



APPLICATION NOTE 3770

## Table Operations for the MAXQ Architecture

*Abstract: The MAXQ microcontroller is a Harvard machine. One should not, however, assume that MAXQ microcontrollers suffer from a common limitation of many Harvard machines that prohibits access to constants stored in code space. On the contrary, tools built into every MAXQ device make such table lookups easy. This application note describes how to perform efficient table operations in the MAXQ microcontroller.*

### Introduction

The [MAXQ](#) architecture describes a powerful, single-cycle RISC microcontroller based on the classic Harvard machine. A Harvard machine differs from the more commonly seen Von Neumann machine in an important design element: the Harvard machine's instructions and data are carried on separate busses. Because there is no contention for a single data bus, MAXQ instructions can execute in only a single cycle. The same operations would require multiple cycles on a traditional Von Neumann architecture.

The rigid separation between data and code in Harvard machines, however, presents its own set of challenges. Storing data tables in code space, a technique common in Von Neumann machines, is problematic for classical Harvard machines. Since only one operation can occur in a single cycle on a given bus, the CPU core cannot fetch an instruction on the code memory bus and fetch a memory operand from a data table in code space in the same cycle.

As a Harvard machine, one might assume that MAXQ microcontrollers also prohibit storage of data elements in code space. But because of ROM tools built into every MAXQ device, such table lookups are actually easy.

### Table Lookup in Code Space

Reading a value from a MAXQ table in code space appears to be simple. However, programmers unfamiliar with the MAXQ architecture can become frustrated on their first attempt to do so.

```
IncorrectTableLookup:
    move    dp[0], #w:StartOfTable
    move    acc, @dp[0]
    .
    .
    .
    ret
.
.
.
StartOfTable:
    dc16    01234h
    dc16    05678h
    dc16    098abh
    dc16    0cdefh
```

This above code will assemble perfectly, but after the second instruction the accumulator will almost certainly *not* contain 0x1234. The reason is simple. Unlike a Von Neumann machine in which there is only a single

memory space and an instruction will take as many cycles as needed to complete, the MAXQ's `move <reg>, @dp [0]` instruction implicitly accesses data space and completes in a single cycle with an instruction fetch from code space and a read from data space. The value loaded into the accumulator is whatever value was in data space *at the same offset as StartOfTable in code space*.

At first this problem seems intractable. After all, accesses to code space require a certain amount of time; the CPU core cannot squeeze two memory accesses into one clock cycle, even if the architecture permitted it. In the MAXQ architecture, however, small details about how the microcontroller maps physical memory blocks into the various memory spaces—and a few routines in the utility ROM—solve the problem.

First, in the MAXQ architecture the mapping of physical memory blocks into code and data spaces is *not* fixed, but changes depending on what physical memory block is being accessed. In general, programmers writing code to run in the flash memory space of most MAXQ microcontrollers will link their software to start at location 0 in code space. They will assume that RAM starts at location 0 in data space, and so it does.

But the MAXQ microcontroller has another piece of physical memory as well, the utility ROM. The utility ROM resides at location 0x8000 in code space in all MAXQ microcontrollers. User code can call routines in the 0x8000 page of the utility ROM to perform certain functions. Furthermore, as long as execution is being performed in the utility ROM, *user code memory is remapped to a new location in data space*.

When executing out of utility ROM, data RAM continues to be accessible at location 0x0000 in data space, but code memory is remapped to location 0x8000 in data space. Since code flash now appears in data space, the code running from the utility ROM can access information stored in user code *as if it were data*. Utility ROM functions are provided that simply read a value indirectly through the pointer registers and return.

Consequently, the routine given above would change somewhat:

```
BetterTableLookup:
    move    dp[0], #w:StartOfTable + 08000h
    call    UtilityROMGetDP0
    .
    .
    .
    ret
.
.
.
StartOfTable:
    dc16    01234h
    dc16    05678h
    dc16    098abh
    dc16    0cdefh
```

In this example, the address to be read is adjusted to reflect the location to which flash is mapped during utility ROM execution, and is then loaded into DP[0]. Rather than trying to read the data directly, a call is made to the utility ROM routine. Of course, instead of one cycle used in the direct data read, this operation takes four cycles: two cycles for a long call, one cycle to perform the read, and one cycle for the return operation.

A greater problem with this code example is that it will not assemble! The label `UtilityROMGetDP0` is not defined, and for a good reason: the utility routines are not in the same place from one MAXQ microcontroller to another. The utility routines are, in fact, not guaranteed to remain in the same place from one version of a particular MAXQ device to another!

