µC/OS-II for the Philips XA

Application Note

AN-1000

Jean J. Labrosse
Jean.Labrosse@uCOS-II.com
www.uCOS-II.com

Acknowledgements

I would like to thank *Tasking Software* for providing me with their fine compiler (V3.0r0) and CrossView Debugger and Instruction Set Simulator (ISS). The port for µC/OS-II was testing with the ISS. I would also like to extend special thanks to Mr. Wieant Nielander (wieant_nielander@tasking.com), Product Manager at *Tasking Software BV* for the exceptional support he has given me in getting µC/OS-II to work with the XA and their tools.

Finally, I would like to thank Mr. Bill Gray (BillG@ADIC.com) for his original contribution of the XA port for the Tasking Compiler.

Summary

A real-time kernel is software that manages the time of a microprocessor or microcontroller to ensure that all time critical events are processed as efficiently as possible. This application note describes how a real-time kernel, µC/OS-II works with the Philips XA microcontroller. The application note assumes that you are familiar with the XA and the C programming language.

Introduction

A real-time kernel allows your project to be divided into multiple independent elements called *tasks*. A task is a simple program which competes for CPU time. With most real-time kernels, each task is given a priority based on its importance. When you design a product using a real-time kernel you split the work to be done into tasks which are responsible for a portion of the problem. A real-time kernel also provides valuable services to your application such as time delays, system time, message passing, synchronization, mutual-exclusion and more.

Most real-time kernels are *preemptive*. A preemptive kernel ensures that the highest-priority task ready-to-run is always given control of the CPU. When an ISR (Interrupt Service Routine) makes a higher-priority task ready-to-run, the higher-priority task will be given control of the CPU as soon as all nested interrupts complete. The execution profile of a system designed using a preemptive kernel is illustrated in figure 1. As shown, a low-priority task is executing ①. An asynchronous event interrupts the microprocessor ②. The microprocessor services the interrupt ③ which makes a high-priority task ready for execution. Upon completion, the ISR invokes a service provided by the kernel which decides to return to the high-priority task instead of the low-priority task ④. The high-priority task executes to completion, unless it also gets interrupted ⑤. At the end of the high-priority task, the kernel resumes the low-priority task ⑥. As you can see, the kernel ensures that time critical tasks are performed first. Furthermore, execution of time critical tasks are deterministic and are almost insensitive to code changes. In fact, in many cases, you can add low-priority tasks without affecting the responsiveness of you system to high-priority tasks. During normal execution, a low-priority task can make a higher-priority task ready for execution. At that point, the kernel immediately suspends execution of the lower priority task in order to resume the higher priority one.

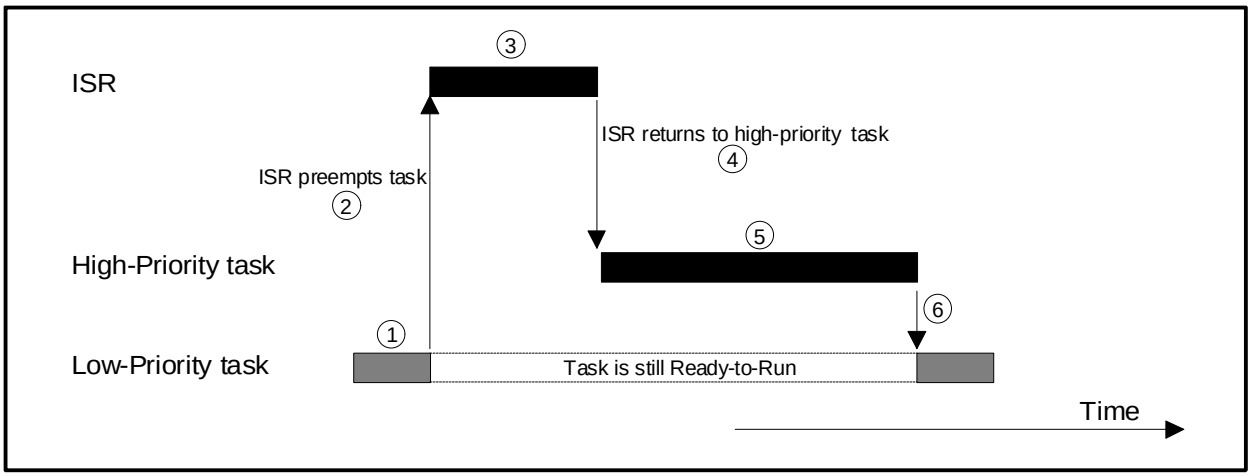

Figure 1, Execution profile of a preemptive kernel such as µC/OS-II.

