developer home contents search feedback support

Optimized String Search with MMX™ Technology

CONTENTS:

1.0. INTRODUCTION

Text string searches work by linearly checking every character of the text with the first character of the search string, and then checking for the second letter when the first is found, and so on. Traditionally, this is done one character at a time. However, with the MMX™ technology extensions to the Intel Architecture, specifically the PCMPEQB instruction, 8 characters can be compared at once. This application note demonstrates two Single-Instruction, Multiple-Data (SIMD) approaches. The first function has two improvements over the traditional method: it compares eight characters at a time, and it also checks the second character of the search string against eight characters of text. The second function shows a case-insensitive version which uses MMX™ instructions to efficiently check the case of eight characters and convert them to lower case if they are capital letters.

2.0. OVERVIEW

Scalar string searches work on a fairly simple algorithm:

1. Read in the first character of the search string.
2. Read in the next character of the text, and compare text character with search string character.
3. If the characters are different, go to Step 2, unless the remaining text is not as long as the search string. If so, then go to Step 6.
4. Read in the next string character and the next text character, and compare.
5. If the characters are the same, go to Step 4, unless it is the end of the search string.
6. Return the address of the start of the search string in the text, or NULL if not found.

This is implemented in the standard C library function strtstr(), which was used for comparision purposes, and will not be explored in further detail.

2.1. MMX™ Technology Version

This function uses MMX™ instructions to check for the first two letters of the search string in eight characters from the text. It has the following basic form:

1. Initialization: loading pointers to the search string (SearchStr) and the text (TxtBuff), propagate the first two characters of the search string (SearchStr[0] and SearchStr[1]) to each fill an MMX™ register, load the first 8 bytes of text, and align the text so that future reads are 8-byte aligned. There is also an early-out test to see whether the text buffer is longer that the search string. If not, it skips to Step 7.
2. Compare the current 8 bytes of text with SearchStr[0] and SearchStr[1], shift the result of SearchStr[0] so that it lines up with the result of SearchStr[1] and compare the two together (see Figure 3, Section 2.2.2). Save the last byte of the SearchStr[0] compare for the next time through the loop. Pack the final result of the compares into 32 bits and save into EAX.
3. This part of the code checks to see if the previous step matched two characters. It does not use MMX™ instructions because it has to check each character individually, and MMX™ instructions are not well-suited to data-dependant operations. First, there is an early-out test to check all 8 characters (each of which is now stored in 4 bits because of the packing instruction) to see if a character-by-character examination is necessary. If there are no positive results, it skips to Step 6. Otherwise, EAX is examined 4 bits at a time. If the low 4 bits are zero, then EAX is shifted to the next 4 bits and the appropriate text pointers are incremented until a positive result is found.
4. This step compares the remaining characters in the search string to the current position in the text buffer one character at a time, using scalar instructions. Insuring aligned access and checking for the end of the search string adds too much overhead for this to be efficiently accomplished using MMX™ instructions. First, the next search character is read in and compared against itself to check for the end of the search string. If it is not null, then the text character is loaded and the two are compared, until a compare fails and it jumps back to Step 3 to look for additional hits in the current 8 bytes, or it reaches the end of the search string and continues to Step 5.
5. EAX is loaded with the address of the start of the search string in the text buffer, and this value is returned.
6. (From Step 3) The character-by-character comparison failed: the first two characters matched but the search string was not found. The text pointer is incremented to the next 8 bytes and those characters are loaded, and the text buffer size is decreased by 8 and if that was not the end of the buffer, execution jumps to Step 2.
7. The search string was not found. The function returns NULL.

Due to the nature of the algorithm, this function will not find one-character search strings, because it will never match the first two characters. There is a special check at the beginning of the function to see if the search string is one character, in which case it is thrown to the regular string search.

Some of the more complex and/or MMX™ instruction intensive sections are explained in greater detail in the following sections.

2.1.1. Initialization

This diagram demonstrates how the PUNPCKL and PUNPKH instructions are used to propogate the first two characters of the search string to each fill a 64-bit register.

