Keywords: low-power,OIS,optical image stabilization,camera

APPLICATION NOTE 4458

Executing Application Code from RAM on the MAXQ8913 Microcontroller

Sep 01, 2009

Abstract: The Harvard memory map architecture used by the MAXQ8913 and other MAXQ® microcontrollers provides users with the flexibility to map different physical memory segments (such as the data SRAM) into either program or data memory space, as needed. Under certain circumstances, executing portions of an application from data SRAM can provide a performance boost and reduce power consumption. These benefits come at the cost of additional application complexity.

Overview

The MAXQ8913, like many other MAXQ microcontrollers, includes an internal SRAM-based data memory region which can be mapped either into data memory space or, optionally, into program memory space. The internal SRAM is generally used as data memory, with the bulk of code execution occurring in the program flash or masked ROM. However, under certain circumstances it can be useful for an application to execute a limited portion of its code from the internal SRAM.

This application note explains how to configure and load assembly-based code so that it will execute properly from the internal SRAM. The advantages and disadvantages of this execution are discussed. Demonstration code for this application note is written for the MAXQ8913, using the assembly-based MAX-IDE environment. The code and project files for the demo application covered in this application note are available for download.

While the code discussed in this application note is targeted specifically for the MAXQ8913 microcontroller, the principles and techniques shown here apply equally well to any other MAXQ-based microcontroller with internal SRAM that can be mapped into program space. Other MAXQ microcontrollers that can execute code in this manner include the MAXQ2000, the MAXQ2010, and the MAXQ3210/MAXQ3212.

This code will run properly on any MAXQ8913-based hardware that provides a serial-port interface (RS-232 or USB-to-serial) for the MAXQ8913's serial port 0. Output from the demonstration code can be viewed by connecting a terminal emulator to the serial port at 9600 baud, 8 data bits, 1 stop bit, no parity.

The latest installation package and documentation for the MAX-IDE environment are available for free download.

- MAX-IDE installation
- MAXQ Core Assembly Guide
- Development Tools Guide

Advantages of Executing Code from RAM

Normally, most application code on MAXQ microcontrollers is set up to execute from the main program space, which is usually implemented using a large internal flash memory or (for masked-ROM devices) a customer-specific application ROM. The main program space is nonvolatile, so it makes sense to store the application code there in most instances. The internal SRAM is used to store variables, a software stack, and similar data that do not need to be saved when the device is powered off.

However, for certain applications there are advantages to executing some code from the data SRAM.

Reduced Power Consumption

In most MAXQ microcontrollers, supply current is reduced when executing code from internal SRAM (or from the Utility ROM) as opposed to the program flash. This power savings occurs because the flash can be powered down dynamically when it is not being accessed. If an application typically spends most of its active time executing a very small amount of code, executing that code from SRAM can dramatically reduce overall power consumption.

Direct Memory Access to Main Program Space

Normally, code executing from the main program flash cannot directly read data that is also stored in the main program flash. This type of data can include constant strings and data tables included with the application data. To read this data, the application must call special-purpose data transfer functions in the Utility ROM. Executing code from the RAM bypasses this restriction and allows data contained in the flash to be read directly using the standard data pointers. This speeds up access. If a small algorithm spends a large amount of time traversing lookup tables or other constant data stored in the flash, executing the algorithm from RAM allows the operation to be completed in a shorter amount of time.

Entire Flash Memory Can Be Rewritten

The Utility ROM on the MAXQ8913, like most flash-based MAXQ microcontrollers, contains standard functions that allow the program flash to be erased and rewritten under application control. This process allows the implementation of user loaders that reload part, or all, of the application using a user-specified interface (such as the serial port, SPI, or I^{2}C). However, if the user loader code is contained in flash, it cannot erase or rewrite the part of the flash that it occupies. Executing user loader code from RAM allows the entire flash program space to be erased and rewritten with new code, including the user loader itself.

Disadvantages of Executing Code from RAM

There are also disadvantages and restrictions that apply when executing application code from the RAM. Some of the disadvantages have work-arounds, while others are inherent in the MAXQ architecture.

Limited Code Space

The RAM is typically much smaller than the program flash, which means that only a small amount of code can be executed from the RAM at any given time. It is possible, however, to run one routine from the RAM, erase it and load a second routine, run the second routine, and so on.