A real-time kernel basically performs two operations: *Scheduling* and *Context Switching*. Scheduling is the process of determining whether there is a higher priority task ready to run. When a higher-priority task needs to be executed, the kernel must save all the information needed to eventually resume the task that is being suspended. The information saved is called the *task context*. The task context generally consist of most, if not all, CPU registers. When switching to a higher priority task, the kernel perform the reverse process by loading the context of the new task into the CPU so that the task can resume execution where it left off.

The Philips XA and Real-Time Kernels

The XA has a number of interresting features which makes it particularly well suited for real-time kernels.

When you use a kernel, each task requires its own stack space. The size of the stack required for each task is application specific but basically depends on function call nesting, allocation of local variables for each function and the worst case interrupt requirements. Unlike other processors, the XA provides two stack spaces: a *System Stack* and a *User Stack*. The System Stack is automatically used when processing interrupts and exceptions. The User Stack is used by your application tasks for subroutine nesting and storage of local variables. The most important benefit of using two stacks is that you don't need to allocate extra space on the stack of each task to accommodate for interrupt nesting. This feature greatly reduces the amount of RAM needed in your product. With the XA, the total amount of RAM needed just for stacks is given by:

$$TotalRAM_{Stack} = ISRStack_{Max} + \sum_{i=1}^{n} TaskStack_i$$

The XA divide its 16 MBytes of data address into 256 *segments* of 64 Kbytes. The stack for each task can be isolated from each other by having them reside in their own segment. The XA protects each stack by preventing a task from accessing another task's stack. This feature can prevent an errant task from corrupting other tasks.

Scheduling and task-switching can eat up valuable CPU time which directly translates to overhead. A processor with an efficient instruction set such as that found on the XA helps reduce the time spent performing scheduling and context switching. For instance, the XA provides two instructions to PUSH and POP multiple registers onto and from the stack, respectively. This feature makes for a fast context switch because all seven registers (R0 through R6) can be saved and restored onto and from the stack in just 42 clock cycles whereas it would take 70 clock cycles to perform the same function with individual PUSH and POP instructions.

µC/OS-II

µC/OS-II (pronounced *micro C OS version 2*) is a portable, ROMable, preemptive, real-time, multitasking kernel and can manage up to 63 tasks. The internals of µC/OS-II are described in my book called: ***MicroC/OS-II, The Real-Time Kernel***. The book also includes a floppy disk containing all the source code. µC/OS-II is written in C for sake of portability, however, microprocessor specific code is written in assembly language. Assembly language and microprocessor specific code is kept to a minimum. µC/OS-II is comparable in performance with many commercially available kernels. The execution time for every service provided by µC/OS-II (except one) is both deterministic and constant. µC/OS-II allows you to:

- Create and manage up to 63 tasks, delete tasks, change the priority of tasks,
- Suspend and resume tasks,
- Create and manage binary or counting semaphores,
- Delay tasks for integral number of ticks, or for a user specified amount of hours, minutes, seconds and milliseconds,
- Lock/Unlock the scheduler,
- Create and manage fixed-sized memory blocks and,
- Send messages from an ISR or a task to other tasks.

Using µC/OS-II

µC/OS-II requires that you call `OSInit()` before you start using any of the other services provided by µC/OS-II. After calling `OSInit()` you will need to create at least one task before you start multitasking (i.e. before calling `OSStart()`). All tasks managed by µC/OS-II needs to be *created*. You create a task by simply calling a service provided by µC/OS-II (described later). You need to create each task in order to prepare them for multitasking. If you want, you can create all your tasks before calling `OSStart()`. Once multitasking starts, µC/OS-II will start executing the highest priority task that has been created. You should note that interrupts will be enabled as soon as the first task starts execution. Your `main()` function will thus look as shown in listing 1.

```
void main (void)
{
    /* Perform XA Initializations                          */
    .
    .
    OSInit();
    .
    .
    /* Create at least one task by calling OSTaskCreate()  */
    .
    .
    OSStart();
}
```

Listing 1, Initializing µC/OS-II and starting multitasking.