To resolve this issue, every MAXQ microcontroller's utility ROM contains an address table for the utility functions, and a pointer to the table at a well-known location: 0x800D. Specifically, the utility ROM contains the following code:

```
org    0800Dh
dw     UtilityFunctionTable
.
```

```

.
.
UtilityFunctionTable:
.
.
.
    dw      GetDP0
    dw      GetDP0Inc
    dw      GetDP0Dec
    dw      GetDP1
    dw      GetDP1Inc
    dw      GetDP1Dec
    dw      GetBP
    dw      GetBPInc
    dw      GetBPDec

```

Note first, that the utility function table consists of *addresses*, not instructions. That is, the application programmer must retrieve the address and **call** it, not simply jump into the table. Second, note that the first memory reference function may not be the first entry in the table. Since each MAXQ microcontroller can contain different types and amounts of memory and different peripherals, each device will likely contain a different list of functions, at different relative offsets within the table.

Each of the three pointer registers has three associated functions, for a total of nine table lookup functions. The first function for each pointer register simply retrieves the value at the given address, while the latter two use the post-increment and post-decrement forms of the indirect load, respectively. In each case, the retrieved data is loaded into the GR register.

Now, our code looks like this:

```

CorrectTableLookup:
    move    dp[0], #0800Dh ; Point to pointer to function table
    move    acc, @dp[0]   ; acc now has pointer to ftable
    add     #3             ; For 2000, GetDP0
    move    dp[0], acc     ; Load ptr + offset to dp0
    move    a[1], @dp[0]  ; Get address of GetDP0 into A1
    move    dp[0], #StartOfTable + 08000h
    call    a[1]          ; This will call GetDP0, finally!
.
.
.
ret
.
.
.
StartOfTable:
    dc16    01234h
    dc16    05678h
    dc16    098abh
    dc16    0cdefh

```

Notice that once the address of the GetDP0 routine has been found, it can be stored away and reused. The first five instructions above only need to be performed once; after that, each table fetch takes only three cycles: the call, the read (executing from utility ROM), and the return (also running from utility ROM.)

## Copying Tables from Flash to RAM

A method for moving an entire table from flash to RAM can now be constructed from the table read functions. If

the destination address is given in BP, for example, one method would be:

SlowTableMove:

```
    move    dp[0], #0800Dh ; Point to pointer to function table
    move    acc, @dp[0]    ; acc now has pointer to ftable
    add     #4              ; For 2000, GetDP0Inc
    move    dp[0], acc     ; Load ptr + offset to dp0
    move    a[1], @dp[0]   ; Get address of GetDP0 into A1
    move    dp[0], #StartOfTable + 08000h
    move    bp, #RAMDest   ; Set this label to desired dest
    move    offs, #0ffh    ; Pre-decremented offset
    move    lc[0], #4      ; Move four words
```

TableMoveLoop:

```
    move    dp[0], dp[0]   ; Set source pointer
    call    a[1]           ; This will call GetDP0inc
    move    @bp[++offs], gr ; Store retrieved word to dest
    djnz   lc[0], TableMoveLoop
    .
    .
    .
    ret
```

StartOfTable:

```
    dc16    01234h
    dc16    05678h
    dc16    098abh
    dc16    0cdefh
```

As before, the first five instructions need only be executed once. After that, the table move can be performed as many times as needed, and the `GetDP0inc` subroutine address will remain in A1. The table move will require six cycles per iteration, plus setup overhead.

The `move dp[0], dp[0]` instruction is required due to a quirk in the MAXQ architecture. As there is only a single address bus associated with data space, it must be set up at least one cycle ahead of any read performed on data space. In the table movement loop, a read is performed at the address given by DP[0], and then the write address is placed on the bus. But without the `move dp[0], dp[0]` instruction, the write address would *still* be on the bus when it is the time to read the next location in the table. By inserting this apparently null instruction, the source operand address bus is refreshed in anticipation of the next read event.

There is, however, a better way to accomplish this task. The utility ROM includes a `copyBuffer` routine that performs this same function, but in fewer cycles. The `copyBuffer` routine is listed immediately below the table lookup routines in the utility ROM.

