Keywords: MAX-IDE,MAXQ

APPLICATION NOTE 4420

Automatically Initializing Data Segment Values in MAX-IDE

Jun 30, 2009

Abstract: This application note discusses the code and data segment facility provided by MAX-IDE for applications programming on MAXQ® microcontrollers. The code and data segment mechanism provides a way to automatically declare variable locations in data memory and to initialize those variables with starting values. Application code can then be used to cache those variable values in flash memory and restore them as needed. This approach allows an assembly-based application to take advantage of the data segment autoloading provided by MAX-IDE, while still operating consistently whether or not the microcontroller is connected to a JTAG debugger. The MAXQ2000 microcontroller EV kit is used to demonstrate the method and code examples are provided in the text.

Overview

Variables in MAXQ assembly applications can be stored either in working registers (such as the accumulators A[0] through A[15]) or in data memory (SRAM). Storing variables in data memory provides a larger working area for the application variables, but does require additional access time.

The MaxQAsm assembler and MAX-IDE environment provide a mechanism to declare separate code and data segments, with a separate hex output file generated for each segment. At run time, MAX-IDE automatically loads the code segment file into program memory (typically flash) and the data segment file into data memory (typically RAM). However, as the data memory is volatile, the data segment contents will not remain intact once the microcontroller is powered off.

This application note uses the MAXQ2000 EV (evaluation) kit to demonstrate first, how to save these preloaded data memory values in flash during the initial application run, and then how to refresh the data segment values from flash when the microcontroller is subsequently powered on again. This two-step process allows the same data segment mechanism to be used to declare and initialize variables, whether the application is in development (connected to a JTAG adapter and MAX-IDE) or running in the field.

Demonstration code for this application note is written for the MAXQ2000 microcontroller and the MAXQ2000 EV kit, but the code and principles shown are applicable to any MAXQ20-based microcontroller with rewriteable program flash memory.

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

- MAX-IDE installation
- MAXQ Core Assembly Guide

Variables and Storage Locations

Embedded applications typically require a certain amount of working space to store state information, configuration settings, intermediate working values, loop counters, and the results of calculations. Values stored in this working space are typically known as variables and share the following characteristics.

1. They are **temporary**. They do not need to be saved if the application is interrupted by a power failure or reset.
2. They are **accessed and updated frequently**. They must be stored in locations that can be read from or written to quickly; there must be no limit on the number of times to which the locations can be written.
3. They often have **defined initial values**. The user's code must set them to a particular value at the beginning of the application.

In applications written in C or another high-level language and compiled into assembly code, the compiler typically handles the allocation of space for variables (as well as the process of initializing variables to predefined starting values) automatically. In this case, the user only needs to declare the variable, its type, and (optionally) its initial value. The compiler handles the rest.

```
unsigned int c = 0x1234;
```

However, when writing applications directly in MAXQ assembly language, the allocation of space for variables and setting the variables to initial values must be performed explicitly. This detailed action allows tighter control of the resources available on the MAXQ microcontroller, but adds some complexity to the system.

For small assembly-based applications or those which do not require a large amount of working space, internal registers can be used to store all application variables. This approach provides two important advantages:

1. Compact, fast code. Register variables can be read from, written to, or copied to another register variable in as little as one instruction cycle, depending on the location of the register. On MAXQ20-based microcontrollers, no more than two instruction cycles will generally be required in the worst case.
2. Direct operations on variables. Some internal register locations can be operated on directly. For example, any of the 16 working accumulators, A[0] through A[15], can be selected (using the AP register) as the active accumulator, Acc. This means that if an operation needs to be performed on a variable stored in one of these registers, it can be performed directly on that register without having to copy the value out, perform the operation, and copy the value back in. Similarly, variables stored in the LC[0] and LC[1] registers can be used directly as loop counters by executing the `djnz` instruction.

A larger application, or an application requiring a larger number of working variables, can benefit from storing some or all of its variables in SRAM-based data memory. This method allows a much larger number of variables to be created, up to the limits imposed by the size of the data memory. Variables stored in this manner are accessed using one of the MAXQ20 core's standard data pointers, which can be used to read and write byte-sized or word-sized variables. (Note: all code examples in this application note assume that DP[0] is configured to operate in word mode.)

```
move   DP[0], #0010h      ; Location of variable in data memory
move   Acc, @DP[0]        ; Read variable
add    #1                 ; Increment variable value by 1
move   @DP[0], Acc        ; Store variable back in data memory
```

If a long series of calculations must be performed on a variable, the value of that variable can be copied into a working register, as shown in the example code above. All intermediate operations can be performed using that working register, and the value can be copied back out to the variable once calculations are complete.