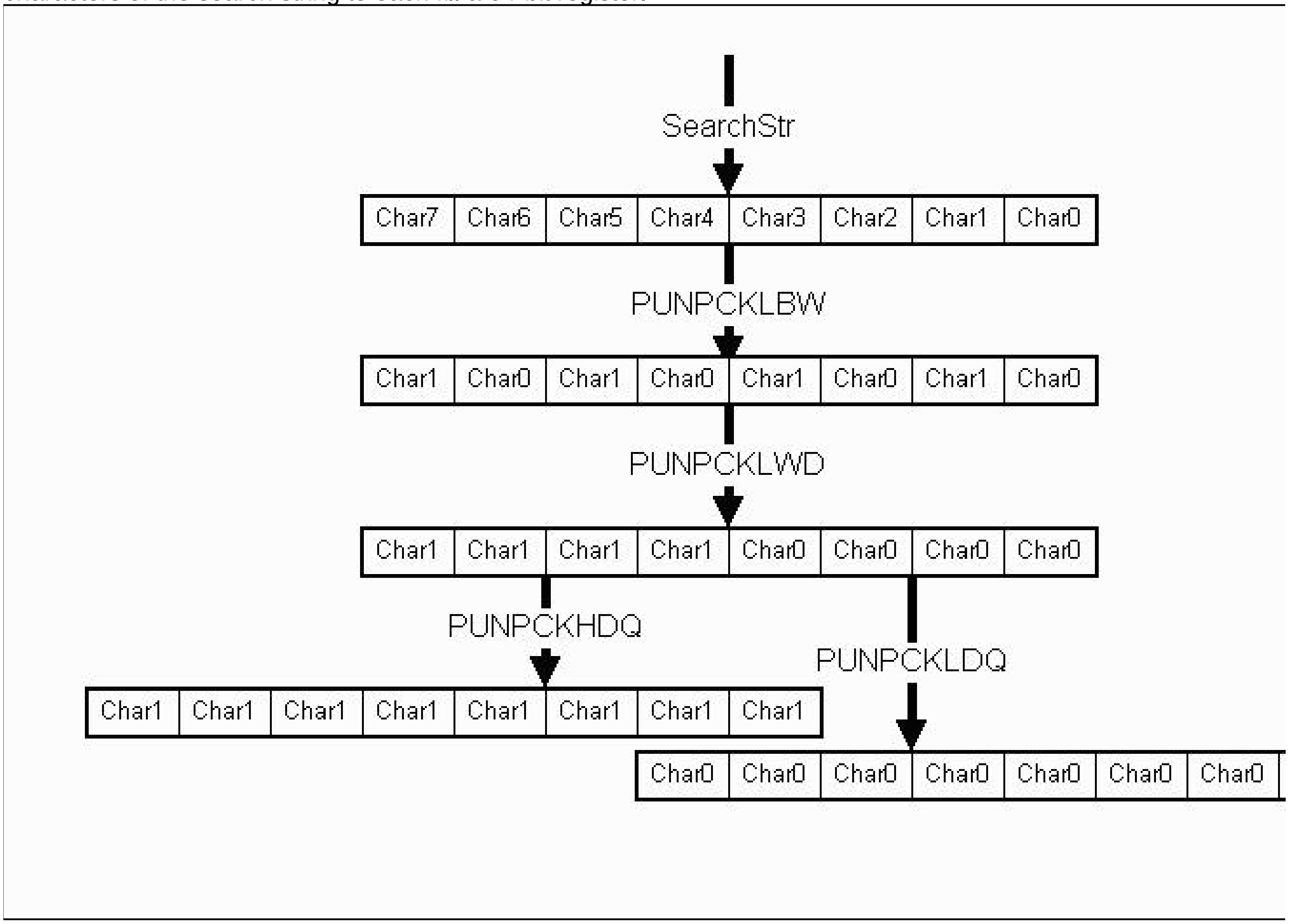

Figure 1. Unpacking the first two characters of SearchStr

The first instruction, PUNPCKLBW, replicates the low bytes throughout the register. Then PUNPCKLWD unpacks the low words into doublewords, and PUNPCKLDQ and PUNPCKHDQ unpacks the low and high doublewords into quadwords for SearchStr[0] and SearchStr[1], respectively.

It is important that the function performs its own memory alignment, because even if the text buffer is originally 8-byte aligned, if the user performs further searches starting where the last one left off, it is unlikely that the start will be on an 8-byte boundary. To handle this, the first text read is performed regardless of alignment, and then the data is shifted so that the next text read will be aligned, although it will re-read some overlapping characters. The search string is not read often enough to make it worth aligning, and as a smaller variable is likely to be aligned anyway.

2.1.2. Compare Two Characters

This is the outermost loop. First, it checks SearchStr[0] and SearchStr[1] against eight characters of text. The figure illustrates searching for 'the' in the text fragment 'that the'. Remember that the lowest byte of the MMX™ register is displayed to the right and the highest byte to the left. This is why the text fragment appears backward.