A task under µC/OS-II must always be written as an infinite loop as shown in listing 2. When your task first executes, it will be passed an argument (`pdata`) which can be made to point to task specific data when the task is created. If you don't use this feature, you should simply equate `pdata` to `pdata` as shown below to prevent the compiler from generating a warning. Even though a task is an infinite loop, it must not use up all of the CPU's time. To allow other tasks to get a chance to execute, you have to write

each task such that the task either suspends itself until some amount of time expires, wait for a semaphore, wait for a message from either another task or an ISR or simply suspend itself indefinitely until explicitely resumed by another task or an ISR. µC/OS-II provides services to accomplish this.

```
void UserTask (void *pdata)
{
    pdata = pdata;
    /* User task initialization                                  */
    while (1) {
        /* User code goes here                                   */

        /* You MUST invoke a service provided by µC/OS-II to: */
        /* ... a) Delay the task for 'n' ticks                   */
        /* ... b) Wait on a semaphore                            */
        /* ... c) Wait for a message from a task or an ISR       */
        /* ... d) Suspend execution of this task                 */
    }
}
```

Listing 2, Layout of a task under µC/OS-II.

A task is created by calling the `OSTaskCreate()` function. `OSTaskCreate()` requires four arguments as shown in the function prototype of listing 3.

```
INT8U OSTaskCreate (void  (*task)(void *pd),
                    void   *pdata,
                    OS_STK *ptos,
                    INT8U   prio);
```

Listing 3, Function prototype of OSTaskCreate().

`task` is a pointer to the task you are creating. *`pdata`* is a pointer to an optional argument that you can pass to your task when it begins execution. This feature allows you to write a generic task which is personalized based on arguments passed to it. For example, you can design a generic serial port driver task which gets passed a pointer to a structure defining the ports parameters such as the address of the port, its interrupt vector, the baud rate etc. *`ptos`* is a pointer to the task's top-of-stack. Finally, *prio* is the task's priority. With µC/OS-II, each task must have a unique priority. The lower the priority number, the more important the task is. In other words, a task having a priority of 10 is more important than a task with a priority of 20.

With µC/OS-II, each task can have a different stack size. This feature greatly reduces the amount of RAM needed because a task with a small stack requirement doesn't get penalized because another task in your system requires a large amount of stack space. You should note that you can locate a task's stack just about anywhere in the XA's address space. This is accomplished by specifying the task's top-of-stack through a constant (or a #define) as shown in the two examples of listing 4.

```
OS_STK Task1Stk[1000];
OS_STK Task2Stk[500];


INT8U OSTaskCreate(Task1, pdata1, (OS_STK *)&Task1Stk[999], prio1);
INT8U OSTaskCreate(Task2, pdata2, (OS_STK *)&Task2Stk[499], prio2);
```

Listing 4, Task stacks.

`OSTaskCreate()` returns a value back to its caller to notify it about whether the task creation was successful or not. When a task is created, µC/OS-II assigns a *Task Control Block (TCB)* to the task. The TCB is used by µC/OS-II to store the priority of the task, the current state of the task (ready, waiting for an event, delayed, etc.), the current location of the task's top-of-stack and, other kernel related data.

µC/OS-II also provides a more powerful function to create a task, `OSTaskCreateExt()`. Details about this function are found in my book: *MicroC/OS-II, The Real-Time Kernel* (ISBN 0-87930-543-6).

Table 1 shows the function prototypes of some of the services provided by µC/OS-II. µC/OS-II actually provides The prototypes are shown in tabular form for sake of discussion. The actual prototype of OSTimeDly() for example is actually:

```
void OSTimeDly(INT16U ticks);
```

Listing 5, Delaying a task until time expires.

You will notice that every function starts with the letters 'OS'. This makes it easier for you to know that the function call is related to a kernel service (i.e. an **O**perating **S**ystem call). Also, the function naming convention groups services by functions: 'OSTask...' are task management functions, 'OSTime...' are time management functions, etc. Another item you should notice is that non-standard data types are in upper-case: INT8U, INT16U, INT32U and OS_EVENT. INT8U, INT16U and INT32U represent an *unsigned 8-bit value*, an *unsigned 16-bit value*, and an *unsigned 32-bit value*, respectively. OS_EVENT is a typedef'ed data structure declared in uCOS_II.H and is used to hold information related to semaphore, message mailboxes and message queues. Your application will in fact have to declare storage for a pointer to this data structure as follows:

```
OS_EVENT *MySem;
```

Listing 6, Pointer to Event Control Block.

This indicates that the pointer `MySem` will be able to access the OS_EVENT data structure which may be located in another bank. OS_EVENT is used in the same capacity as the FILE data-type used in standard C library. `OSSemCreate()`, `OSMboxCreate()` and `OSQCreate()` return a pointer which is used to identify the semaphore, mailbox or queue, respectively.

