Keywords: stack,assembly,MAX-IDE

APPLICATION NOTE 4238

Implementing a Soft Stack in Data Memory on the MAXQ2000

Jun 02, 2008

Abstract: This application note demonstrates a simple method for implementing a soft stack in data memory for assembly-based applications. This method uses the MAXQ2000 and other MAXQ20-based microcontrollers. The example code is written using the macro preprocessing features of MAX-IDE, Maxim's project-based application development and debugging environment for the MAXQ® family.

Overview

The MAXQ2000 microcontroller, like other MAXQ devices in Maxim's RISC microcontroller family, is based on the MAXQ20 core. MAXQ20-based microcontrollers typically implement a 16-bit-wide hardware stack with a fixed number of levels (16 on the MAXQ2000) stored in a dedicated internal memory separate from data and code space. This hardware stack is used to save and restore the microcontroller's operating state across subroutine calls and interrupt operations.

While perfectly adequate for small, tightly-focused applications, the hardware stack quickly runs out of space when deeply nested subroutines (or subroutines that save and restore more than a few working registers on the stack) are used in larger assembly applications. Applications written in the C programming language (using compilers such as IAR's Embedded Workbench®) or Rowley Associates' Crossworks for MAXQ avoid this problem by utilizing a "soft stack" contained in data memory. (For more information on these development tools, go to: Development Tools.) This soft stack stores call/return addresses and local working variables for subroutines. However, there is no built-in mechanism on the MAXQ20 core to locate the stack in data memory which can be used in assembly-only applications.

This application note demonstrates a simple method to implement a soft stack in data memory for assembly-based applications. The code presented in this application note can be used on the MAXQ2000 and other MAXQ20-based microcontrollers. The example code is written using the macro preprocessing features of MAX-IDE, Maxim's project-based application development and debugging environment for the MAXQ family.

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

- MAX-IDE Installation (ZIP)
- MAXQ Core Assembly Guide (PDF)
- Development Tools Guide (PDF)

Hardware Stack Operations in the MAXQ20 Core

There are two different types of stack operations used by the MAXQ20 core:

- PUSH operations (which include the opcodes PUSH, LCALL, and SCALL) are used to store data on the stack. These operations preincrement the stack pointer SP by 1, and then store data at the stack location pointed to by the SP pointer (@SP).
- POP operations (which include the opcodes POP, POPI, RET, and RETI) are used to retrieve data from the stack. These operations retrieve data from the stack location pointed to by SP, and then postdecrement the stack pointer by 1.

Since one of the main functions of the hardware stack is to save and restore addresses when calling subroutines, the stack consists of 16-bit (word) locations. This width allows the 16-bit Instruction Pointer (IP) register to be saved or restored in a single push or pop operation. Even if a PUSH or POP is used to save or restore an 8-bit register (such as AP) to or from the stack, an entire 16-bit word is always used by each stack operation.

Besides the various opcodes which utilize the stack, there are two additional circumstances under which the microcontroller automatically uses the hardware stack:

- When an interrupt is serviced, the current program execution point is pushed onto the stack before execution of the interrupt service routine (which is pointed to by the interrupt vector register IV) begins.
- When certain debugging commands are invoked (such as Read Register and Write Data Memory) which require execution of code in the Utility ROM to complete, the current execution point is pushed onto the stack before control is transferred to the Utility ROM. Once the debug routine in the Utility ROM has finished, the execution point is popped back off the stack and the previous state of the processor is restored.

Vectoring to an interrupt service routine or executing a debugger command will always require use of the hardware stack. Since this behavior is embedded in the hardware, there is no way to work around it. However, for the more common PUSH/POP and CALL/RET instruction pairs, a soft stack can be implemented instead.

Note that the MAXQ2000 (as with other devices based on the MAXQ20 core) does not provide any error detection or warning when either of the following stack error conditions occur:

- A **stack overflow**: occurs when a value is pushed on the stack which is already full. This error causes the oldest value in the stack to be overwritten.
- A **stack underflow**: occurs when a value is popped from the stack when the stack is empty. This error causes an invalid data value to be returned. For example, if a RET is executed when the stack is empty, execution will be transferred to an incorrect (potentially random) address.