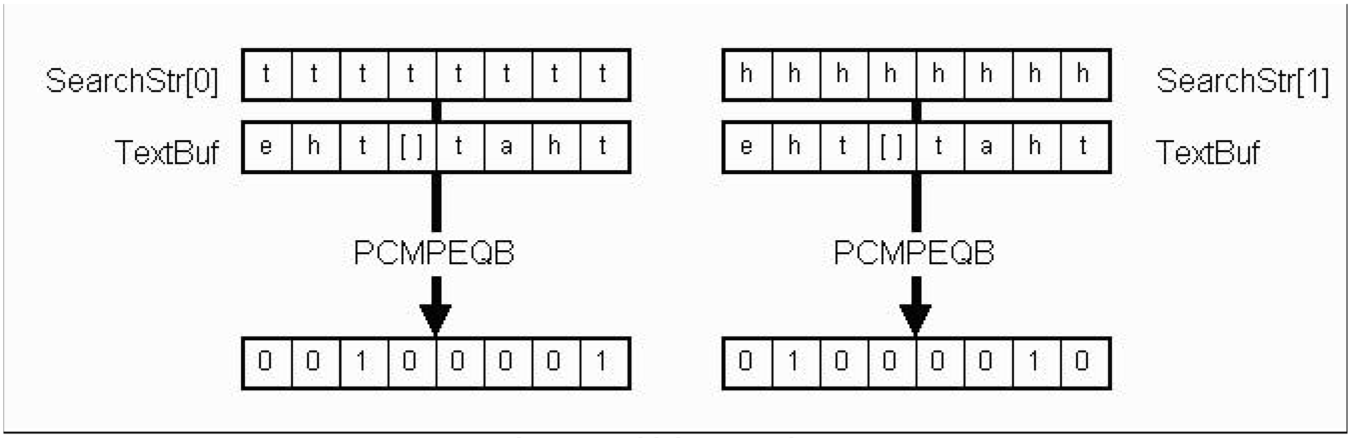

Figure 2. Initial Comparisons

The results from Figure 2 cannot immediately be used. If the text contains SearchStr[0], we want to see whether it also contains SearchStr[1] in the next character. This means that one of the results must be shifted one byte to line up with the other result. If the SearchStr[1] result was shifted to the right, then the high byte would need to contain the comparison result for the character *after* the last byte in SearchStr[0] , which will not be obtained until the next time through the loop. Shifting the SearchStr[1] result to the left requires the last byte from the previous SearchStr[1] result, which is much easier to obtain.

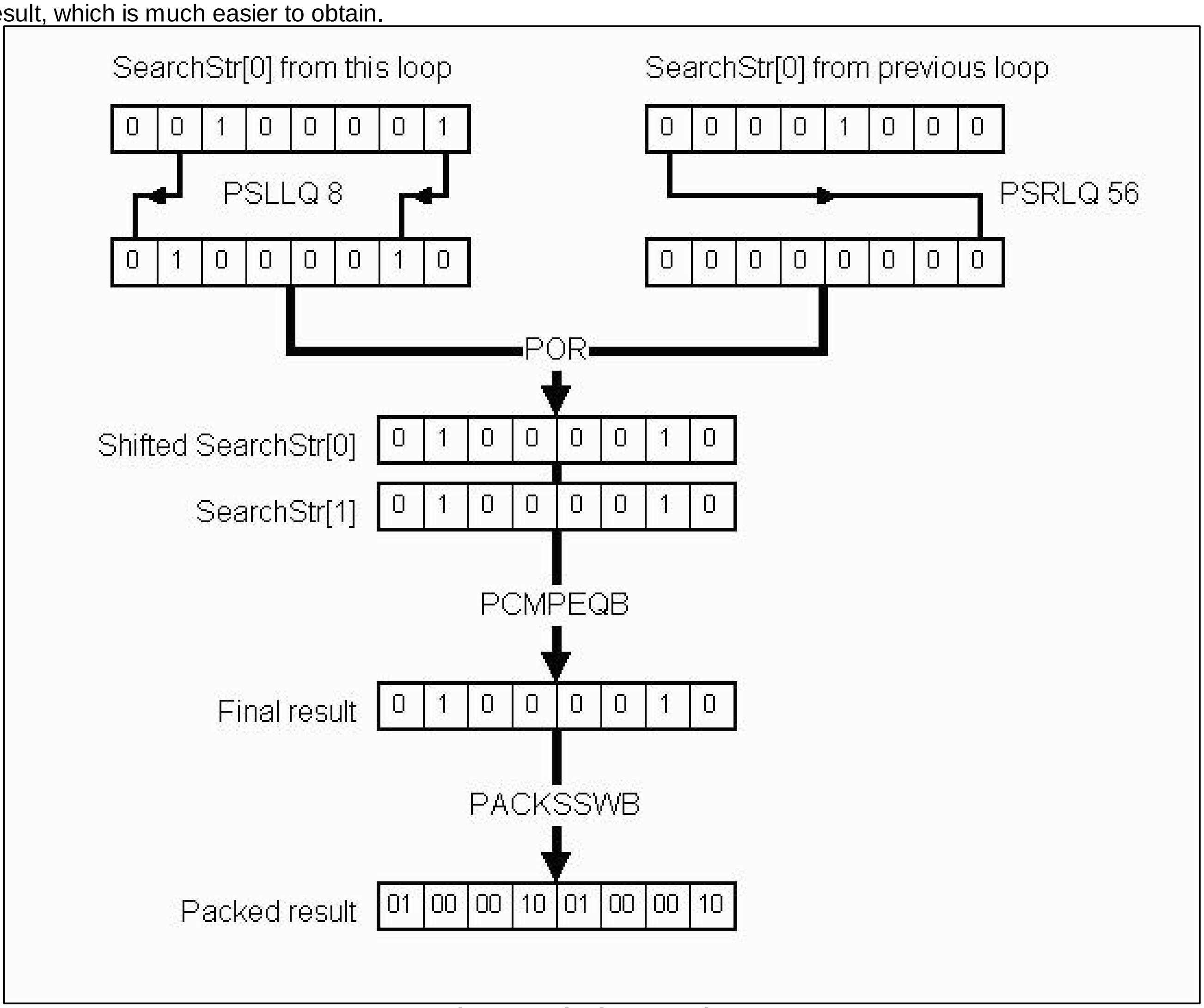

Figure 3. Final Comparisons

Figure 3 shows the shifts involved in preparing SearchStr[1] for comparison with SearchStr[0]. The SearchStr[1] comparison result from this loop is shifted one byte to the left, and is ORed with a copy of SearchStr[1] from the last loop that has been shifted to the right by 7 bytes. The POR instruction logically ORs all 64 bits. No masks are needed, because the left shift leaves the low byte zero, and the right shift leaves the high 7 bytes zero. Now, the SearchStr[0] and SearchStr[1] results can be compared and packed with saturation from words to bytes so that the entire MMX™ register can be represented with 32 bits.

