI.

ZStrCpy          proc
                 push     si
                 push     di
                 push     ax

ZSCLp:           mov      al, [si]
                 inc      si
                 mov      [di], al
                 inc      di
                 cmp      al, 0
                 jne      ZSCLp

                 pop      ax
                 pop      di
                 pop      si
                 ret
ZStrCpy          endp


; ZStrCat-       Concatenates the string pointed at by SI to the end
;                of the string pointed at by DI using zero terminated
;                strings.

ZStrCat          proc
                 push     si
                 push     di
                 push     cx
                 push     ax

                 cld

; Find the end of the destination string:

                 mov      cx, 0FFFFh
                 mov      al, 0        ;Look for zero byte.
       repne     scasb

; Copy the source string to the end of the destination string:

ZcatLp:          mov      al, [si]
                 inc      si
                 mov      [di], al
                 inc      di
                 cmp      al, 0
                 jne      ZCatLp

                 pop      ax
                 pop      cx
                 pop      di
                 pop      si
                 ret
ZStrCat          endp

; ZStrCmp-       Compares two zero terminated strings.
;                This is actually easier than the length
;                prefixed comparison.

ZStrCmp          proc
                 push     cx
                 push     si
                 push     di

; Compare the two strings until they are not equal
; or until we encounter a zero byte.  They are equal
; if we encounter a zero byte after comparing the
; two characters from the strings.

ZCmpLp:          mov      al, [si]
```

```
                inc     si
                cmp     al, [di]
                jne     ZCmpDone
                inc     di
                cmp     al, 0
                jne     ZCmpLp

ZCmpDone:       pop     di
                pop     si
                pop     cx
                ret
ZStrCmp         endp


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

                print
                byte    "The following code does 1,000,000 string "
                byte    "operations using",cr,lf
                byte    "length prefixed strings.  Measure the amount "
                byte    "of time this code",cr,lf
                byte    "takes to run.",cr,lf,lf
                byte    "Press any key to begin:",0

                getc
                putcr

                mov     edx, 1000000
LStrCpyLp:      lea     si, LStr1
                lea     di, LResult
                call    LStrCpy
                call    LStrCat
                call    LStrCat
                call    LStrCat
                call    LStrCpy
                call    LStrCmp
                call    LStrCat
                call    LStrCmp

                dec     edx
                jne     LStrCpyLp

                print
                byte    "The following code does 1,000,000 string "
                byte    "operations using",cr,lf
                byte    "zero terminated strings.  Measure the amount "
                byte    "of time this code",cr,lf
                byte    "takes to run.",cr,lf,lf
                byte    "Press any key to begin:",0

                getc
                putcr

                mov     edx, 1000000
ZStrCpyLp:      lea     si, ZStr1
                lea     di, ZResult
                call    ZStrCpy
                call    ZStrCat
                call    ZStrCat
                call    ZStrCat
                call    ZStrCpy
                call    ZStrCmp
                call    ZStrCat
                call    ZStrCmp