Code Must Be Mirrored

Before the code can be executed from the RAM, it must be copied to the RAM. This process requires time and code space to implement. Additionally, the code must be copied from somewhere, so in effect the code is stored twice: once in the flash or program ROM, and once in the RAM. Even though the code is not intended to be executed from the flash, it must still be stored there, thus consuming additional space.

RAM Cannot Be Directly Accessed

When executing code from the internal RAM, the RAM is no longer visible in the data memory space. This means that data pointers cannot be used to read from, or write to, RAM locations directly. It is possible to work around this restriction in the same manner as application code running from the flash. Use the Utility ROM data transfer functions (UROM_moveDP0 and similar functions) to read from the RAM and, by writing similar functions in the flash, perform an indirect write to the RAM. However, this work-around takes additional time and application space.

Assembling Code to Execute from RAM

When writing application code that will be executed from the data RAM, one major factor must be understood. Each word of code will be assembled at one address and loaded into the flash at that address, but it will be executed in RAM at a different address. For example, if a piece of application code is loaded into the flash starting at program word address 0100h and is copied to the RAM starting at data word address 0100h, it is not possible to jump to address 0100h to execute the code in RAM. Address 0100h is still the address of the code in flash. The address of the code in RAM in program space is its data memory address plus an offset of A000h, as shown in **Figure 1** below.

