# Task State Segment

From OSDev Wiki

The Task State Segment (TSS) is a special data structure for x86 processors which holds information about a task. The TSS is primarily suited for hardware multitasking, where each individual process has its own TSS. In Software multitasking, one or two TSS's are also generally used, as they allow for entering ring0 code after an interrupt.

## Contents



## x86 Structure

| offset | 31-16 | 15-0 |
|---|---|---|
| **0x00** | reserved | LINK |
| **0x04** | ESP0 | |
| **0x08** | reserved | SS0 |
| **0x0C** | ESP1 | |
| **0x10** | reserved | SS1 |
| **0x14** | ESP2 | |
| **0x18** | reserved | SS2 |
| **0x1C** | CR3 | |
| **0x20** | EIP | |
| **0x24** | EFLAGS | |
| **0x28** | EAX | |
| **0x2C** | ECX | |
| **0x30** | EDX | |
| **0x34** | EBX | |
| **0x38** | ESP | |
| **0x3C** | EBP | |
| **0x40** | ESI | |
| **0x44** | EDI | |
| **0x48** | reserved | ES |
| **0x4C** | reserved | CS |
| **0x50** | reserved | SS |
| **0x54** | reserved | DS |
| **0x58** | reserved | FS |

| | | |
|---|---|---|
| **0x5C** | reserved | GS |
| **0x60** | reserved | LDTR |
| **0x64** | IOPB offset | reserved |

# x86_64 Structure

| offset | 31-16 | 15-0 |
|---|---|---|
| **0x00** | reserved | |
| **0x04** | RSP0 (low) | |
| **0x08** | RSP0 (high) | |
| **0x0C** | RSP1 (low) | |
| **0x10** | RSP1 (high) | |
| **0x14** | RSP2 (low) | |
| **0x18** | RSP2 (high) | |
| **0x1C** | reserved | |
| **0x20** | reserved | |
| **0x24** | IST1 (low) | |
| **0x28** | IST1 (high) | |
| **0x2C** | IST2 (low) | |
| **0x30** | IST2 (high) | |
| **0x34** | IST3 (low) | |
| **0x38** | IST3 (high) | |
| **0x3C** | IST4 (low) | |
| **0x40** | IST4 (high) | |
| **0x44** | IST5 (low) | |
| **0x48** | IST5 (high) | |
| **0x4C** | IST6 (low) | |
| **0x50** | IST6 (high) | |
| **0x54** | IST7 (low) | |
| **0x58** | IST7 (high) | |
| **0x5C** | reserved | |
| **0x60** | reserved | |
| **0x64** | IOPB offset | reserved |

The x86_64 structure is slightly different. The RSP0, RSP1 and RSP2 fields remain. RSPx is loaded in whenever an interrupt causes the CPU to change PL to x. The TSS in long mode also holds the Interrupt Stack Table, which is a table of 7 known good stack pointers that can be used for handling interrupts. You can set an interrupt vector to use an IST entry in the Interrupt Descriptor Table by giving it a number from 0 - 7. If 0 is selected, then the IST mechanism is not used. If any other number is selected then when that interrupt vector is called the CPU will load RSP from the corresponding IST entry. This is useful for handling things like double faults, since you don't have to worry about switching stacks; the CPU will do it for you.

You can also use the IST for NMIs, but if you do so then be aware that you are opening yourself up to the possibility of a race condition when NMIs nest. Imagine a scenario in which a NMI handler is called, then another exception occurs, which in turn leads to a second NMI. The second NMI will have loaded the exact same stack pointer as the first, and it will then proceed to overwrite it, so upon return to the first NMI (if you get that far) your whole stack will be corrupted. If you

support the SYSCALL instruction, you are in a bad spot. Imagine a scenario in which the syscall instruction is executed from ring 3. The CPU is then switched to ring 0. Then if an NMI occurs before your syscall handler has had a chance to switch to a kernel stack, your NMI handler will now be executing on a user controlled stack. To fix this you have little choice but to use the IST, which leads to the race condition mentioned above. However, this race condition is not a problem if you just panic in the NMI handler anyway, since the second NMI will never return.

Note that in any case, when an interrupt occurs, SS is forced to NULL and the SS selector's RPL field is set to the new CPL.

# TSS in software multitasking

For each CPU which executes processes possibly wanting to do system calls via interrupts, one TSS is required. The only interesting fields are SS0 and ESP0. Whenever a system call occurs, the CPU gets the SS0 and ESP0-value in its TSS and assigns the stack-pointer to it. So one or more kernel-stacks need to be set up for processes doing system calls. Be aware that a thread's/process' time-slice may end during a system call, passing control to another thread/process which may as well perform a system call, ending up in the same stack. Solutions are to create a private kernel-stack for each thread/process and re-assign esp0 at any task-switch or to disable scheduling during a system-call.

Setting up a TSS is straight-forward. An entry in the Global Descriptor Table is needed (see also the GDT Tutorial), specifying the TSS' address as "base", TSS' size as "limit", 0x89 (Present|Executable|Accessed) as "access byte" and 0x40 (Size-bit) as "flags". In the TSS itself, the members "SS0", "ESP0" and "IOPB offset" are to be set:

- SS0 gets the kernel datasegment descriptor (e.g. 0x10 if the third entry in your GDT describes your kernel's data)
- ESP0 gets the value the stack-pointer shall get at a system call
- IOPB may get the value sizeof(TSS) (which is 104) if you don't plan to use this io-bitmap further (according to mystran in http://forum.osdev.org/viewtopic.php?t=13678)

The actual loading of the TSS must take place in protected mode and after the GDT has been loaded. The loading is simple as:

```
mov ax, 0x??  ;The descriptor of the TSS in the GDT (e.g. 0x28 if the sixths en
ltr ax        ;The actual load
```

# See Also

- GDT Tutorial
- System Calls
- Getting to Ring 3

## Threads

- Do I need a TSS?

## External Links

- Task State Segment on Wikipedia