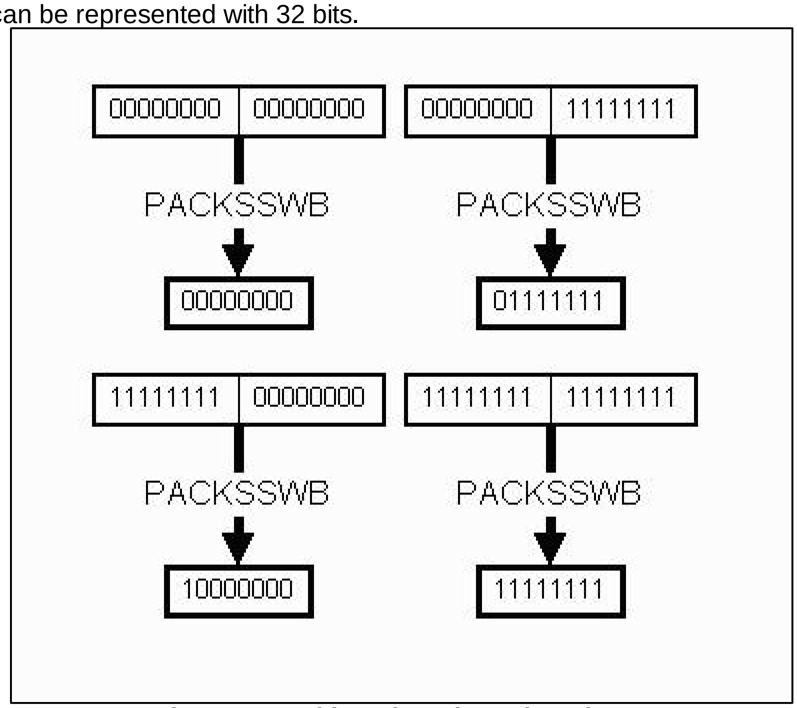

Figure 4. Packing signed words to bytes

Figure 3 illustrates the four possible cases of the PACKSSWB instruction and their results. Clearly, the value of the original byte can be found by checking every fourth bit. The remaining steps are faily simple and have been adequately described above.

2.2. Case-Insensitive Version

The case-insensitive version requires that both the SearchStr and the TextBuf are converted to the same case before the characters are compared. Because the search string is small and will be used many times, an aligned copy is created. This is the only change in the initialization section. During the outer loop, the same technique is used to convert the current 8 characters of text to lower case before being compared to SearchStr[0] and SearchStr[1].

The only difference within the inner, scalar loop occurs before it checks the text character-by-character. It can read directly from the copy of SearchStr, but because the text is still mixed-case, it has to be converted before it can be used. To do this, eight characters of text are read, converted, and stored into a temporary buffer, and the inner loop reads from there. When all eight characters have been verified, another eight are read in and converted.

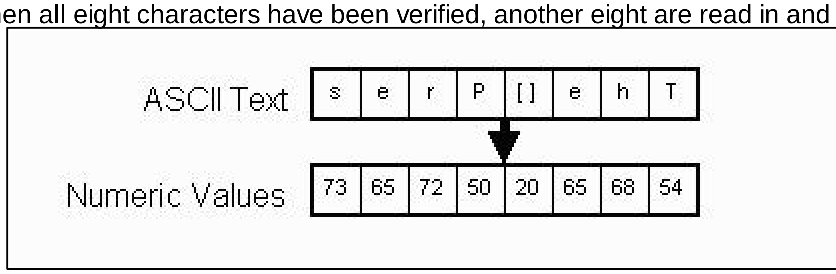

Figure 5. Sample ASCII Text and its numeric equivalent (in hex).

The sample text "The Pres" is compared against the upper and lower bounds for uppercase ASCII letters.

