

μC/OS-II and Borland's C/C++ Floating-Point Emulation

Application Note

AN-1001

**Jean J. Labrosse
and
Bogdan Kowalczyk (PetroVend)**
Jean.Labrosse@uCOS-II.com
www.uCOS-II.com

Acknowledgements

I would like to thank Mr. Bogdan Kowalczyk from PetroVend in Poland (bkowalczyk@petrovend.com) for contributing a major portion of this application note.

Summary

The Borland C/C++ Floating-Point Emulation (FPE) assumes that about 300 bytes starting at `SS:0x0000` are reserved to hold floating-point emulation variables. This applies to the 'large memory model' only. To accommodate this, a special function (`OSTaskStkInit_FPE_x86()`) must be called prior to calling either `OSTaskCreate()` or `OSTaskCreateExt()` to properly initialize the stack frame of each task needing to perform floating-point operations. This function is described in this application note and applies to Borland C/C++ V3.1 and V4.5.

Introduction

When floating point emulation is turn on the stack of the Borland C++ compiled program is organized as shown in figure 1. The compiler assumes that the application runs in a single threaded environment.

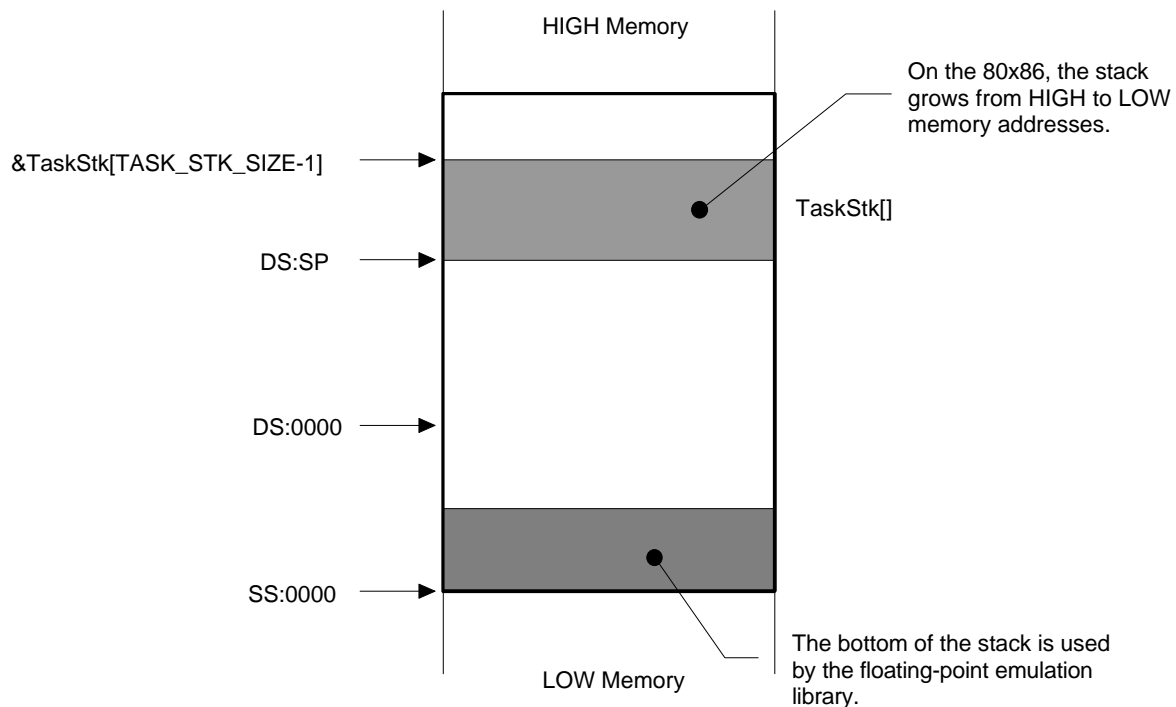


Figure 1. Stack organization with Borland C/C++.

The floating-point emulation library stores its data within the reserved space in relation to the current `SS` register value, assuming that some space starting from `SS` up (from `SS:0x0000` up) is reserved for floating point operations.