FasterTableMove:

```
    move    dp[0], #0800Dh ; Point to pointer to function table
    move    acc, @dp[0]    ; acc now has pointer to ftable
    add     #12            ; For 2000, copyBuffer
    move    dp[0], acc     ; Load ptr + offset to dp0
    move    a[1], @dp[0]   ; Get address of GetDP0 into A1
    move    dp[0], #StartOfTable + 08000h
    move    bp, #RAMDest   ; Set this label to desired dest
    move    offs, #0       ; No need to pre-decrement offset
    move    lc[0], #4      ; Move four words
    call    a[1]           ; This will call copyBuffer
```

```
    .
    .
    .
```

```

        ret
.
.
.
StartOfTable:
        dc16    01234h
        dc16    05678h
        dc16    098abh
        dc16    0cdefh

```

This `copyBuffer` routine reduces the cycle count per iteration to three, saving approximately half the time over the previous method. When the `copyBuffer` routine returns, `LC[0]` will be zero, and the `OFFS` register will point to the next location after the last destination location written. Because `OFFS` is an 8-bit register, tables up to 256 words can be copied in this way.

## An Example: String Output

A common task in many microcontroller-based projects is to output one of a number of canned messages to a console. Often, each message is given a number and a common routine must convert the number to the text of the message.

A commonly-used technique for this task is to zero-terminate each message string, `.` and provide a table that converts message numbers into the addresses at which each individual string resides. This technique is reliable and quick, but it means that two data structures must be established: the table of addresses and the strings themselves. A second technique simply places the zero-terminated strings into one large, contiguous memory space and searches linearly. While economical, this method is time-consuming at execution time because *every character* up to the intended string must be searched before the output begins.

A useful compromise uses *length-delimiting* instead of zero-delimiting. In this technique, the length of each string is given first, followed by the actual bytes of the message. In this way, unused messages can be quickly skipped, and the table itself is no longer than in the zero-delimited case. The only limitation to this compromise technique is that each string in the table is restricted to no more than 255 characters.

```

;
; Output String
;
; Enter with ACC=an index value (one based) indicating which
; string to output.
;
; On exit, LC0=0, DPC=0, ACC, A1, A2, DP0 used.
;
output_string:
        move    lc[0], acc                ;Set LC0 to index of string
move    dpc, #4                          ;Set DP0 to word mode
move    dp[0], #800dh                    ;Point to table of pointers
move    acc, @dp[0]                      ;Get address of table
add     #3                                ;Offset to GETDP0 routine
move    dp[0], acc                        ;Load pointer to table
move    a[1], @dp[0]++                   ;Get GETDP0
move    a[2], @dp[0]                     ;Get GETDP0INC
move    dpc, #0                          ;Set DP0 to byte mode
move    dp[0], #string_table + 8000h

str_search_loop:
call    a[1]                             ;Get a string length
djnz   lc[0], next_str                   ;If not this string, go to next
move    lc[0], gr                         ;Otherwise, put len in LC0
move    acc, @dp[0]++                    ;...and point past length

```

```

out_loop:
call    a[2]                ;Get a char and bump pointer
call    char_out           ;Output the character
djnz    lc[0], out_loop    ;If more characters, loop
ret     ;Otherwise, we're done.

next_str:
move    acc, gr            ;GR contains len of this string
add     dp[0]              ;Add current ptr to current len...
move    dp[0], acc         ;...to create a new pointer
jump    str_search_loop    ;Jump back and test index again

;
; Each entry in the string table begins with the string length
; followed by the string characters.
;
string_table:
dc8     string1 - string_table
dc8     "This is the first string."
string1:
dc8     string2 - string1
dc8     "This is a second example of a string"
string2:
dc8     string3 - string2
dc8     "A third string."
string3:
dc8     string4 - string3
dc8     "Finally, a fourth string in the array!!!"
string4:

```

---

Application Note 3770: [www.maxim-ic.com/an3770](http://www.maxim-ic.com/an3770)

### More Information

For technical support: [www.maxim-ic.com/support](http://www.maxim-ic.com/support)

For samples: [www.maxim-ic.com/samples](http://www.maxim-ic.com/samples)

Other questions and comments: [www.maxim-ic.com/contact](http://www.maxim-ic.com/contact)

---

### Automatic Updates

Would you like to be automatically notified when new application notes are published in your areas of interest?  
[Sign up for EE-Mail.](#)

---

### Related Parts

MAXQ2000: [QuickView](#) -- [Full \(PDF\) Data Sheet](#) -- [Free Samples](#)

MAXQ3210: [QuickView](#) -- [Full \(PDF\) Data Sheet](#) -- [Free Samples](#)

MAXQ3212: [QuickView](#) -- [Full \(PDF\) Data Sheet](#) -- [Free Samples](#)

MAXQ7665: [QuickView](#) -- [Full \(PDF\) Data Sheet](#)

AN3770, AN 3770, APP3770, Appnote3770, Appnote 3770

Copyright © by Maxim Integrated Products

Additional legal notices: [www.maxim-ic.com/legal](http://www.maxim-ic.com/legal)