Application Note 605
Data Pointer Decrement Feature Simplifies Copy Operation for Overlapping Memory Buffers

www.maxim-ic.com

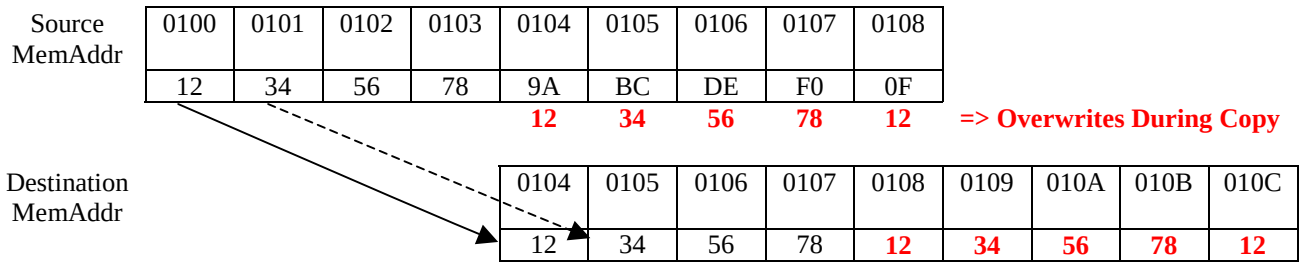
OVERVIEW

One of the essential operations of any microcontroller is the ability to store and retrieve data to/from memory. The MOVX operation provides one facility for doing this on the 8051 architecture. Applications often require the microcontroller to copy and move blocks of data memory around within its MOVX address space. This memory transfer operation is quite simple when the source and destination address ranges do not overlap, i.e., an iterative read/write loop. When the ranges overlap, however, the process requires some intelligence to avoid overwriting (corrupting) the original data before it is transferred. This application note will offer two possible solutions for transferring data between source and destination buffers, which overlap, and explain how Dallas' data pointer decrement feature simplifies the solution.

OVERLAPPING MEMORY PROBLEM

Most generic memory copy routines do not determine whether the source and destination copy ranges overlap. Without making this assessment prior to the execution of the copy routine, bytes copied to an intended destination range that overlaps the original source range can overwrite and corrupt the original data. A simple illustration of how this might occur is given in Figure 1. As can be seen, the destination address range begins at address = 0104h, which also happens to be an address within the original source byte-array range. As alluded to earlier, the standard **memcpy()** routine in this situation would not generate the desired destination data array. When data integrity must be maintained for such a transfer, the **memmove()** operation is generally used to ensure that bytes in the source array are not overwritten when copying to a destination.

Figure 1. **PROBLEM: OVERLAPPING MEMORY COPY**



POSSIBLE SOLUTIONS

With some observation, one can see that the overwrites that occur to the source array (prior to the copy) can be avoided in a couple of ways: 1) determine the overlap and transfer **first** those bytes in the source buffer that overlap the desired destination buffer or 2) determine the overlap and transfer bytes from the source buffer to the destination buffer in reverse order. These two solutions are shown in Figures 2 and 3. Note that an overlap in the opposite direction (copying source array to destination lower in memory) poses no problem for the standard copy loop that transfers data in ascending address order.

Given the two visuals below (Figures 2 and 3), one should also be able to see that Solution #1 suffers additional overhead in order to calculate, store, and pass different source, destination, and length variables for multiple copy operations, whereas the second solution must do this only once.

Figure 2. **OVERLAPPING MEMORY COPY SOLUTION #1**

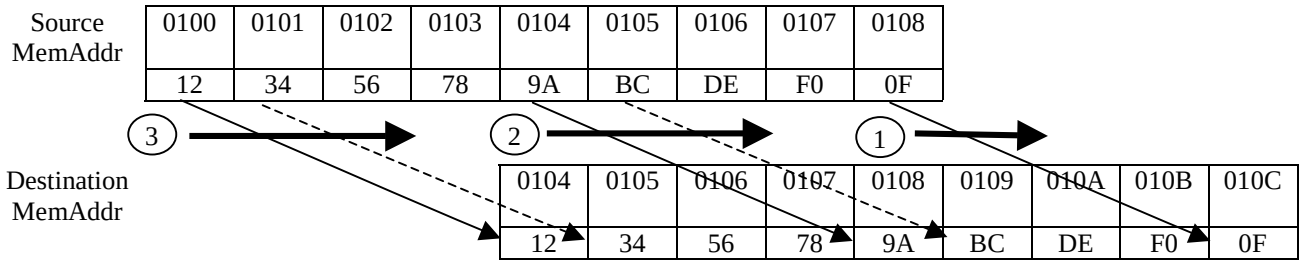
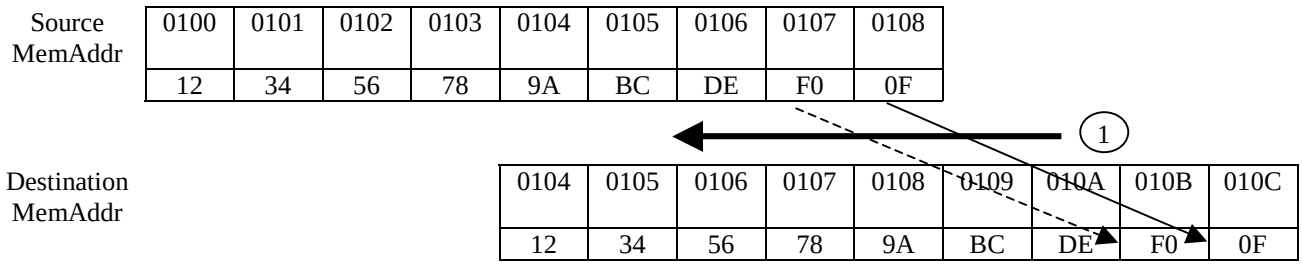


