TWO STACKS IN C166

The basic design of the 80C166 was deliberately biased towards allowing structured languages like C to run more efficiently than on older CPUs. One of the most useful instruction set features is the provision of 16 additional stack pointers. These are the result of the `MOV [Ri+], mem` instructions which are ideal for creating local stacks.

Under normal circumstances, C166 uses just one of the 16 potential stack pointers (i.e. general purpose registers), namely R0. The stack created by R0 is placed in a special section in NDATA called ?C_USERSTACK. The "user stack" with R0 as its user stack pointer is used by C166 for parameter passing and local automatic variables. When a function is called, any variables or other data that cannot be fitted into registers are "pushed" on to the user stack by the `MOV [R0-], parameter` instruction. The "R0-" causes R0 to point at the next free location on the user stack. Once in the called function, the parameter is moved off the user stack by the inverse instruction `MOV reg, [R0+]`. Note that the R0+ moves the user stack point. As with the true system stack pointer, SP, every `MOV [R0-],xxx` is matched with a `MOV xxx, [R0+]` so that the user stack pointer is always restored to its original value after a function call.

Setting the Size of the User Stack

Due to C166 placing up to 8 parameters and 15 locals (automatics) in registers, it is fairly rare for the user stack to be used at all. If the optimizer is disabled, you will instantly see a large number of `MOV [-R0], R11` type instructions as C166 starts to move things onto the user stack.

Note: If optimization is disabled, the user stack size is increased greatly!

```
; FUNCTION interp_sub (BEGIN RMASK=@0x2030)
;   unsigned char interp_sub (unsigned char x, unsigned char y,
                                                ; SOURCE LINE # 11
      MOV   [-R0],R11
      MOV   [-R0],R10
      MOV   [-R0],R9
      MOV   [-R0],R8
      SUB   R0,#2
;                               unsigned int n, unsigned int d) {
;     unsigned char t;
;     if (y>x)
                                                ; SOURCE LINE # 15
      MOVB RL5, [R0+#2]
      MOVB RL4, [R0+#4]
      CMPB  RL4, RL5
      JMP   cc_ULE, ?C0001
;       {
                                                ; SOURCE LINE # 16
;       t = y-x;
                                                ; SOURCE LINE # 17
      MOVB  RL5, [R0+#2]
```

Note: This habit of pushing everything on the user stack is why conventional processors like the 68000 and the 8086 have relatively poor performance in C.
C166 register variable handling seriously reduces the load on the user stack and often does not use it at all.

```
; FUNCTION interp_sub (BEGIN RMASK=@0x2030)
;   unsigned char interp_sub (unsigned char x, unsigned char y,
                                                ; SOURCE LINE # 11
;-- Variable 'd?00' assigned to Register 'R11" –
;-- Variable 'n?00' assigned to Register 'R10" --
;-- Variable 'y?00' assigned to Register 'R9" --
;-- Variable 'x?00' assigned to Register 'R8" --
;-- Variable 't?00' assigned to Register 'RL6" --
;                               unsigned int n, unsigned int d) {
;     unsigned char t;
;     if (y>x)
                                                ; SOURCE LINE # 15
      MOV   R5, R8      <- Very fast, 100ns
      MOV   R4, R9
      CMPB  RL4, RL5
      JMP   cc_ULE, ?C0001
```

This can make the maximum size of the user stack a great deal less than might be expected. However, C166 will only use registers if optimization is set to maximum. As shown above, if the optimization is disabled, the user stack will suddenly grow and may well exceed the allocated space, resulting in a program crash.

The default user stack size is 1000H bytes and this is adequate for very large programs - the size is set in the STARTUP.A66 file. It is a good idea to leave the stack at this size until the bulk of your program has been written, and then examine the actual worst-case stack used. If you are using the registerbanks and register mask properly, you should be able to reduce this to 100H or less. This can be estimated by working out the maximum function/interrupt nesting and adding up the total number of `MOV [R0-]` statements possible. Alternatively, an in-circuit emulator can be used to monitor activity in the designated ?C_USERSTACK area.

Variables that end up on the user stack are considerably slower to access as there are no ADD, SUB, or CMP instructions which can use the `Rw, [R0+#offset16]` addressing mode. In other words, variables on the user stack must be moved off the stack into a register, operated on, and then moved back onto the stack.