µC/OS-II (Philips XA, Large Model)						
Return Value	**Function Name**	**Argument #1**	**Argument #2**	**Argument #3**	**Argument #4**	**Called From...**
Initialization						
void	**OSInit**	void	-	-	-	main()
void	**OSStart**	void	-	-	-	main()
void	**OSStatInit**	void	-	-	-	main()
Miscellaneous						
void	**OSSchedLock**	void	-	-	-	Task or ISR
void	**OSSchedUnlock**	void	-	-	-	Task or ISR
INT16U	**OSVersion**	void	-	-	-	Task or ISR
Task Management						
INT8U	**OSTaskChangePrio**	INT8U oldprio	INT8U newprio	-	-	Task
INT8U	**OSTaskCreate**	void (task)(void *pd)	void *pdata	OS_STK *pstk	INT8U prio	main() or Task
INT8U	**OSTaskCreateExt**	void (task)(void *pd)	void *pdata	OS_STK *pstk	INT8U prio	main() or Task
		INT16U id	OS_STK *pbos	INT32U stk_size	void *pext	
		INT16U opt				
INT8U	**OSTaskDel**	INT8U prio	-	-	-	Task
INT8U	**OSTaskDelReq**	INT8U prio	-	-	-	Task
INT8U	**OSTaskQuery**	INT8U prio	OS_TCB *pdata	-	-	Task or ISR
INT8U	**OSTaskResume**	INT8U prio	-	-	-	Task or ISR
INT8U	**OSTaskStkChk**	INT8U prio	INT32U *pfree	INT32U *pused	-	Task
INT8U	**OSTaskSuspend**	INT8U prio	-	-	-	Task or ISR
Time Management						
void	**OSTimeDly**	INT16U ticks	-	-	-	Task
void	**OSTimeDlyHMSM**	INT8U hours	INT8U minutes	INT8U seconds	INT8U milli	Task
INT8U	**OSTimeDlyResume**	INT8U prio	-	-	-	Task
INT32U	**OSTimeGet**	void	-	-	-	Task or ISR
void	**OSTimeSet**	INT32U ticks	-	-	-	Task or ISR
void	**OSTimeTick**	void	-	-	-	Task or ISR
Semaphore Management						
INT16U	**OSSemAccept**	OS_EVENT *pevent	-	-	-	Task or ISR
OS_EVENT *	**OSSemCreate**	INT16U value	-	-	-	Task
void	**OSSemPend**	OS_EVENT *pevent	INT16U timeout	INT8U *err		Task
INT8U	**OSSemPost**	OS_EVENT *pevent	-	-	-	Task or ISR
INT8U	**OSSemQuery**	OS_EVENT *pevent	OS_SEM_DATA *pdata	-	-	Task or ISR
Message Mailbox Management						
void *	**OSMboxAccept**	OS_EVENT *pevent	-	-	-	Task or ISR
OS_EVENT *	**OSMboxCreate**	void *msg	-	-	-	Task
void *	**OSMboxPend**	OS_EVENT *pevent	INT16U timeout	INT8U *err	-	Task
INT8U	**OSMboxPost**	OS_EVENT *pevent	void *msg	-	-	Task or ISR
INT8U	**OSMboxQuery**	OS_EVENT *pevent	OS_MBOX_DATA *pdata	-	-	Task or ISR
Message Queue Management						
void *	**OSQAccept**	OS_EVENT *pevent	-	-	-	Task or ISR
OS_EVENT *	**OSQCreate**	void **start	INT8U size	-	-	Task
INT8U	**OSQFlush**	OS_EVENT *pevent	-	-	-	Task
void *	**OSQPend**	OS_EVENT *pevent	INT16U timeout	INT8U *err	-	Task
INT8U	**OSQPost**	OS_EVENT *pevent	void *msg	-	-	Task or ISR
INT8U	**OSQPostFront**	OS_EVENT *pevent	void *msg	-	-	Task or ISR
INT8U	**OSQQuery**	OS_EVENT *pevent	OS_Q_DATA *pdata	-	-	Task or ISR
Memory Management						
OS_MEM *	**OSMemCreate**	void *addr	INT32U nblks	INT32U blksize	INT8U *err	Task or startup
void *	**OSMemGet**	OS_MEM *pmem	INT8U *err	-	-	Task or ISR
INT8U	**OSMemPut**	OS_MEM *pmem	void *pblk	-	-	Task or ISR
INT8U	**OSMemQuery**	OS_MEM *pmem	OS_MEM_DATA *pdata	-	-	Task or ISR
Interrupt Management						
void	**OSIntEnter**	void	-	-	-	ISR
void	**OSIntExit**	void	-	-	-	ISR