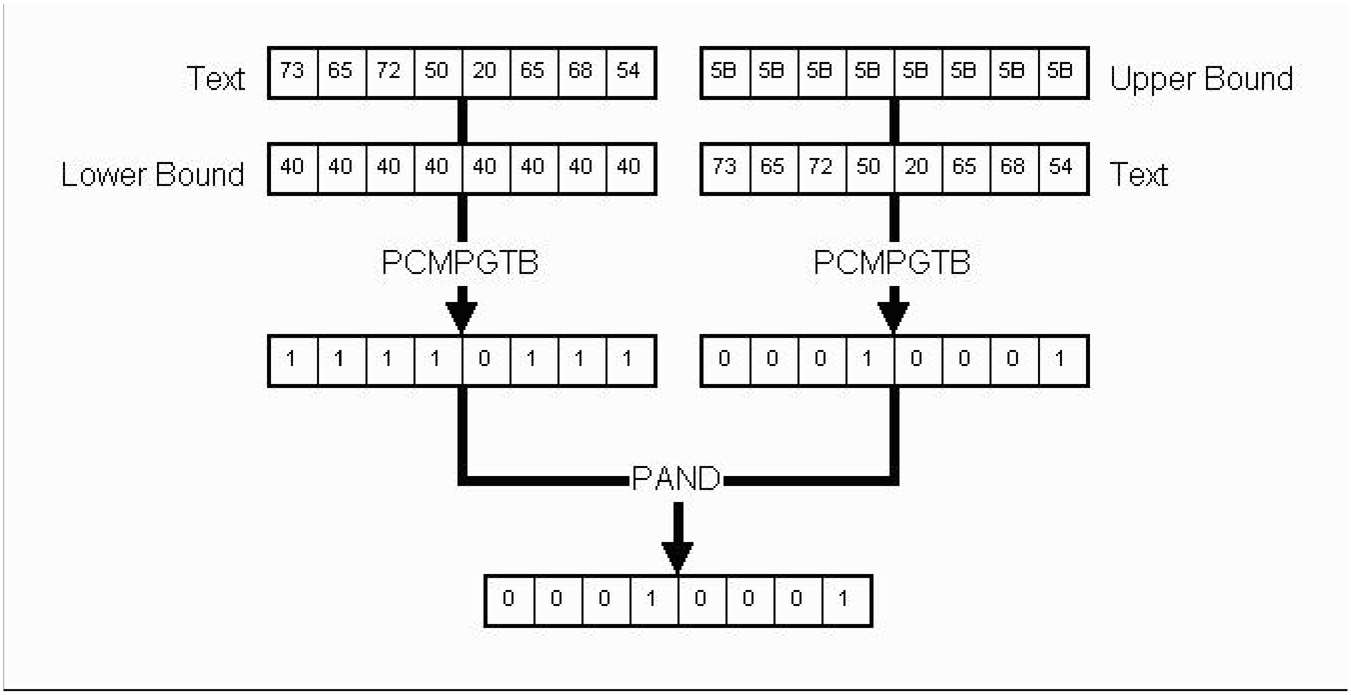

Figure 6. Comparing text against the upper and lower bounds for ASCII uppercase letters.

If a character is greater than the lower bound of 40h, and is lower than the upper bound of 5Bh (the actual compare tests whether the characters are greater than the upper bound, which is equivalent), then that byte will contain all ones at the end of the compares.

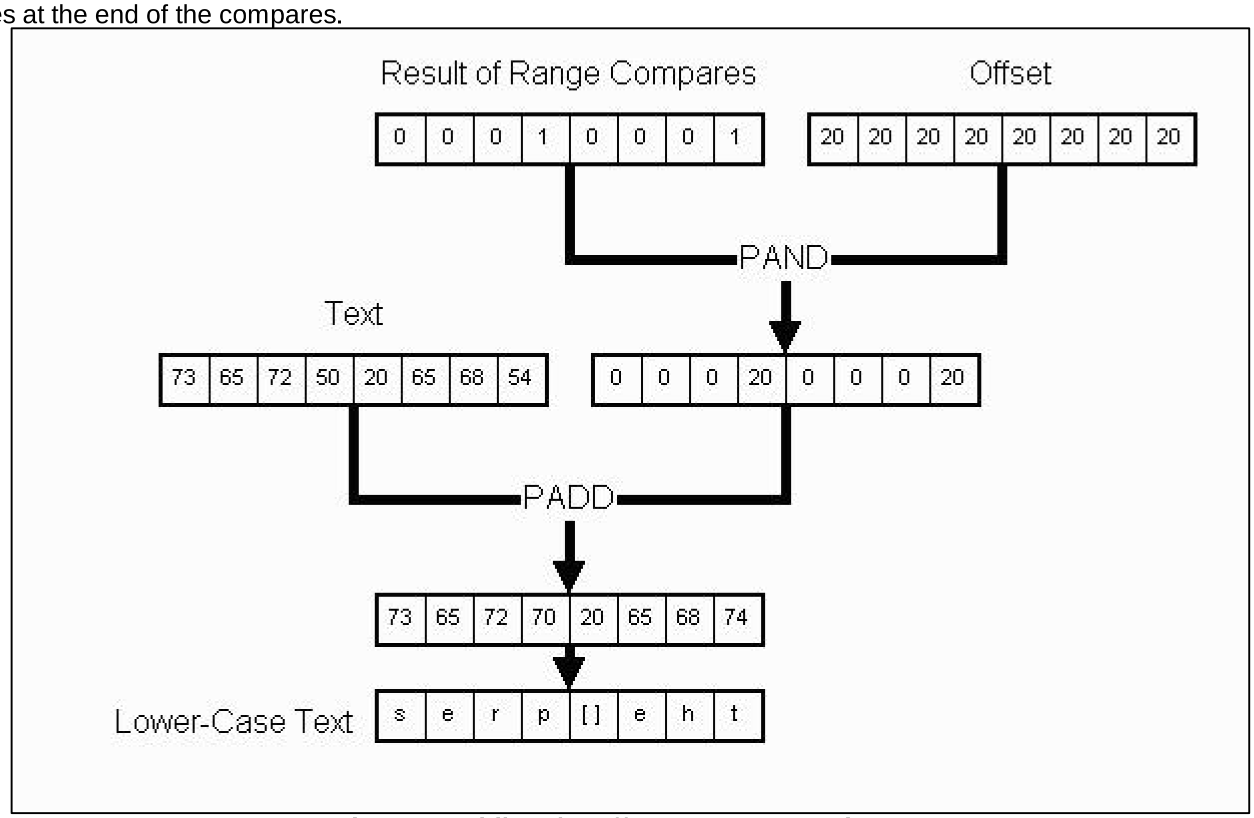

Figure 7. Adding the offset to uppercase letters.