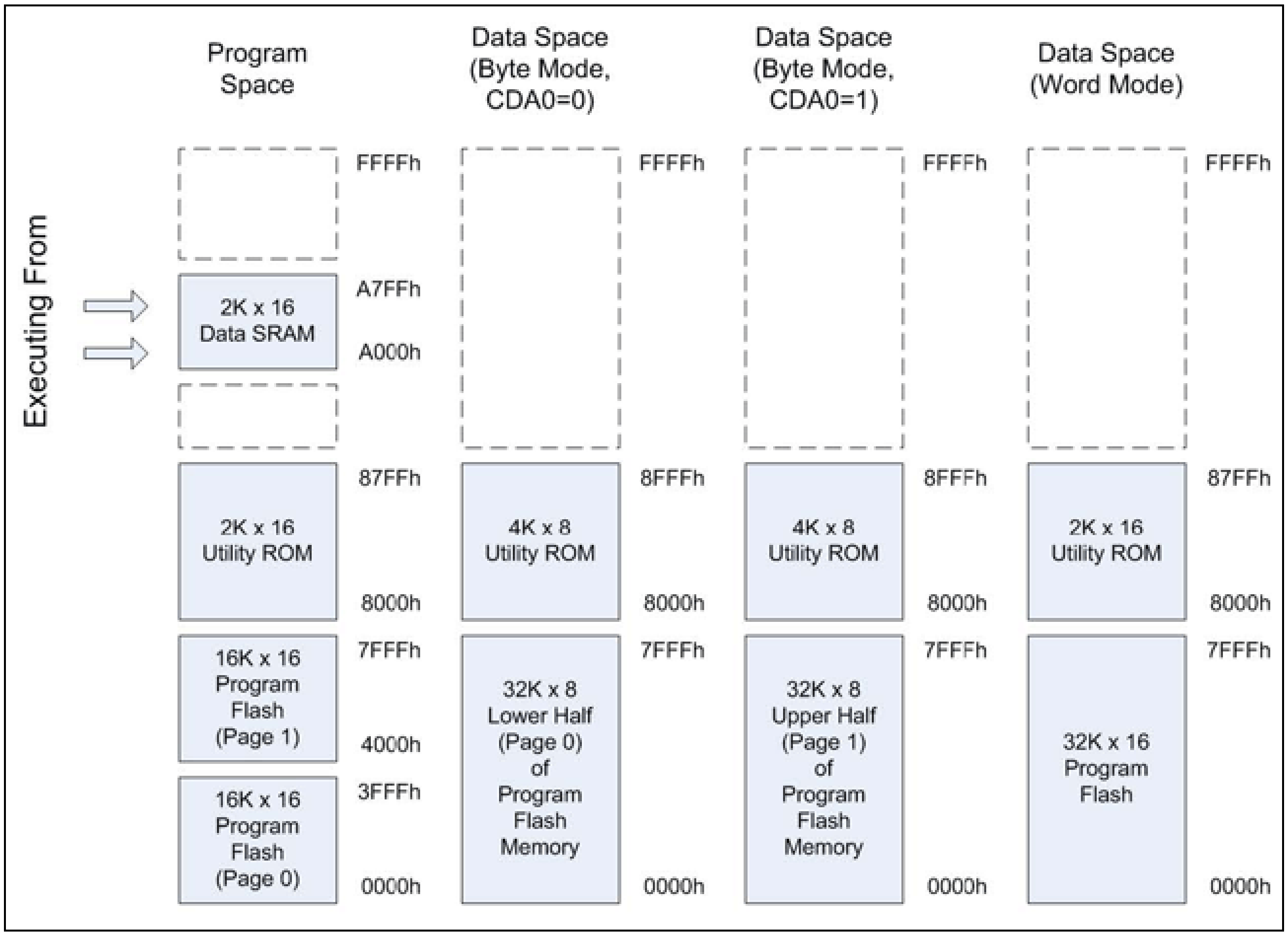

Program Space
Data Space (Byte Mode, CDA0=0)
Data Space (Byte Mode, CDA0=1)
Data Space (Word Mode)
Executing From
FFFFh
A7FFh
A000h
2K x 16 Data SRAM
87FFh
8FFFh
2K x 16 Utility ROM
4K x 8 Utility ROM
8000h
7FFFh
16K x 16 Program Flash (Page 1)
4000h
3FFFh
16K x 16 Program Flash (Page 0)
32K x 8 Lower Half (Page 0) of Program Flash Memory
32K x 8 Upper Half (Page 1) of Program Flash Memory
32K x 16 Program Flash
0000h

Figure 1. Memory mapping for the MAXQ8913 when executing code from RAM.

To execute the application code that was copied to the RAM at data memory address 0100h, you must jump to program address A100h.

Executing the code from RAM causes a difficulty for the MAX-IDE assembler. MAX-IDE is simply not aware that you will be executing the code at a different address from where it is being assembled. For example, suppose that you have a routine called subOne that begins at flash address 0080h and another routine located at 0300h that calls the first routine. This code is shown below.

```
org 0080h

subOne:
   ....perform various calculations...
   ret

...

org 0300h

subTwo:
   call  subOne
   ...and so on...
```

What happens if both these routines are copied into RAM and executed there? Assuming that the routines are copied to the same data memory addresses in RAM as the program memory addresses which they occupy in flash, then subOne will be located at program address A080h and subTwo will be located at A300h.

Because the distance between the line "call subOne" and the destination label subOne is more than the relative jump distance (+127/-128 words), the instruction must be assembled as an absolute LCALL. However, the only address that the assembler has for subOne is 0080h, so the instruction will be assembled as "LCALL 0080h." When subTwo executes, it will not call the copy of subOne located in the RAM, but instead will call the version located in flash.

There are two possible work-arounds to this dilemma. The first and simplest method is to force the assembler to always use relative jumps and calls, and to keep the routines close enough together in RAM that they can call one another in this way. Instead of using the JUMP and CALL opcodes (which allow the assembler to choose short or long jumps), always use SJUMP and SCALL. This will force the use of the relative jump versions of the instructions.

There is, however, a caveat to this approach. If the amount of code that you are running from RAM is longer than 128 words, it is possible that a relative jump will not be long enough for one routine in RAM to call another routine. The solution in this case is to fix addresses for the various routines by using ORG statements, and then define equates that contain their corrected addresses in RAM. These equates can be used in LCALL and LJUMP statements as shown below.

```
subOne  equ  0A080h

org 0080h

; subOne
   ....perform various calculations...
   ret

...

org 0300h

subTwo:
   lcall  #subOne
   ...and so on...
```

This process forces the assembler to use the correct address for the LCALL.

Copying Code to RAM

Before code can be executed from the RAM, it must first be copied into the RAM. The easiest way to copy a large amount of code from the flash to the RAM is to use the Utility ROM copyBuffer function. This function takes two data pointers (DP[0] and BP[Offs]) and a length value (LC[0]) as inputs. It copies the number of specified bytes/words from the source DP[0] to the destination BP[Offs]; it can copy up to 256 bytes/words at a time.