Table 1, List of services provided by µC/OS-II

µC/OS-II and the Philips XA

µC/OS-II was ported to the XA using the *Tasking XA* tool chain and the complete source code for both µC/OS-II and the port to the XA's *Large Memory Model* are available from www.uCOS-II.com. The large memory model allows you to write very large applications (up to 16 Mbytes of code) and access a lot of data memory (up to 16 Mbytes). The XA port has been tested on the *Future Design, Inc.* XTEND-G3 evaluation board and the test code provided with the port is assumed to run on this target. The test code can, however, be easily modified to support other environments. The large model requires that your XTEND board has at least two pages of data RAM. In other words, you must have more than 64K bytes of RAM in the XA's addressable data area. This can be easily accomplished by replacing the two 32K bytes data RAM chips with two 128K bytes chips.

A number of assumptions have been made about how µC/OS-II uses the XA. µC/OS-II will run the XA in *Native Mode*. This allows the compiler to use as many new features of the XA as possible and does not make any effort to be backwards compatible with the 80C51.

Your application code and most of µC/OS-II services will be executing in *User* mode. The XA will automatically be placed in *System* mode when either an interrupt, an exception occurs or, when µC/OS-II performs a context switch. Each of your application task will require its own stack space in *Banked RAM* while all interrupts will share the system mode stack. µC/OS-II allows you to specify a different stack size for each task. In other words, µC/OS-II doesn't require that the stack for each task be the same size. This feature prevents you from waisting valuable RAM when the stack requirements for each task varies.

µC/OS-II will only manipulate the registers in bank #0. If your application code changes register bank, you will need to ensure that your code restores register bank #0 prior to using any of µC/OS-II's services.

µC/OS-II requires a periodic interrupt source to maintain system time and provide time delay and timeout services. This periodic interrupt is called a *System Tick* and needs to occur between 10 and 100 times per second. The system tick can be generated by using any of the XA's three internal timers or externally through the INT0 or INT1 inputs. For lack of a better choice, I used timer #0 and configured it for a 100 Hz tick rate. The tick interrupt vectors to an assembly language function called `OSTickISR`. If you application requires the use of all of the XA's timers then you will have to find another source for the 'ticker'. For example, if your system is powered from a power grid, you can bring the line frequency (50 or 60 Hz) in through either INT0 or INT1.

µC/OS-II also requires one of the 16 TRAP vectors in order to perform a context switch. I decided to use `TRAP #15` which is defined in the C macro `OS_TASK_SW()`. A context switch will force the XA into system mode and push the return address and the PSW onto the system stack.

As previously mentioned, you must prepare your tasks for multitasking by calling `OSTaskCreate()`. `OSTaskCreate()` builds the stack frame for the task being created as illustrated in figure 2. DS:USP indicates that once the task gets to execute, the stack will be located in the bank selected by the DS register at an offset supplied by the USP. You should note that pointers in the large model are 32-bits but, only the least significant 24 bits are used. `OSTaskCreate()` first sets up the stack to make it look as if your task has just been called by another C function ①. In other words, when your task first executes, it will think it was called by another function since the stack pointer will point as shown in ②. For the Tasking compiler, the address of `pdata` is passed in registers R1 (upper 16-bits) and R0 (lower 16-bits). `OSTaskCreate()` then simulates the stacking order of a `PUSHU R0-R6` instruction ③ which is needed for a context switch. The initial value of each register is set to the values shown for debugging purposes and can thus be changed as needed. Next, `OSTaskCreate()` stacks both the ES register and the SSEL register ④. Even though both the ES and SSEL registers are 8-bit, they are stacked as two 16-bit values because all XA stacking operations are 16-bit. The SSEL register is initialized to 0x80 to allow

your task to read and write data anywhere in the 16-MBytes data address space. You may not want to change the initial value of the SSEL register because the compiler will not know that write through the ES register is not allowed (run-time) but, it will generate code (compile-time) as if it was. During an interrupt or a context switch, the XA pushes the PC and the PSW onto the system stack. The stacking order of these registers as shown on the stack frame of figure 2 is reversed because `OSTaskCreate()` simulates a move of these registers from the system stack to the user stack ⑤.