Figure 3. **OVERLAPPING MEMORY COPY SOLUTION #2**



DALLAS HARDWARE SIMPLIFIES SOLUTION #2

Many Dallas microcontroller products (list provided in Appendix A) implement an Increment/Decrement (IDx) bit for each available data pointer to designate whether the 'INC DPTR' instruction will increment or decrement the active data pointer. Using the data pointer decrement feature, Solution #2 is particularly easy to implement on Dallas products, allowing a linear transfer and minimizing execution time.

In order to take advantage of the data pointer decrement feature, the application code first must determine if and how the source and destination ranges overlap, a task that would be executed even if the data pointer decrement feature weren't available. When a potentially problematic source/destination buffer overlap is detected, the data pointers are placed at the end of the respective source/destination copy ranges and the IDx bits are configured to enable the data pointer decrement mode. Example code for solution #2 is provided below. Note that the DPTR toggling ('INC DPS') and increment/decrement ('INC DPTR') functions have been included in the code for the purpose of understanding only and can be removed if the respective auto-toggle and/or auto-increment/decrement bits have been set.

EXAMPLE APPLICATION CODE FOR SOLUTION #2

```
;------------------------------------------------------------
; Example memmove code which tests for overlapping src/dest
; address ranges before the copy operation.  Code assumes that
; length>0, the copy is being done within a single 64kB xdata
; bank, and DPTR0 used otherwise (i.e. only DPTR0 saved).
;
; input:    R6:R7  (pointer to destination)
;           R4:R5  (pointer to source)
;           R2:R3  (length)
; uses:     R1     (src-dest pointer delta temp store)
;           R0:R1  (length-1 later)
;------------------------------------------------------------
memmove:
      push  dps               ;save DPTR0
      push  dpl
      push  dph
;-------< DETECT PROBLEMATIC SRC/DEST BUFFER OVERLAP>---------
      clr   c                 ;
      mov   a, r5             ; check delta between
      subb  a, r7             ; src/dest pointers
      mov   r1, a             ;
      mov   a, r4             ;
      subb  a, r6             ;
      jnc   domemcpy          ; dest pointer <
                              ;  src pointer addr?
      xch   a, r1             ; NO
      add   a, r3             ; check delta vs.
      xch   a, r1             ; length
      addc  a, r2
      jnc   domemcpy          ; length < delta?
      xrl   a, r1             ; NO
      jz    domemcpy          ; length = delta?
;-----< DATA POINTER DECREMENT MODE / ADJUST POINTERS >-------
overlap:                      ; NO.
      mov   dps, #0C0h        ; ID1,ID0=11b (dec)
      mov   a, r3             ;
      add   a, #0ffh          ; add (-1) to length
      mov   r1, a             ; prior to adding to
      mov   a, r2             ; original src/dest
      jc    lenless1          ; start pointers
      dec   a
lenless1:
      mov   r0, a             ; R0:R1 = (length-1)
      mov   a, r5             ; src pointer
      add   a, r1             ; starts at the end
      mov   r5, a             ; =src+length-1
      mov   a, r4
      addc  a, r0
      mov   r4, a
      mov   a, r7             ; dest pointer
      add   a, r1             ; starts at the end
      mov   r7, a             ; =dest+length-1
      mov   a, r6
      addc  a, r0
      mov   r6, a
;------------------< NORMAL COPY LOOP >------------------
domemcpy:
      mov   a, r3
      jz    nomod             ; inc msb if lsb<>00h
```

```
      inc   r2                ; for proper loop cntrl
nomod:
      mov   dph, r4           ; load source
      mov   dpl, r5
      mov   dph1, r6          ; load dest
      mov   dpl1, r7
copyloop:
      movx  a, @dptr          ; read
      inc   dptr              ; inc/dec src pointer
      inc   dps               ; select dest
      movx  @dptr, a          ; write
      inc   dptr              ; inc/dec dest pointer
      inc   dps               ; select src
      djnz  r3, copyloop
      djnz  r2, copyloop
      pop   dps
      pop   dph
      pop   dpl
      ret
```

APPENDIX A: DATA POINTER FEATURES

HIGH-SPEED µC	DATA POINTERS	INCREMENT/ DECREMENT SELECTION BITS	AUTO-TOGGLE FEATURE	AUTO-INCREMENT/ DECREMENT FEATURE
DS80C310	2	No	No	No
DS80C320/323	2	No	No	No
DS87C520/530	2	No	No	No
DS87C550	2	Yes	Yes	No
DS80C390	2	Yes	Yes	No
DS89C420	2	Yes	Yes	Yes
DS80C400	4	Yes	Yes	Yes