µC/OS-II's task stacks are generally allocated statically as shown below.

```
OS_STK Task1Stk[TASK_STK_SIZE]; /* stack table for task 1 */
OS_TSK Task2Stk[TASK_STK_SIZE]; /* stack table for task 2 */
```

When a task is created by µC/OS-II the highest table address of the stack is pass to OSTaskCreate() (or OSTaskCreateExt()) as shown below:

```
OSTaskCreate(Task1, (void*)0, &Task1Stk[TASK_STK_SIZE-1], prio1);
OSTaskCreate(Task2, (void*)0, &Task2Stk[TASK_STK_SIZE-1], prio2);
```

The stack of Task1() starts at DS:&Task1Stk[TASK_STK_SIZE-1] while the stack of Task2() starts at DS:&Task2Stk[TASK_STK_SIZE-1]. Once initialized by µC/OS-II, the tasks top-of-stack (TOS) is saved in the task's OS_TCB (Task Control Block).

The stack of the two tasks created from the previous code is shown in figure 2. As can be seen, both tasks are part of the same segment and, more importantly, they share the same segment base since both stacks are allocated from the same data segment. When µC/OS-II loads a task during a context switch, it sets the SS register to the value of the DS register of the stack. This causes a problem since both tasks would have to share the same floating-point emulation variables!

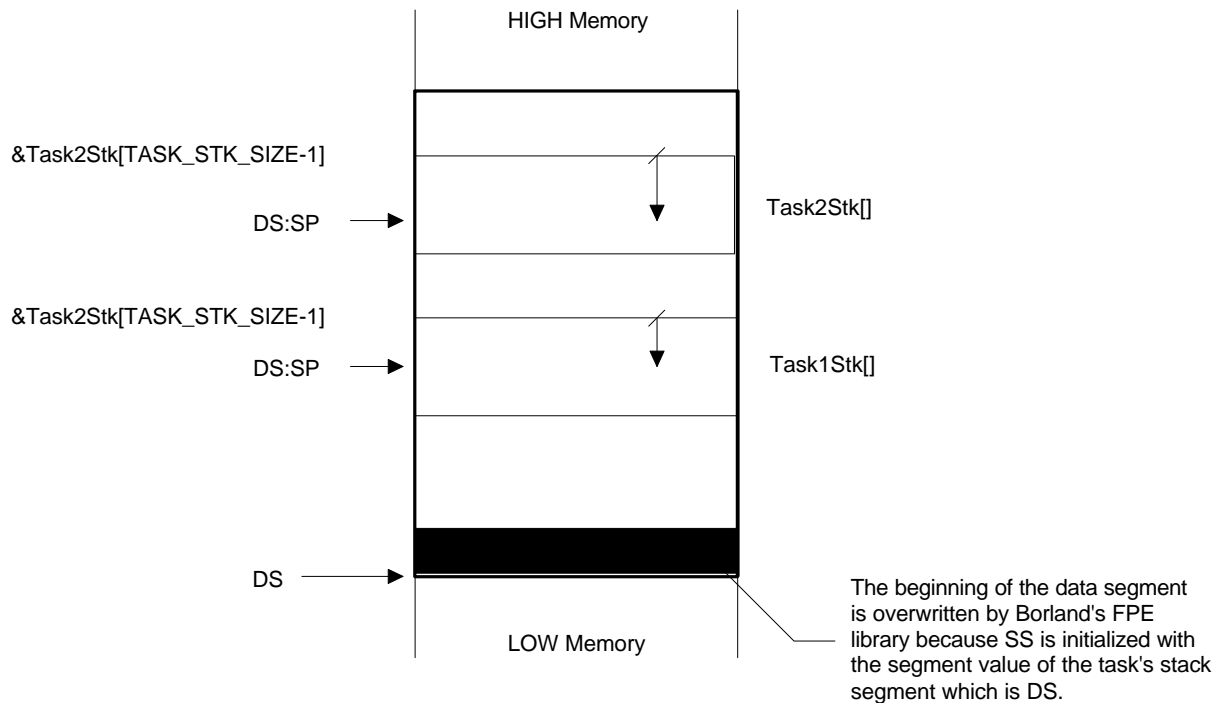


Figure 2. Stacks under µC/OS-II and the Borland FPE library.