Creating a Soft Stack in Data Memory

The first step to creating a soft stack in data memory is to define what portion of the data memory will be used. Then the data memory pointer (DP[0], DP[1], or BP[Offs]) to track the current location of the top of the stack must be defined. **Note**: Take care that the application software does not use the data memory dedicated for the stack for other purposes (e.g., variables or buffers).

One simple way to define and initialize such a soft stack involves the BP[Offs] register pair and one equate.

```
SS_BASE  equ  0100h

ss_init:
```

```
   move    DPC,  #1Ch         ; Set all pointers to word mode
   move    BP,   #SS_BASE     ; Set base pointer to stack base location
   move    Offs, #0           ; Set stack to start
   ret
```

If the base pointer (BP) register is set to the SS_BASE location, then the Offs register can be used to point to the current top of the stack. Since the Offs register is only 8 bits wide, hardware automatically limits the stack to the range (BP .. (BP+255)) in data memory. If a push occurs when Offs is equal to 255 (overflow), or if a pop occurs when Offs is equal to 0 (underflow), then the Offs register will simply wrap around to the other end of the stack. This action simulates the way that the hardware stack operates and allows a simple soft stack implementation; this does not detect underflow and overflow conditions.

The `ss_init` routine must be called by the main application before the soft stack is used. It sets the BP[Offs] pointer to word mode, which must be done because the soft stack is implemented as a 16-bit wide stack. It also points the BP[Offs] register pair to the beginning of the stack.

Soft Stack Operations

The built-in op codes which use the stack (PUSH, POP, CALL, RET, etc.) cannot be redefined to use the soft stack, since their meanings are hardwired into the MAXQ assembler. Moreover, it might still be necessary to use the standard hardware stack in certain parts of the application, such as interrupt service routines. For these reasons, the soft stack will be accessed by macros which replicate the standard op codes.

```
mpush  MACRO  Reg
   move    @BP[++Offs], Reg  ; Push value to soft stack
endm

mpop   MACRO  Reg
   move    Reg, @BP[Offs--]  ; Pop value from soft stack
endm

mcall  MACRO  Addr
LOCAL  return
   move    @BP[++Offs], #return   ; Push return destination to soft stack
   jump    Addr
return:
endm

mret   MACRO
   jump    @BP[Offs--]       ; Jump to popped destination from soft stack
endm
```

We will now discuss how these macros operate.

`mpush` <reg>

This macro is used in the same manner as the PUSH op code. It allows an 8-bit or 16-bit register, or an immediate value to be pushed to the stack.

```
   mpush   A[0]              ; Save the value of the A[0] register
   mpush   A[1]              ; Save A[1]
   mpush   A[2]              ; Save A[2]

   ...                       ; code which destroys A[0]-A[2]

   mpop    A[2]              ; Restore the value of A[2] (pop in reverse
order)
   mpop    A[1]              ; Restore A[1]
   mpop    A[0]              ; Restore A[0]
```

`mpop` <reg>

This macro is used in the same manner as the POP op code. It allows an 8-bit or 16-bit register to be loaded from the stack, as shown above. Note that if a 16-bit value is pushed and this value is popped into an 8-bit register, only the low byte will be stored in the register. The high byte will be lost. This is identical to the behavior of the built-in hardware stack.

```
subroutine:
   mpush   A[0]              ; Save the current value of A[0]

   ...                       ; Code which destroys A[0]

   mpop    A[1]              ; Restore A[0]
   mret
```

`mcall` <address>
`mret`

The `mcall` macro is used in the same manner as the CALL op code to execute a subroutine. This subroutine must use the `mret` macro (and not the standard RET op code) to return once it has completed execution.

```
   mcall   mySub

   ...

mySub:
   mpush   A[0]              ; Save A[0]
   mpush   A[1]              ; Save A[1]
   ...                       ; Perform calculations, etc.
   mpop    A[1]
   mpop    A[0]
   mret
```

Extending the Size of the Soft Stack

Some applications require a soft stack larger than 256 levels. It is possible to implement a stack of any size (up to the limits of available data memory) by using one of the other data pointers (DP[0] or DP[1]).

```
SS_BASE  equ  0000h
SS_TOP   equ  01FFh

ss_init:
   move    DPC,    #1Ch       ; Set all data pointers to word mode
   move    DP[0], #SS_BASE    ; Set pointer to stack base location
   ret
```

The code shown above reserves the locations 0000h through 01FFh in data memory, thus creating a stack that can hold up to 511 levels. (One location in the stack memory space is left unused; this allows a shorter, more efficient implementation of the `mpush/mpop/mcall/mret` macros).