Our demonstration application will copy the first 512 words of itself from flash to RAM, and then jump to the copy in RAM to begin executing code. The source pointer (DP[0]) points to the location of the program flash in the Utility ROM's memory map, which begins at 8000h. Note that to avoid an endless loop we jump to the portion of the copy in RAM that follows the RAM-copying code.

```
org 0020h

copyToRAM:
   move    DPC,    #1Ch        ; Ensure all pointers are operating in word mode.
   move    DP[0], #8000h      ; Start of program flash from UROM's perspective.
   move    BP,     #0          ; Start of data memory.
   move    Offs,  #0
   move    LC[0], #256        ; The Offs register limits us to a 256-word copy.
   lcall   UROM_copyBuffer

   move    DP[0], #8100h      ; Copy second half.
   move    BP,     #0100h
   move    Offs,  #0
   move    LC[0], #256
   lcall   UROM_copyBuffer

   ljump   #0A040h            ; Begin execution of code from RAM.


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;  Executing from RAM
;;

org 0040h

   move    LC[0], #1000
delayLoop:
   move    LC[1], #8000
   sdjnz   LC[1], $
   sdjnz   LC[0], delayLoop

;; Initialize serial port.

   move    SCON.6, #1         ; Set to mode 1 (10-bit asynchronous).
   move    SMD.1,  #1         ; Baud rate = 16 x baud clock
   move    PR, #009D4h        ; P = 2^21 * 9600/8.000MHz
   move    SCON.1, #0         ; Clear transmit character flag.
```

Data Transfer Operations

As mentioned above, two things about the memory map change when executing code from RAM. First, the program flash is now mapped into data memory. This means that we can read data directly from the program flash by using any of the data pointers, as shown below.

```
;; Read the banner string from flash and output it over the serial port.  Since
;; we are running from RAM, we can read from the flash directly without having
;; to use the Utility ROM data transfer functions (moveDP0inc, etc...).

   move    SC.4,  #0
   move    DPC,   #0                ; Set pointers to byte mode.
   move    DP[0], #(stringData * 2)   ; Point to byte address of string data.

stringLoop:
   move    Acc, @DP[0]++
   sjump   Z, stringEnd
   lcall   #TxChar
   sjump   stringLoop
stringEnd:
   move    DPC, #1Ch         ; Set pointers to word mode.

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;  This portion of the code (addresses 200h and higher) will remain in flash.

org 0200h

stringData:
   db      0Dh, 0Ah, "Executing code from RAM....", 00h
```

Note that as shown in Figure 1, the SC.4 (CDA0) bit affects which half of the program flash (upper page or lower page) is mapped into the data memory in byte mode. When using word mode pointers, the entire program flash is mapped into data memory at once.

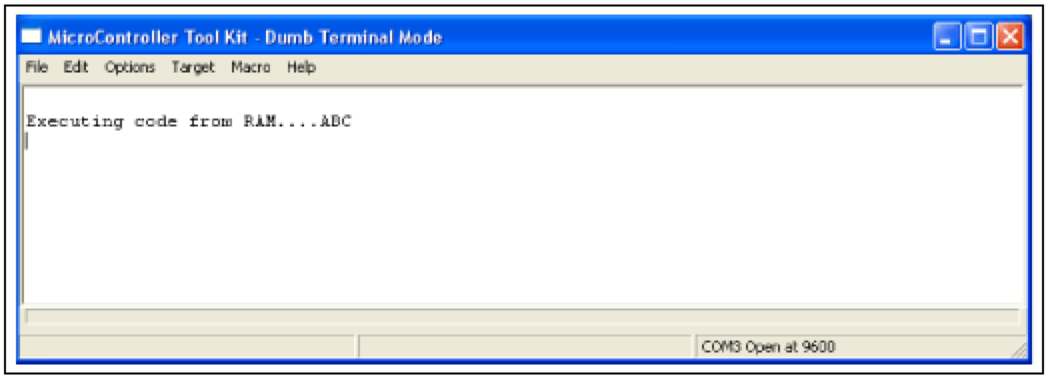
Second, while the flash memory is now accessible in data space, the SRAM is no longer directly accessible. This means that to read from, or to write to, SRAM locations, the application must do so indirectly. Reading from SRAM locations can be performed in the same way that code running in flash reads from flash memory locations—it uses the Utility ROM data transfer functions (moveDP0inc, etc.). Since there are no similar functions in the Utility ROM to perform indirect writes, however, the application must include a small function that remains in flash which can be called by the RAM-resident code to perform a write.