The beginning of the data segment is overwritten with the floating-point emulation library even when we use a semaphore. Protecting this resource with a semaphore would allow exclusive access to the

floating-point variables but it does not protect the data segment from being overwritten. Even a single µC/OS-II task using floating point overwrites the data segment! Further system behavior depends on what data are overwritten and typically data segment overwriting crashes the system.

A similar situation occurs when the stacks are allocated from the heap since we don't know what part of memory is being overwritten. Typically, the heap is corrupted because the floating-point emulation library overwrites the header of the heap allocated block.

OSTaskStkInit_FPE_x86() fixes the problem

To fix this problem, the function `OSTaskStkInit_FPE_x86()` (see listing 1) needs to be called prior to creating a task. This function basically 'normalizes' the stack so that every stack starts at `SS:0x0000` and, the function reserves and properly initializes the floating-point emulation variables for the task being created.

As can be seen from the code, you need to pass three arguments to `OSTaskStkInit_FPE_x86()`:

`pptos` is a pointer to the task's top-of-stack (TOS) pointer (a pointer to a pointer). The task's TOS is passed to `OSTaskCreate()` or `OSTaskCreateExt()` when you create a task. The stack is allocated from the data space and consist of a value for the DS register and an offset from this segment register. Because `OSTaskStkInit_FPE_x86()` normalizes the TOS, a pointer to the initial TOS is passed to this function so that it can be altered.

`ppbos` is a pointer to the task's bottom-of-stack (BOS) pointer (a pointer to a pointer). The task's BOS is not passed to `OSTaskCreate()` however, it is passed to `OSTaskCreateExt()`. In other words, `ppbos` is necessary for `OSTaskCreateExt()`. The bottom of this stack is generally not located at `DS:0000` but instead, at some offset from the DS register. Because `OSTaskStkInit_FPE_x86()` normalizes the BOS, a pointer to the initial BOS is passed to this function so that it can be altered.

`pssize` is a pointer to a variable which contains the size of the stack.. The task's size is not needed by `OSTaskCreate()` but it is for `OSTaskCreateExt()`. Because `OSTaskStkInit_FPE_x86()` reserves storage for the floating-point emulation variables, the available stack size is actually altered by this function which is why a pointer to the size is passed.

`OSTaskStkInit_FPE_x86()` starts off by decomposing the TOS into its segment and offset components [L1(1)]. We then convert the address of the TOS into a linear address [L1(2)]. Remember that on the 80x86 (Real Mode), the segment is multiplied by 16 and added to the offset to form the actual memory address. We then determine the size of the stack (in number of bytes) [L1(3)]. Remember that with µC/OS-II, you must declare a stack using the `OS_STK` data type which may represent an 8-bit wide stack, a 16-bit wide stack or a 32-bit wide stack.

The linear address for the BOS is then determined by subtracting the number of bytes allocated to the stack from the TOS address [L1(4)]. You should note that I added 15 bytes to the bottom of the stack and ANDed it with `0xFFFFFFF0L` so that I would align the BOS on a 'paragraph' boundary (i.e. a 16-byte boundary).

From the BOS's linear address, we determine the new segment of the BOS [L1(5)]. A far pointer with an offset of `0x0000` is then created and assigned to the new BOS pointer [L1(6)].

To initialize the floating-point emulation variables of the task's stack, we can simply copy the bottom of the calling's task stack into the new stack [L1(7)]. You should note that the calling task MUST have also

been created from a task that has its stack initialized with the floating-point emulation variables. Failure to do this could cause unpredictable results. Note that I decided to copy 384 bytes (0x0180). It turns out that you don't need to copy this many bytes but I find it safe to add a little extra in case of expansion. This also means that your task stack MUST have at least 384 bytes PLUS the anticipated stack requirements of your task (including ISR nesting, of course).

The next step is to determine the *normalize* address of the TOS. We first need to subtract 16 bytes [L1(8)] because we aligned the stack on a page boundary. If I could guaranty that you would always align your stacks to a paragraph boundary, I would not have to do this. The new TOS is determined by making a far pointer using the new segment (found in [L1(6)]) and the new size of the stack (aligned to a paragraph) [L1(9)].