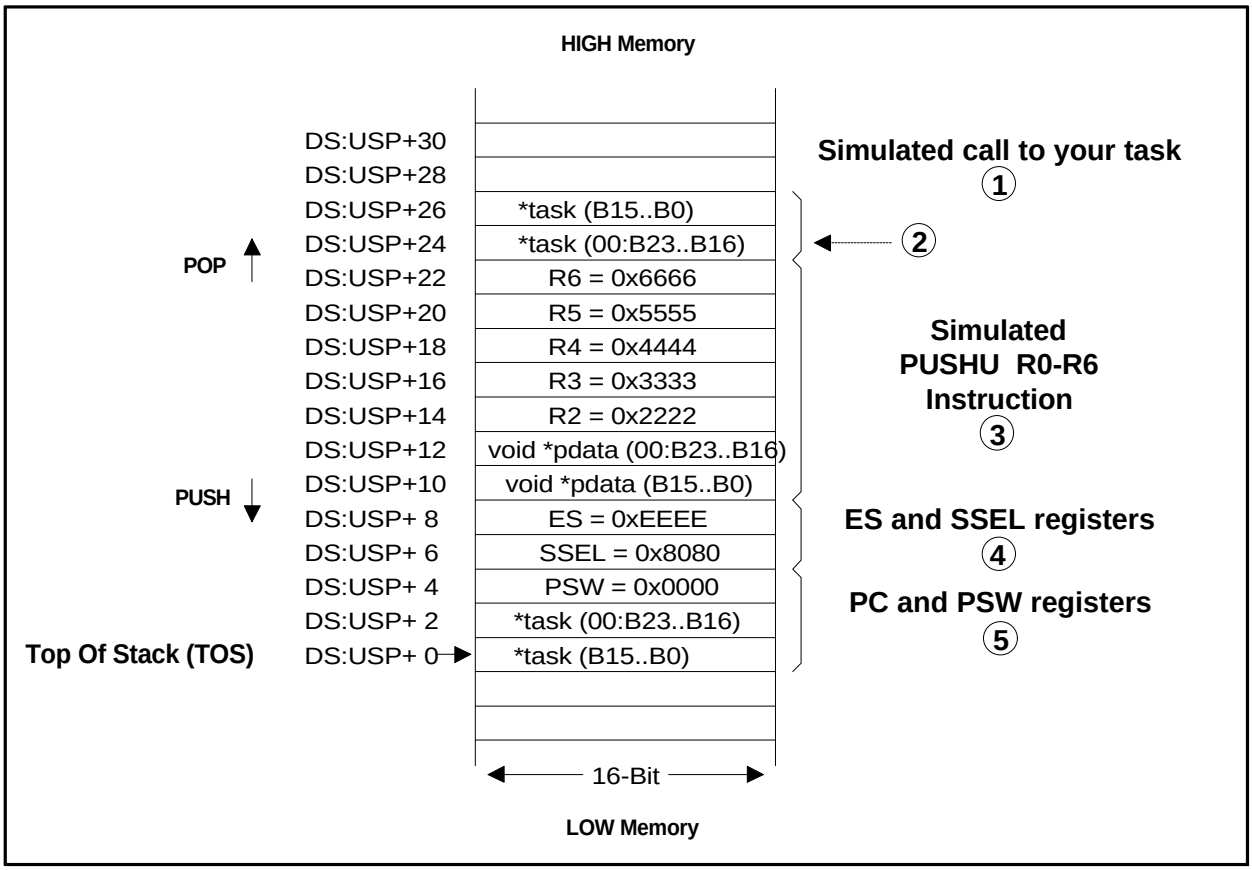

Figure 2, Stack frame of a task when a task is created.

As previously mentioned, multitasking starts when you call `OSStart()`. Figure 3 illustrates the process. `OSStart()` finds the TCB of the highest priority task that you created, loads the pointer `OSTCBHighRdy` to point to that TCB ① and calls the assembly language function `OSStartHighRdy()`. `OSStartHighRdy()` loads the USP and the DS register from the task's TCB ② and then moves the start address of your task, along with the PSW from the user stack to the system stack ③. `OSStartHighRdy()` then pops the remaining registers from the user stack ④ and finally, `OSStartHighRdy()` executes a return from interrupt which loads the PC and PSW from the system stack ⑤ into the XA. Because the PSW was initialized to 0x0000, the XA will now execute the first instructions of your task in user mode with all interrupts enabled.