ANDing the result of the compares with the 20h offset between 'A' and 'a' and adding that result to the original text converts any uppercase letters to lowercase letters. Lowercase letters and other special characters are unaffected.

3.0. PERFORMANCE GAINS

Performance measurements were taken on a 150MHz Pentium® Processor with MMX™ technology and a 233MHz Pentium® II Processor. Relative performance did not vary across processors; that is, the ratio of scalar to MMX technology was the same on both the Pentium and Pentium II processor.

The MMX™ technology enhanced string-find works 50% faster than the pure scalar string-find. Even though the MMX™ technology version works eight characters at a time, it is not eight times faster because of the additional overhead of the more complex algorithm. Also, because of this overhead, on very short searches (less than 50 characters) it may be slower than the scalar string-find. During longer searches, more time is spent in the outer loop, which is that part that MMX™ technology accelerated most.

The MMX™ technology case-insensitive string-find is 4.5X faster than the C-version. The perfomance improvement here is much greater than the case-sensitive version because MMX™ instructions can convert the case of eight characters at once, without branches.

4.0. CODE LISTINGS

The MMX™ functions are inline assembly with C wrappers, compiled with Microsoft Visual C++*, version 5.0.

4.1. Case-Sensitive

4.1.1. STRSTRMMX

```
__int64    mask=0x07;

char *strstrmmx(const char *TxtBuff, const char *SearchStr)
{
if (SearchStr[1] == '\0')              //Will not work with 1 character search strings
      strstr(TxtBuff, SearchStr);
__asm{
//mix2_strfind PROC C USES edi esi ebx ecx,
//            TxtBuff:PTR BYTE, TxtBuff_Size:DWORD, SearchStr:PTR BYTE, SearchStr_Size:DWORD, No_Case:DWORD

//Initialization
      mov     edi,SearchStr          ;save addr of searchstr
      mov     esi,TxtBuff            ;save addr of txtbuffer

      xor     ecx,ecx                ;zero ecx to indicate continue

      movq    mm0,dword ptr[edi]     ;load the first 8 char from searchstr

      movd    mm7,esi                ;alignment code
      punpcklbw mm0,mm0                   start the propagation of searchstr[0]

      punpcklwd mm0,mm0
      pand    mm7,mask               ;find offset of TxtBuff from 8 byte

      lea     edx,[edi+2]            ;save searchstr[2] address
      movq    mm1, mm0                    ;copy of SearchStr

      movq    mm4,dword ptr[esi]     ;copy the next 8 byts of text buffer into mm4
      punpckldq mm0,mm0                   finish propagating searchstr[0] into whole mmx reg

      punpckhdq mm1,mm1                   finish propagating searchstr[1] into whole mmx reg
      movq    mm2,mm0                ;save propagate searchstr[0] to mm2 for later recovery

      psllq   mm7,3
      pxor    mm5,mm5

      movq    mm3,mm1                ;save propagate searchstr[1] to mm3 for later recovery
      psllq   mm4,mm7                ;shift first 8 bytes by offset
```

```
        and     esi,0FFFFFFF8h          ;align TxtBuff pointer
                                ;end alignment code

MATCH_2_CHARS:

        pcmpeqb mm0,mm4                 ;compare 8 bytes of searchstr[0] to txtbuffer
        pcmpeqb mm1,mm4                 ;compare 8 bytes of searchstr[1] to txtbuffer

        movq    mm6,mm0                 ;copy of results of searchstr[0] compare
        psllq   mm0,8                   ;shift left 1 byte to line up with searchstr[1]

        por     mm0,mm5                 ;combine the current searchstr[0] with the last byte of the previous compare
        psrlq   mm6,56                  ;save the last byte of searchstr[0]

        pand    mm0,mm1                 ;compare searchstr[0] and searchstr[1]
        lea     edi,[esi+1]             ;copy textbuffer just in case we will do byte by byte compare

        packsswb mm0,mm0                ;reduce to 32bits
        push    ecx                     ;ecx indicates stop (1) or continue (0)

        ;1 penalty cycle on PPMT

        movd    eax,mm0                 ;copy ldword of quad compare into eax
        movq    mm5,mm6                 ;save last byte of searchstr[0] for next iteration

        test    eax,eax                 ;set flags
        jz      NO_MATCH_FOUND

FIND_MATCHES:
        test    al,1000b                ;does this byte have info if not then advance 4 bits for next byte
        jz      NEXT_BYTE               ;no match found for this byte

;scalar asm*****************************/
        mov     ecx,edi                 ;copy address of text buffer to ecx
        push    edx                     ;save searchstr[2] address cuz we are clobbering it

mainlupe:
        mov     bl,[edx]                ;copy searchstr[i] to bl
        inc     edx

        or      bl,bl                   ; if we've reached the end of search str, we've
        jz      short success           ; found the first matching substring

        mov     bh,[ecx]                ;copy textbuffer[i] to bh
        inc     ecx

        cmp     bl,bh                   ; characters match?
        je      short mainlupe

        ; failed comparison. recover pointer to searchstr+1
        ; and work on next byte

        pop     edx
        jmp     NEXT_BYTE

success:
        lea     eax,[edi-2]             ;since edi contains textbuffer+2 we must give proper address in buffer
        pop     ecx

        pop     ecx
        jmp     DONE

;************************************/

NEXT_BYTE:
        shr     eax,4                   ;shift right 4 bits
        inc     edi                     ;inc textbuffer offset

        cmp     eax,08h                 ;do we have any more info to process
        jae     FIND_MATCHES

NO_MATCH_FOUND:                         ;no match for the quad words
        add     esi,8                   ;advance the text buffer by 8
        pop     ecx                     ;load remaining size of buffer
```

```
        test    ecx,ecx                ;have we reached the end of the text buffer?
        jnz     DONE_NOT_FOUND          ;if ecx is not zero, quit

        movq    mm4,dword ptr[esi]      ;copy the next 8 byts of text buffer into mm4
        psubb   mm0,mm0                 ;zero out mm0

        movq    mm1,mm4                 ;copy text

        pcmpeqb mm1,mm0                 ;compare text with zero

        packsswb mm1,mm1                ;pack to 32bits
        movq    mm0,mm2                 ;copy SearchStr[0]

        movd    ecx,mm1                 ;store in ecx
        movq    mm1,mm3                 ;copy SearchStr[1]

        jmp     MATCH_2_CHARS

DONE_NOT_FOUND:
        xor     eax,eax                 ;no match found

DONE:
        emms
        }
}       //eax is returned
```