The final step is to move the BOS up by 384 bytes in case the BOS is used to perform stack checking (i.e. if your application calls OSTaskStkChk()) [L1(10)]. If you use stack checking, µC/OS-II needs to know the size of the new stack [L1(11-12)].

Figure 3 shows what OSTaskStkInit_FPE_x86() does. Note that paragraph alignment is not shown in figure 3.

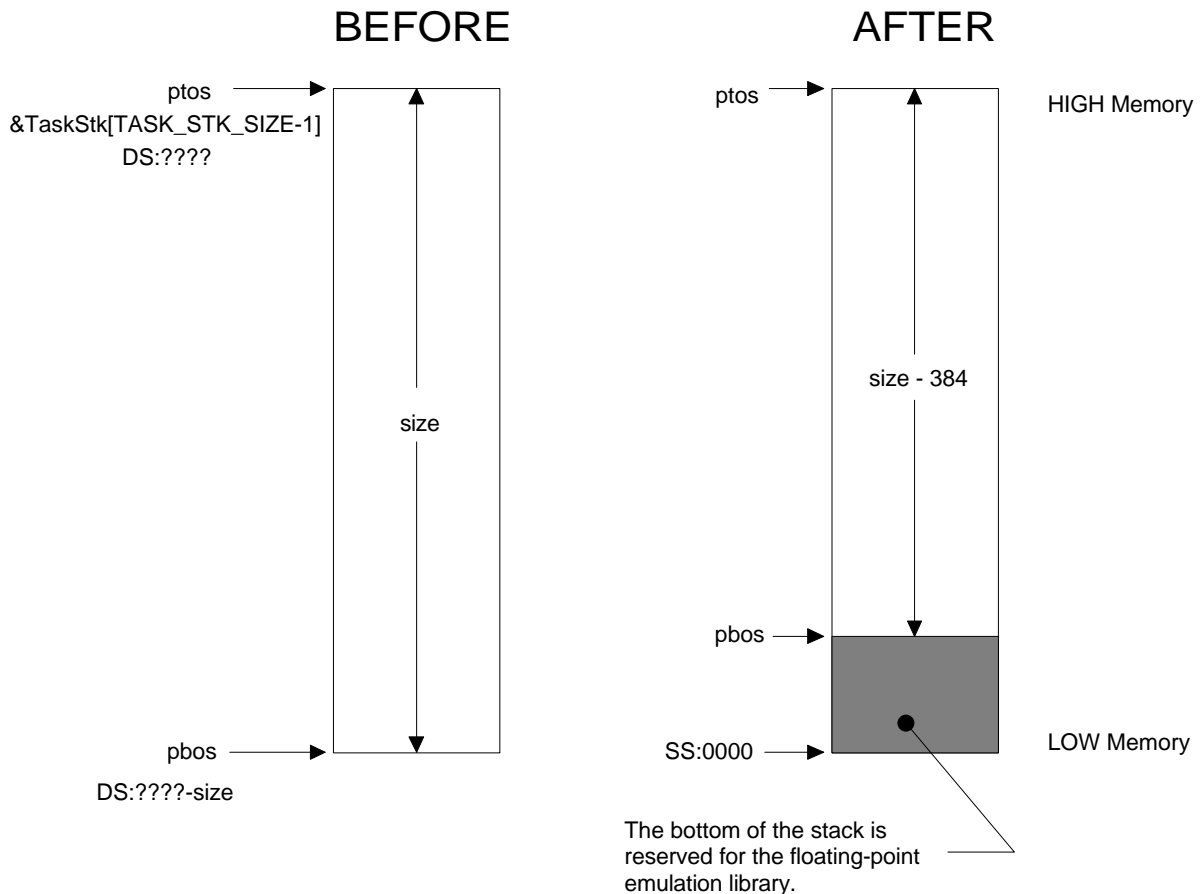


Figure 3. Normalization of the stack to support the Borland FPE library.

```

/*
*****
*
*           INITIALIZE A TASK'S STACK FOR FLOATING POINT EMULATION
*
*
* Description: This function MUST be called BEFORE calling either OSTaskCreate() or OSTaskCreateExt() in
*              order to initialize the task's stack to allow the task to use the Borland floating-point
*              emulation. The returned pointer MUST be used in the task creation call.
*
*
*           Ex.:  OS_STK TaskStk[1000];
*
*
*           void main (void)
*           {
*               OS_STK *ptos;
*               OS_STK *pbos;
*               INT32U size;
*
*
*               OSInit();
*               .
*               .
*               ptos = &TaskStk[999];
*               pbos = &TaskStk[0];
*               size = 1000;
*               OSTaskStkInit_FPE_x86(&ptos, &pbos, &size);
*               OSTaskCreate(Task, (void *)0, ptos, 10);
*               .
*               .
*               OSStart();
*           }
*
* Arguments : pptos      is the pointer to the task's top-of-stack pointer which would be passed to
*                       OSTaskCreate() or OSTaskCreateExt().
*
*           ppbos      is the pointer to the new bottom of stack pointer which would be passed to
*                       OSTaskCreateExt().
*
*           psize      is a pointer to the size of the stack (in number of stack elements). You
*                       MUST allocate sufficient stack space to leave at least 384 bytes for the
*                       floating-point emulation.
*
* Returns   : The new size of the stack once memory is allocated to the floating-point emulation.
*
* Note(s)   : 1) _SS is a Borland 'pseudoregister' and returns the contents of the Stack Segment (SS)
*              2) The pointer to the top-of-stack (pptos) will be modified so that it points to the new
*                 top-of-stack.