Segment Declarations in MAX-IDE

Once the decision is made to store application variables in SRAM-based data memory, how do you determine where to store the variables?

Typically, all of the data memory is available for application use, except for the highest 32 bytes in memory which are used by the debugger. This means that declaring a variable is simply a matter of defining a location for it in data memory. This location is then used by code whenever the variable is read or written. The `#define` macro can be used to associate a variable location with a symbolic name.

```
#define VarA  #0020h
#define VarB  #0021h
#define VarC  #0022h

   move   DP[0], VarA        ; Point to VarA variable
   move   Acc, @DP[0]        ; Read value of variable
   move   DP[0], VarB        ; Point to VarB variable
   move   @DP[0], Acc        ; Copy VarA to VarB
   move   DP[0], VarC        ; Point to VarC variable
   move   @DP[0], #1234h     ; Set VarC = 1234h
```

This approach works well enough, but there are several problems with it.

- The location of each variable must be determined in advance. This task can be time-consuming, especially if it is decided later to move all variables to a different area of data memory.
- Care must be taken not to accidentally use the same location for more than one variable. If this mistake is made, it can be difficult to track bugs.
- Any initial (starting) values for variables must be explicitly loaded by application code, as shown in the last line above. This action can consume a large amount of code space if there are many variables to initialize in this manner.

A more efficient approach takes advantage of MAX-IDE's mechanism to declare separate code and data segments. This method allows the application author to specify which portions of the assembly code file are destined for code space and which portions are destined for data space.

```
segment code

   move  DP[0], #VarA       ; Point to VarA
   move  Acc, @DP[0]        ; Get current value of VarA
   add   #1                 ; Increment it
   move  @DP[0], Acc        ; Store value back in VarA

segment data

VarA:
   dw    0394h              ; Initial value for VarA
```

In the above approach, addresses for variables declared in the data segment are determined automatically by the assembler as it parses the file with the same method used to assign addresses to labels in code space. Labels are used to assign symbolic names to these variable addresses, and the `dw` and `db` statements can be used to initialize word-sized and byte-sized variables with starting values. In

this case, assuming that no previous `segment data` directive was found in the assembly file, the assembler will begin the data segment at address 0000h. This means that `VarA` will be stored at word address 0000h. As in code space, the `org` statement can force variables to be located at the beginning of a specified address.

Initializing the Data Segment

In the previous code listing, variable `VarA` is defined (using the `dw` statement) to have an initial value of 0394h. But this value is never loaded into `VarA` in the code. How, then, is this value initialized? The answer is that the initialization of the data segment is performed automatically by MAX-IDE when the project is compiled and executed.

The MaxQAsm assembler responds to the `segment data` directive by generating a secondary hex output file. Normally, a hex file is generated for a project containing code data. For example, if project "example.prj" is compiled, a hex file will be created named "example.hex" containing the code data generated by assembling the project files. If a data segment is defined, an additional hex file will be created named "example_d.hex", which contains the data assembled in this segment.

When the project is executed, MAX-IDE checks to see if a data segment file (ending in _d.hex) was generated during project compilation. If a data segment file exists, MAX-IDE uses the standard JTAG loader to load the data from this segment into the data SRAM of the device. This is done after the standard hex file has been loaded into program memory.

This method works well during the development cycle, when the device is connected to a JTAG adapter and MAX-IDE reloads the code and segment data before each application run. However, once the device is powered off and on and allowed to run independently (with no debugger connected), MAX-IDE no longer has the ability to load the data segment with the proper values before each run. The variables will no longer be set to their expected values, and the application may execute incorrectly as a result. This type of failure can be difficult to analyze, because as soon as the device is hooked back up to the debugger, MAX-IDE will begin loading the data segment again before each run and the problem will instantly vanish.

Saving and Restoring the Data Segment

A question remains: how can you make the application operate consistently, whether it is connected to a debugger (with MAX-IDE reloading code and data before each run) or free-running (with no particular contents in RAM guaranteed following power-up). The obvious solution is a two-step process: have the application save the variable values (once they have been initialized) in flash memory, and restore the values following each reset or power-up.

As a first step, the application must save values to flash memory. This action occurs the first time that the application is executed following each master erase and code load cycle.