4.1.2. STRSTR

This is the strstr(char *text, char *string) function that is part of the standard C library included with Microsoft Visual C++, version 5.0, defined in <string.h>. Source code is not available.

4.2. Case-Insensitive

4.2.1. STRISTRMMX

```
__int64    mask=0x07,
        difference=     0x2020202020202020,
        low_bound=      0x4040404040404040,
        up_bound=       0x5B5B5B5B5B5B5B5B;
int EndUncapBuf;

char *stristrmmx(char *TxtBuff,char const *SearchStr)
{
if (SearchStr[1] == '\0')             //Will not work with 1 character search strings
        stristr(TxtBuff, SearchStr);
__asm{
//mix2_strfind PROC C USES edi esi ebx ecx,
//              TxtBuff:PTR BYTE, TxtBuff_Size:DWORD, SearchStr:PTR BYTE, SearchStr_Size:DWORD, No_Case:DWORD

//PREP:
        mov     edi,SearchStr           ;save addr of searchstr
        psubb   mm1,mm1                 ;zero mm1

        mov     esi,TxtBuff             ;save addr of txtbuffer
        xor     ecx,ecx                 ;ecx initialized to continue (0)

;capitalize the SearchStr
        xor     eax, eax                ;load with continue (0)
        mov     ebx, dword ptr Search   ;load pointer to copy of lowercase SearchStr
CAP_SEARCH:
        test    eax,eax
        jnz     END_CAP_SEARCH          ;if a null was found, end of string

        movq    mm6,[edi]               ;search string

        movq    mm5,[up_bound]
        movq    mm4,mm6                 ;copy search string

        pcmpgtb mm4,[low_bound]         ;char>low_bound
        movq    mm0,mm6                 ;copy search string
```

```
        pcmpgtb mm5,mm6                ;up_bound>char?

        pand    mm5,[difference]
        pand    mm5,mm4

        add     edi,8                  ;increment Search pointer
        pcmpeqb mm0,mm1                ;compare search with null

        paddb   mm6,mm5
        packsswb mm0,mm0               ;pack null compare to 32bits

        movq    [ebx],mm6              ;save to Search
        add     ebx,8

        movd    eax,mm0                ;store compacted null compare

        jmp     CAP_SEARCH             ;repeat if above 0

END_CAP_SEARCH:
        mov     edi,dword ptr Search   ;reload SearchStr

        movq    mm0,dword ptr[edi]     ;load the first 8 char from searchstr

        punpcklbw mm0,mm0              ;start the propagation of searchstr[0]

        punpcklwd mm0,mm0

        movq    mm1, mm0
        punpckldq mm0,mm0              ;finish propagating searchstr[0] into whole mmx reg

        lea     edx,[edi+2]            ;save searchstr[2] address
        punpckhdq mm1,mm1              ;finish propagating searchstr[1] into whole mmx reg

        movq    mm4,dword ptr[esi]     ;copy the next 8 byts of text buffer into mm4
        movq    mm2,mm0                ;save propagate searchstr[0] to mm2 for later recovery

        pxor    mm5,mm5
        movq    mm3,mm1                ;save propagate searchstr[1] to mm3 for later recovery

;alignment code
        movd    mm7,esi
        pand    mm7,mask               ;find offset of TxtBuff from 8 byte
        psllq   mm7,3                  ;multiply number of bytes by 8 to get number of bits to shift
        psllq   mm4,mm7                ;shift first 8 bytes by offset
        and     esi,0FFFFFFF8h         ;align TxtBuff pointer
;end alignment code


MATCH_1ST_CHAR:
;capitalize this qw
        movq    mm6,[up_bound]
        movq    mm7,mm4

        pcmpgtb mm7,[low_bound]        ;char>low_bound
        pcmpgtb mm6,mm4                ;up_bound>char?

        pand    mm6,[difference]

        pand    mm6,mm7

        paddb   mm4,mm6
;end cap
        pcmpeqb mm0,mm4                ;compare 8 bytes of searchstr[0] to txtbuffer
        pcmpeqb mm1,mm4                ;compare 8 bytes of searchstr[1] to txtbuffer

        movq    mm6,mm0
        psllq   mm0,8

        por     mm0,mm5
        psrlq   mm6,56

        pand    mm0,mm1                ;comp with 1st byte
        lea     edi,[esi+1]            ;copy textbuffer just in case we will do byte by byte compare

        packsswb mm0,mm0               ;reduce to 32bits
```

```
        push    ecx                     ;ecx stop (1) or continue (0)
;1 penalty cycle on PPMT

        movd    eax,mm0                 ;copy ldword of quad compare into eax
        movq    mm5,mm6

        test    eax,eax                 ;early out
        jz      NO_MATCH_FOUND

FIND_MATCHES:
        test    al,1000b                ;does this byte have info if not then advance 4 bits for next byte
        jz      NEXT_BYTE               ;no match found for this byte

;scalar asm*****************************
        push    edx                     ;save searchstr[1] address cuz we are clobbering it
        push    edi
uncap:
        movq  mm4,[edi]

        movq    mm0,[up_bound]
        movq    mm1,mm4

        pcmpgtb mm1,[low_bound]         ;char>low_bound
        pcmpgtb mm0,mm4                 ;up_bound>char?

        pand    mm0,[difference]
        pand    mm0,mm1

        mov     ecx,[UncapBuf]          ;copy address of text buffer to ecx
        paddb   mm4,mm0

        movq    [ecx],mm4               ;save in CapBuffer
;uncapped


mainlupe:

        mov     bl,[edx]                ;copy searchstr[i] to bl
        inc     edx

        or      bl,bl                   ; if we've reached the end of search str, we've
        jz      short success           ; found the first matching substring

        mov     bh,[ecx]                ;copy textbuffer[i] to bh
        inc     ecx

        cmp     bl,bh                   ; characters match?
        jne     short mainlupefailed

;check to see if done with current 8 bytes
        cmp     ecx,[EndUncapBuf]
        jne     mainlupe

        add     edi,8                   ;add 8 to position in text buffer
        jmp     uncap                   ;load next 8 chars and uncap

success:
        pop     edi
        pop     ecx

        lea     eax,[edi-2]             ;since edi contains textbuffer+2 we must give proper address in buffer
        pop     ecx

        jmp     DONE

;************************************
mainlupefailed:
    ; failed comparison. recover pointer to searchstr+1
    ; and work on next byte
        pop     edi
        pop     edx

NEXT_BYTE:
        shr     eax,4                   ;shift right 4 bits
        inc     edi                     ;inc textbuffer offset
```

```
        cmp     eax,08h                 ;do we have any more info to process
        jae     FIND_MATCHES

NO_MATCH_FOUND:                         ;no match for the quad words
        add     esi,8                   ;advance the text buffer by 8
        pop     ecx                     ;load stop/cont (1/0)

        test    ecx,ecx
        jnz     DONE_NOT_FOUND

        movq    mm4,dword ptr[esi]      ;copy the next 8 byts of text buffer into mm4
        psubb   mm0,mm0                 ;zero out mm0

        movq    mm1,mm4                 ;copy text

        pcmpeqb mm1,mm0                 ;compare text with zero
        movq    mm0,mm2                 ;copy SearchStr[0]

        packsswb mm1,mm1                ;pack to 32bits

        movd    ecx,mm1                 ;store in ecx
        movq    mm1,mm3                 ;copy SearchStr[1]

        jmp     MATCH_1ST_CHAR          ;have we reached the end of the text buffer?

DONE_NOT_FOUND:
        xor     eax,eax                 ;no match found

DONE:
        emms
        }
}       //eax is returned
```