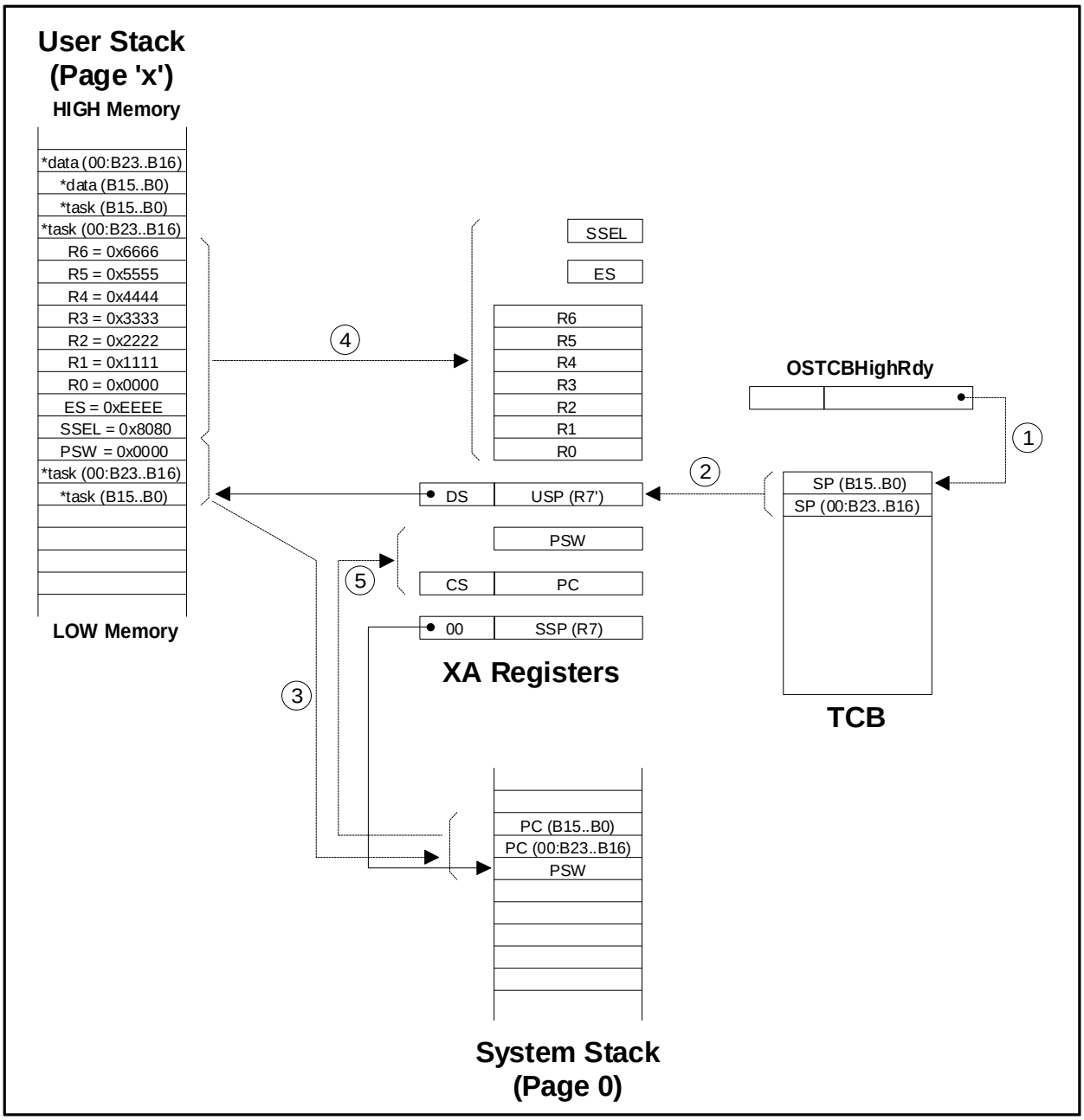

Figure 7

Context switching with µC/OS-II

Because µC/OS-II is a preemptive kernel, it always executes the highest priority task that is ready to run. As your tasks execute they will eventually invoke a service provided by µC/OS-II to either wait for time to expire, wait on a semaphore or wait for a message from another task or an ISR. A context switch will result when the outcome of the service is such that the currently running task cannot continue execution. For example, figure 4 shows what happens when a task decides to delay itself for a number of ticks. In ①, the task calls `OSTimeDly()` which is a service provided by µC/OS-II. `OSTimeDly()` places the task in a list of tasks waiting for time to expire ②. Because the task is no longer able to execute, the scheduler (`OSSched()`) is invoked to find the next most important task to run ③. A context switch is

performed by issuing a TRAP #15 instruction ④. The function `OSCtxSw()` is written entirely in assembly language because it directly manipulates XA registers. All execution times are shown assuming a 24 MHz crystal and the large model. The highest priority task executes at the completion of the XA's RETI instruction ⑤.