*              3) The pointer to the bottom-of-stack (ppbos) will be modified so that it points to the new
*                 bottom-of-stack.
*              4) The new size of the stack is adjusted to reflect the fact that memory was reserved on
*                 the stack for the floating-point emulation.
*****
*/

void OSTaskStkInit_FPE_x86 (OS_STK **pptos, OS_STK **ppbos, INT32U *psize)
{
    INT32U  lin_tos;                /* 'Linear' version of top-of-stack address */
    INT32U  lin_bos;                /* 'Linear' version of bottom-of-stack address */
    INT16U  seg;
    INT16U  off;
    INT32U  bytes;

    seg     = FP_SEG(*pptos);        /* (1) Decompose top-of-stack pointer into seg:off */
    off     = FP_OFF(*pptos);
    lin_tos = ((INT32U)seg << 4) + (INT32U)off; /* (2) Convert seg:off to linear address */
    bytes   = *psize * sizeof(OS_STK); /* (3) Determine how many bytes for the stack */
    lin_bos = (lin_tos - bytes + 15) & 0xFFFFFFF0L; /* (4) Ensure paragraph alignment for BOS */

    seg     = (INT16U)(lin_bos >> 4); /* (5) Get new 'normalized' segment */
    *ppbos  = (OS_STK *)MK_FP(seg, 0x0000); /* (6) Create 'normalized' BOS pointer */
    memcpy(*ppbos, MK_FP(_SS, 0), 384); /* (7) Copy FP emulation memory to task's stack */
    bytes   = bytes - 16; /* (8) Loose 16 bytes because of alignment */
    *pptos  = (OS_STK *)MK_FP(seg, (INT16U)bytes); /* (9) Determine new top-of-stack */
    *ppbos  = (OS_STK *)MK_FP(seg, 384); /* (10) Determine new bottom-of-stack */
    bytes   = bytes - 384; /* (11) */
    *size   = bytes / sizeof(OS_STK); /* (12) Determine new stack size */
}

```

Listing 1. Function to call before creating a task.

Final notes

You should be careful that your code doesn't generate any floating-point exception (e.g. divide by zero) because the floating-point library would not work properly under these circumstances. Run-time exceptions can, however, be avoided by adding range testing code.

References

μC/OS-II, The Real-Time Kernel

Jean J. Labrosse
R&D Technical Books, 1998
ISBN 0-87930-543-6

Contacts

Jean J. Labrosse

949 Crestview Circle
Weston, FL 33327
954-217-2036
954-217-2037 (FAX)
e-mail: Jean.Labrosse@uCOS-II.com
WEB: www.uCOS-II.com

R&D Books, Inc.

1601 W. 23rd St., Suite 200
Lawrence, KS 66046-9950
(785) 841-1631
(785) 841-2624 (FAX)
WEB: <http://www.rdbooks.com>
e-mail: rdorders@rdbooks.com