4.2.2. STRISTR

This code was obtained from the public domain collection SNIPPETS . The file Stristr.C was modified, primarily to remove two very slow strlen() calls to get the lengths of the strings. It now checks for null terminating characters and is roughly 5X faster than the original SNIPPETS function.

```
/* +++Date last modified: 4-Aug-1997 */

/*
** Designation:  StriStr
**
** Call syntax:  char *stristr(char *String, char *Pattern)
**
** Description:  This function is an ANSI version of strstr() with
**               case insensitivity.
**
** Return item:  char *pointer if Pattern is found in String, else
**               pointer to 0
**
** Rev History:  16/07/97  Greg Thayer  Optimized (and possibly de-ANSI-fied)
**               07/04/95  Bob Stout    ANSI-fy
**               02/03/94  Fred Cole    Original
**
** Hereby donated to public domain.
*/


__inline char toupper(char c)
{
return ((c>(char)0x60) && (c<(char)0x7b))? c-0x20:c;
}


char *stristr(const char *String, const char *Pattern)
{
     char *pptr, *sptr, *start;

     for (start = (char *)String; *start != '\0'; start++)
     {
```

```
        /* find start of pattern in string */
        for (;((*start!='\0') && (toupper(*start) != toupper(*Pattern))); start++);

        pptr = (char *)Pattern;
        sptr = (char *)start;

        while (toupper(*sptr) == toupper(*pptr))
        {
            sptr++;
            pptr++;

            /* if end of pattern then pattern was found */
            if ('\0' == *pptr)
                return (start);
        }
    }
    return(0);
}
```

* Other brands and names are the property of their respective owners.