1. The application checks a "flag" location to verify that the variables have not been previously copied to flash. This flag can be a special-purpose, nonvariable location, or it can be shared with a variable as long as that variable has a nonzero initial value (to distinguish it from a blank RAM location).
2. The application copies each variable value from data RAM to flash memory. On most MAXQ microcontrollers with rewriteable flash (such as the MAXQ2000) this is done using the UROM_flashWrite function.
3. The application writes a flag in flash memory to indicate that the variables have been stored.

As the secondly step, on subsequent runs the application must restore the variable values from flash memory to their expected locations in data RAM.

1. The application checks the flag location in flash to verify that the variable values have been stored.
2. The application uses the UROM_copyBuffer routine to copy the variable values from flash memory to their proper locations in data RAM.

The code listing below demonstrates this saving-restoring method with the MAXQ2000 EV kit. In this code, variable values are stored in flash memory at addresses 7000h–71FFh.

```
$include(maxQ2000.inc)

;; Code memory (flash) : 0000h-7FFFh  (word addr)
;; Data memory (RAM)   : 0000h-03FFh  (word addr)

org 0000h

   ljump   start             ; Skip over password area

org 0020h

start:
   move    DPC, #1Ch         ; Set all pointers to word mode
   move    DP[0], #0F000h    ; Check first variable value (flag)
   lcall   UROM_moveDP0      ; 'move GR, @DP[0]' executed by Utility ROM
   move    Acc, GR
   cmp     #1234h
   jump    NE, copyToFlash

;; This is the "free-running" code, executed on subsequent power-ups, that
copies
;; values from the flash back into their proper data segment locations.

   move    DP[0], #0F000h    ; Source: Flash location 7000h
   move    BP,    #0         ; Dest:   Start of RAM
   move    Offs,  #0
   move    LC[0], #100h      ; Copy 256 words
   lcall   UROM_copyBuffer

   jump    main

;; This is the first-pass code. A bit of a trick here; because MAX-IDE enters
;; and exits the loader separately when loading the code and data segment
files,
;; the application is allowed to execute briefly before the data segment file
;; has been loaded. The first four lines under copyFlash ensure that the
;; application waits for MAX-IDE to load the data segment file before
continuing.

copyToFlash:
   move    DP[0], #0h        ; Wait for flag variable to be loaded by MAX-
IDE.
   move    Acc, @DP[0]       ;    Note that this will reset the application;
the
   cmp     #1234h            ;    data segment is not loaded while the
application
   jump    NE, copyToFlash   ;    is still running.

   move    DP[0], #0         ; Start of RAM variable area
   move    A[4],  #7000h     ; Location in flash to write to
   move    LC[0], #100h      ; Store 256 words in flash 7000h-70FFh

copyToFlash_loop:
   move    DP[0], DP[0]      ; Refresh the data pointer to read values
correctly,
                             ;    because calling UROM_flashWrite changes
memory
                             ;    contexts and affects the cached @DP[0]
value
   move    A[0], A[4]        ; Location to write
   move    A[1], @DP[0]++    ; Value to write (taken from RAM)
```

```
   lcall   UROM_flashWrite
   move    Acc, A[4]
   add     #1
   move    A[4], Acc
   djnz    LC[0], copyToFlash_loop

main:
   move    PD0,   #0FFh      ; Set all port 0 pins to output
   move    PO0,   #000h      ; Drive all port 0 pins low (LEDs off)

   move    DPC,   #1Ch       ; Set pointers to word mode
   move    DP[0], #varA
   move    Acc,   @DP[0]
   cmp     #1234h            ; Verify that the variable is set correctly
   jump    NE, fail

pass:
   move    PO0, #55h
   sjump   $

fail:
   sjump   $

segment data

org 0000h

varA:
   dw  1234h

org 00FFh

varB:
   dw  5678h

end
```

Conclusion

The code and data segment facility provided by MAX-IDE provides a way to automatically declare variable locations in data memory and initialize those variables with starting values. Application code can then be used to cache those variable values in flash memory and restore them as needed. This approach allows an assembly-based application to take advantage of the data segment autoloading provided by MAX-IDE, while still operating consistently whether or not the microcontroller is connected to a JTAG debugger.

Related Parts		
MAXQ2000	Low-Power LCD Microcontroller	Free Samples
MAXQ2010	16-Bit Mixed-Signal Microcontroller with LCD Interface	Free Samples
MAXQ8913	16-Bit, Mixed-Signal Microcontroller with Op Amps, ADC, and DACs for All-in-One Servo Loop Control	Free Samples

More Information