A significant performance advantage for interrupt functions or those with a large number of local variables can be had by forcing the compiler to put locals that cannot fit into registers (R1-R15) into static (near) RAM segments to create a "compiled" stack. This is similar to the C51 compiler. The common ADD, SUB, and CMP instructions all can operate directly on RAM so that there is little performance loss when compared to register variables. This is accomplished with the **Use static memory for non-register automatics** in the C166 Options menu.

Please note that any functions within modules compiled with this control will no longer be reentrant, i.e. the same function cannot be called both from an interrupt and a background function. In this case, the values acquired by "static" variables in the background call would be destroyed by those originating from the interrupt function. This control is best used as a `#pragma STATIC` with only those modules which contain functions which can be used in a non-reentrant form such as interrupt routines.

Placing the User Stack in On-Chip RAM

As previously stated, the user stack is fixed in a section named ?C_USERSTACK, part of the NDATA class. However, it is quite simple to move it to other memory spaces. The most common action is to place it in the IDATA class so that it can be on-chip. As the user stack is rarely vary large, IDATA should be able to contain it easily.

A small modification is required to START167.A66 to achieve this:

Upper part of START167.A66:

```
DUMMY SECTION DATA PUBLIC 'NDATA'     ; dummy section to establish NDATA class
DUM   DS   1              ; define a single byte to get rid of empty sections linker warning
DUMMY ENDS

?C_USERSTACK    SECTION    DATA PUBLIC 'IDATA'

$IF NOT TINY
NDATA      DGROUP     DUMMY
$ENDIF
?C_USRSTKBOT:
           DS    40H
?C_USERSTKTOP:
?C_USERSTACK    ENDS
…….
```

Further down, after the EINIT instruction:

```
;
$IF NOT TINY
     MOV   R0, #DPP3:?C_USERSTKTOP
$ENDIF
$IF TINY
     MOV   R0, #?C_USERSTKTOP
$ENDIF
```

This makes R0 load with DPP3 for an on-chip user stack, rather than the default DPP2. Remember to add the copy of START167.A66 that you modify to your project.

The System Stack

With the user stack taking care of function parameters and local variables, the system stack is used for storing return addresses, the current PSW and CP plus any general purpose registers in the current register bank used for local register variables. This stack is always located on-chip and defaults to 256 words in length (80C166) at 0xFBFF down to 0xFA00. The required stack size is set in the START167.A66 file. Values of 32, 64, 128, and 256 words can be selected vian SYSCON. The CPU register "SP" has its top 5 bits hardwired to '1,' the stack is always in the range 0xF800 to 0xFFFE, i.e. on-chip.

Setting the System Stack Size

```
; STKSZ: Maximum System Stack Size selection(SYSCON.13 ..
; SYSCON.14)
_STKSZ     EQU   0
;                0 = 256 words (Reset Value)
;                1 = 128 words
;                2 =  64 words
;                3 =  32 words
```

The 80C166 has two special registers, STKOV and STKUN, which set the top and bottom limits of the stack. The default value of STKOV (stack overflow) is 0xFA00; STKUN defaults to 0xF000, in-line with the default 256 words.

The address of the stack is defined by loading the STKOV register in STARTUP.A66 as such:

```
; Setup Stack Overflow
_TOS  EQU   0FC00H                      ; top of system stack
_BOS  EQU   _TOS - (512 >> _STKSZ)      ; bottom of system stack

PUBLIC           ?C_SYSSTKBOT
?C_SYSSTKBOT     EQU   _BOS

      MOV   STKOV, #(_BOS+6*2)          ; init stack ovfl register
```


In the USA:
Keil Software, Inc.
16990 Dallas Parkway, Suite 120
Dallas, TX 75248-1903
USA

Sales: 800-348-8051
Phone: 972-735-8052
FAX: 972-735-8055

E-mail: sales.us@keil.com
support.us@keil.com

Internet: http://www.keil.com/

In Europe:
Keil Elektronik GmbH
Bretonischer Ring 15
D-85630 Grasbrunn b. Munchen
Germany

Phone: (49) (089) 45 60 40 - 0
FAX: (49) (089) 46 81 62

E-mail: sales.intl@keil.com
support.intl@keil.com