                dec     edx
```

```
                  jne      ZStrCpyLp

Quit:             ExitPgm             ;DOS macro to quit program.
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
```

15.9 Programming Projects

1) Write a SubStr function that extracts a substring from a zero terminated string. Pass a pointer to the string in ds:si, a pointer to the destination string in es:di, the starting position in the string in ax, and the length of the substring in cx. Follow all the rules given in section 15.3.1 concerning degenerate conditions.

2) Write a word *iterator* (see “Iterators” on page 663) to which you pass a string (by reference, on the stack). Each each iteration of the corresponding foreach loop should extract a word from this string, malloc sufficient storage for this string on the heap, copy that word (substring) to the malloc’d location, and return a pointer to the word. Write a main program that calls the iterator with various strings to test it.

3) Modify the *find.asm* program (see “Find.asm” on page 860) so that it searches for the desired string in several files using ambiguous filenames (i.e., wildcard characters). See “Find First File” on page 729 for details about processing filenames that contain wildcard characters. You should write a loop that processes all matching filenames and executes the find.asm core code on each filename that matches the ambiguous filename a user supplies.

4) Write a strncpy routine that behaves like strcpy except it copies a maximum of n characters (including the zero terminating byte). Pass the source string’s address in es:di, the destination string’s address in dx:si, and the maximum length in cx.

5) The movsb instruction may not work properly if the source and destination blocks overlap (see “The MOVS Instruction” on page 822). Write a procedure “bcopy” to which you pass the address of a source block, the address of a destination block, and a length, that will properly copy the data even if the source and destination blocks overlap. Do this by checking to see if the blocks overlap and adjusting the source pointer, destination pointer, and direction flag if necessary.

6) As you discovered in the lab experiments, the movsd instruction can move a block of data much faster than movsb or movsw can move that same block. Unfortunately, it can only move a block that contains an even multiple of four bytes. Write a “fastcopy” routine that uses the movsd instruction to copy all but the last one to three bytes of a source block to the destination block and then manually copies the remaining bytes between the blocks. Write a main program with several boundary test cases to verify correct operation. Compare the performance of your fastcopy procedure against the use of the movsb instruction.

15.10 Summary

The 80sx86 provides a powerful set of string instructions. However, these instructions are very primitive, useful mainly for manipulating blocks of bytes. They do not correspond to the string instructions one expects to find in a high level language. You can, however, use the 80x86 string instructions to synthesize those functions normally associated with HLLs. This chapter explains how to construct many of the more popular string functions. Of course, it's foolish to constantly reinvent the wheel, so this chapter also describes many of the string functions available in the UCR Standard Library.

The 80x86 string instructions provide the basis for many of the string operations appearing in this chapter. Therefore, this chapter begins with a review and in-depth discussion of the 80x86 string instructions: the repeat prefixes, and the direction flag. This chapter discusses the operation of each of the string instructions and describes how you can use each of them to perform string related tasks. To see how the 80x86 string instructions operate, check out the following sections:

- "The 80x86 String Instructions" on page 819
- "How the String Instructions Operate" on page 819
- "The REP/REPE/REPZ and REPNZ/REPNE Prefixes" on page 820
- "The Direction Flag" on page 821
- "The MOVS Instruction" on page 822
- "The CMPS Instruction" on page 826
- "The SCAS Instruction" on page 828
- "The STOS Instruction" on page 828
- "The LODS Instruction" on page 829
- "Building Complex String Functions from LODS and STOS" on page 830
- "Prefixes and the String Instructions" on page 830

Although Intel calls them "string instructions" they do not actually work on the abstract data type we normally think of as a character string. The string instructions simply manipulate arrays of bytes, words, or double words. It takes a little work to get these instructions to deal with true character strings. Unfortunately, there isn't a single definition of a character string which, no doubt, is the reason there aren't any instructions specifically for character strings in the 80x86 instruction set. Two of the more popular character string types include length prefixed strings and zero terminated strings which Pascal and C use, respectively. Details on string formats appear in the following sections:

- "Character Strings" on page 831
- "Types of Strings" on page 831

Once you decide on a specific data type for you character strings, the next step is to implement various functions to process those strings. This chapter provides examples of several different string functions designed specifically for length prefixed strings. To learn about these functions and see the code that implements them, look at the following sections:

- "String Assignment" on page 832
- "String Comparison" on page 834
- "Character String Functions" on page 835
- "Substr" on page 835
- "Index" on page 838
- "Repeat" on page 840
- "Insert" on page 841
- "Delete" on page 843
- "Concatenation" on page 844

The UCR Standard Library provides a very rich set of string functions specifically designed for zero germinated strings. For a description of many of these routines, read the following sections:

- "String Functions in the UCR Standard Library" on page 845
- "StrBDel, StrBDelm" on page 846
- "Strcat, Strcatl, Strcatm, Strcatml" on page 847
- "Strchr" on page 848
- "Strcmp, Strcmpl, Stricmp, Stricmpl" on page 848
- "Strcpy, Strcpyl, Strdup, Strdupl" on page 849

As mentioned earlier, the string instructions are quite useful for many operations beyond character string manipulation. This chapter closes with some sections describing other uses for the string instructions. See

The set is another common abstract data type commonly found in programs today. A set is a data structure which represent membership (or lack thereof) of some group of objects. If all objects are of the same underlying base type and there is a limited number of possible objects in the set, then we can use a *bit vector* (array of booleans) to represent the set. The bit vector implementation is very efficient for small sets. The UCR Standard Library provides several routines to manipulate character sets and other sets with a maximum of 256 members. For more details,

15.11 Questions

1) What are the repeat prefixes used for?

2) Which string prefixes are used with the following instructions?

   a) MOVS b) CMPS c) STOS d) SCAS

3) Why aren't the repeat prefixes normally used with the LODS instruction?

4) What happens to the SI, DI, and CX registers when the MOVSB instruction is executed (without a repeat prefix) and:

   a) the direction flag is set. b) the direction flag is clear.

5) Explain how the MOVSB and MOVSW instructions work. Describe how they affect memory and registers with and without the repeat prefix. Describe what happens when the direction flag is set and clear.

6) How do you preserve the value of the direction flag across a procedure call?

7) How can you ensure that the direction flag always contains a proper value before a string instruction without saving it inside a procedure?

8) What is the difference between the "MOVSB", "MOVSW", and "MOVS oprnd1,oprnd2" instructions?

9) Consider the following Pascal array definition:

```
a:array [0..31] of record
           a,b,c:char;
           i,j,k:integer;
          end;
```

   Assuming A[0] has been initialized to some value, explain how you can use the MOVS instruction to initialize the remaining elements of A to the same value as A[0].

10) Give an example of a MOVS operation which requires the direction flag to be:

   a) clear b) set

11) How does the CMPS instruction operate? (what does it do, how does it affect the registers and flags, etc.)

12) Which segment contains the source string? The destination string?

13) What is the SCAS instruction used for?

14) How would you quickly initialize an array to all zeros?

15) How are the LODS and STOS instructions used to build complex string operations?

16) How would you use the SUBSTR function to extract a substring of length 6 starting at offset 3 in the StrVar variable, storing the substring in the NewStr variable?

17) What types of errors can occur when the SUBSTR function is executed?

18) Give an example demonstrating the use of each of the following string functions:

   a) INDEX b) REPEAT c) INSERT d) DELETE e) CONCAT

19) Write a short loop which multiplies each element of a single dimensional array by 10. Use the string instructions to fetch and store each array element.

20) The UCR Standard Library does not provide an STRCPYM routine. What is the routine which performs this task?

21) Suppose you are writing an "adventure game" into which the player types sentences and you want to pick out the two words "GO" and "NORTH", if they are present, in the input line. What (non-UCR StdLib) string function appearing in this chapter would you use to search for these words? What UCR Standard Library routine would you use?

22) Explain how to perform an extended precision integer comparison using CMPS