Simply changing the data pointer and leaving everything else in the code the same will extend the stack. Nonetheless, since DP[0] is not restricted to a certain range in data memory, there is nothing to prevent a push or pop operation from incrementing or decrementing DP[0] beyond the assigned boundaries of the soft stack. To avoid this, some simple underflow/overflow checking can be added.

Adding Underflow and Overflow Checking

To keep the macros (which are expanded into code each time that they are used) as short as possible, the underflow and overflow checking is performed in subroutines which are called by the macros using

the hardware stack.

```
mpush  MACRO  Reg
   call    ss_check_over     ; Check for possible overflow
   move    @++DP[0], Reg     ; Push value to soft stack
endm

mpop   MACRO  Reg
   call    ss_check_under    ; Check for possible underflow
   move    Reg, @DP[0]--     ; Pop value from soft stack
endm

mcall  MACRO  Addr
LOCAL  return
   call    ss_check_over     ; Check for possible overflow
   move    @++DP[0], #return ; Push return destination to soft stack
   jump    Addr
return:
endm

mret   MACRO
   call    ss_check_under    ; Check for possible underflow
   jump    @DP[0]--          ; Jump to popped destination from soft stack
endm

ss_check_under:
   push    A[0]
   push    AP
   push    APC
   push    PSF

   move    APC, #80h         ; Set Acc to A[0], standard mode, no auto
inc/dec
   move    Acc, DP[0]        ; Get current value of stack pointer
   cmp     #SS_BASE
   jump    NE, ss_check_under_ok
   nop                       ; < Error handler should be implemented here >
ss_check_under_ok:
   pop     PSF
   pop     APC
   pop     AP
   pop     A[0]
   ret

ss_check_over:
   push    A[0]
   push    AP
   push    APC
   push    PSF

   move    APC, #80h         ; Set Acc to A[0], standard mode, no auto
inc/dec
   move    Acc, DP[0]        ; Get current value of stack pointer
   cmp     #SS_TOP
   jump    NE, ss_check_over_ok
   nop                       ; < Error handler should be implemented here >
ss_check_over_ok:
   pop     PSF
   pop     APC
   pop     AP
   pop     A[0]
   ret
```

The above code causes the current stack position to be checked before a push or pop (or call or ret) operation occurs. If an overflow or underflow error is detected, the response will vary from application to application. Normally, this sort of error is something that should only occur during application development; it should not occur if the code is written correctly. If the error does occur, it should

generally be considered a fatal error, as it would be if an underflow/overflow occurred while using the hardware stack. Possible responses to this error include halting and transmitting an error message or blinking an LED. A useful trick during development is to set a breakpoint in MAX-IDE inside each of these two routines (i.e., at the "Error handler should be implemented here" line) to allow immediate feedback if an underflow or overflow occurs.

However, sometimes the application can recover from a stack underflow or overflow (for example, by reloading and restarting application subtasks from the beginning). In that situation it might be desirable to simply set a flag which indicates that a stack error has occurred. Possible candidates for such flags include the two general-purpose flags (GPF0 and GPF1) located in the PSF register. Since there are two bit flags available, one of them could be used to indicate an overflow and the other to indicate an underflow.

Conclusion

The powerful macro preprocessing capabilities provided by MAX-IDE allow straightforward implementation of a replacement soft stack in data memory on the MAXQ2000 and other MAXQ20-based microcontrollers. This soft stack assists development of larger assembly-based applications by allowing subroutines to be more modular and reusable. The stack also allows detection of stack-based errors.

Related Parts		
MAXQ2000	Low-Power LCD Microcontroller	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 4238: http://www.maximintegrated.com/an4238
APPLICATION NOTE 4238, AN4238, AN 4238, APP4238, Appnote4238, Appnote 4238