The code below demonstrates both methods used to read and write the RAM variable varA, whose initial contents are copied from flash to RAM along with the rest of the application code located in the address range 0000h-01FFh.

```
   scall   printVar
   scall   incrVar
   scall   printVar
   scall   incrVar
   scall   printVar
   scall   incrVar

   move    Acc, #0Dh
   lcall   #TxChar
   move    Acc, #0Ah
   lcall   #TxChar

   sjump   $


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;  Variables stored in RAM (program) space.  They can be read using the
;;  Utility ROM data transfer functions (such as UROM_moveDP0) and written
;;  using the writeDP0 function which remains in flash.
;;

varA:
   dw 'A'
```

```
;==============================================================================
;=
;=  printVar
;=
;=  Reads the varA RAM variable value and sends it over the serial port.
;=

printVar:
   move    DPC, #1Ch          ; Word mode
   move    DP[0], #varA       ; Variable's location in UROM data space
   lcall   UROM_moveDP0       ; Moves variable value into GR.
   move    Acc, GR
   lcall   #TxChar
   ret


;==============================================================================
;=
;=  incrVar
;=
;=  Reads the varA RAM variable value, adds 1 to it, and stores it back in RAM.
;=

incrVar:
   move    DPC, #1Ch          ; Word mode
   move    DP[0], #varA       ; Variable's location in UROM data space
   lcall   UROM_moveDP0       ; Moves variable value into GR.

   move    Acc, GR
   add     #1
   move    GR, Acc
   lcall   writeDP0

   ret



;==============================================================================
;=
;=  TxChar
;=
;=  Outputs a character to the serial port.
;=
;=  Inputs  : Acc.L - Character to send.
;=

org 01F0h
   move    SBUF, Acc          ; Send character.
TxChar_Loop:
   move    C, SCON.1          ; Check transmit flag.
   sjump   NC, TxChar_Loop    ; Stall until last transmit has completed.
   move    SCON.1, #0         ; Clear the transmit flag.
   ret


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;  This portion of the code (addresses 200h and higher) will remain in flash.

org 0200h

stringData:
   db      0Dh, 0Ah, "Executing code from RAM....", 00h


;==============================================================================
;=
;=  WriteRAM
;=
;=  This is a routine that can be called by code running in the RAM to load
;=  a new value into a byte or word location in the RAM.
;=
;=  Inputs   : DP[0] - Location to write (absolute starting at 0000h) in RAM.
;=             GR    - Value to write to the RAM location.
;=
;=  Notes    : DP[0] must be configured to operate in word or byte mode as
;=             desired before calling this function.  Following a call to this
;=             function, DP[0] must be refreshed before it is used to read data.
;=

writeDP0:
   move    @DP[0], GR
   ret
```

When executed, the demonstration code outputs the following text (**Figure 2**) over the serial port.


MicroController Tool Kit - Dumb Terminal Mode
File Edit Options Target Macro Help
COM3 Open at 9600

```
Executing code from RAM....ADC
```

Figure 2. Text output over serial port by demonstration code.

Conclusion

The Harvard memory map architecture used by the MAXQ8913 and other MAXQ microcontrollers allows you to map different physical memory segments (such as the data SRAM) into either program or data memory space. Executing portions of an application from data SRAM provides the potential for a performance boost and reduced power consumption. The process does require additional application complexity.

MAXQ is a registered trademark of Maxim Integrated Products, Inc.

Related Parts		
MAXQ8913	16-Bit, Mixed-Signal Microcontroller with Op Amps, ADC, and DACs for All-in-One Servo Loop Control	Free Samples

More Information
For Technical Support: http://www.maximintegrated.com/support
For Samples: http://www.maximintegrated.com/samples
Other Questions and Comments: http://www.maximintegrated.com/contact

Application Note 4458: http://www.maximintegrated.com/an4458
APPLICATION NOTE 4458, AN4458, AN 4458, APP4458, Appnote4458, Appnote 4458