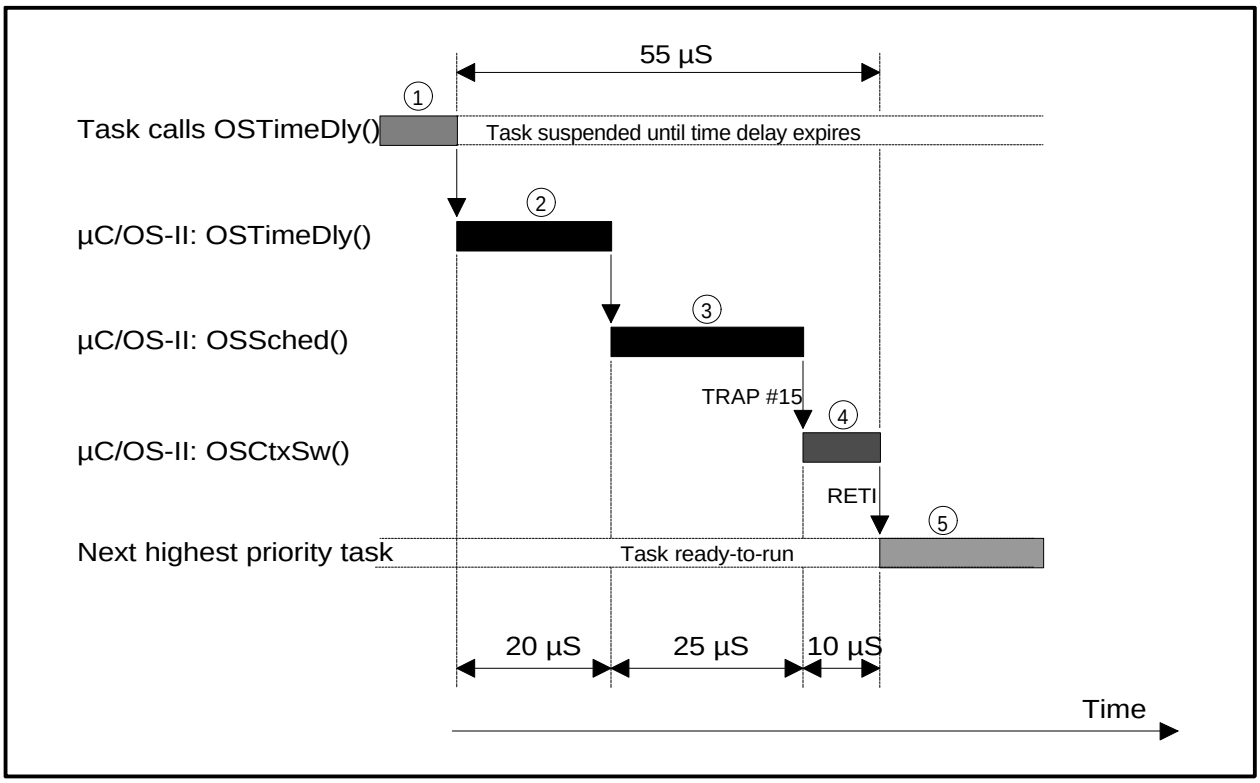

Figure 4

The work done by `OSCtxSw()` is illustrated in figure 5. The scheduler loads `OSTCBHighRdy` with the address of the new task's TCB ① before invoking `OSCtxSw()` which is done through the `TRAP #15` instruction. `OSTCBCur` already points to the TCB of the task to suspend. The `TRAP #15` instruction automatically pushes the return address and the PSW onto the system stack ②. `OSCtxSw()` starts off by saving the remainder of the XA's registers onto the user stack ③ and then, moves the saved PSW and PC from the system stack to the user stack ④. The final step in saving the context of the task to be suspended is to store the top-of-stack into the current task's TCB ⑤. The second half of the context switch operation restores the context of the new task. This is performed in the following four steps. First, the user stack pointer is loaded with the new task's top-of-stack ⑥. Second, the PC and PSW of the task to resume is moved from the user stack to the system stack ⑦. Third, the remainder of the XA's registers are restored from the user stack ⑧. Finally, a return from interrupt instruction (RETI) is executed ⑨ to retrieve the new task's PC and PSW from the system stack which causes the new task to resume execution where it left off. As shown in figure 4, a context switch for the large model takes only about 10 µS at 24 MHz.

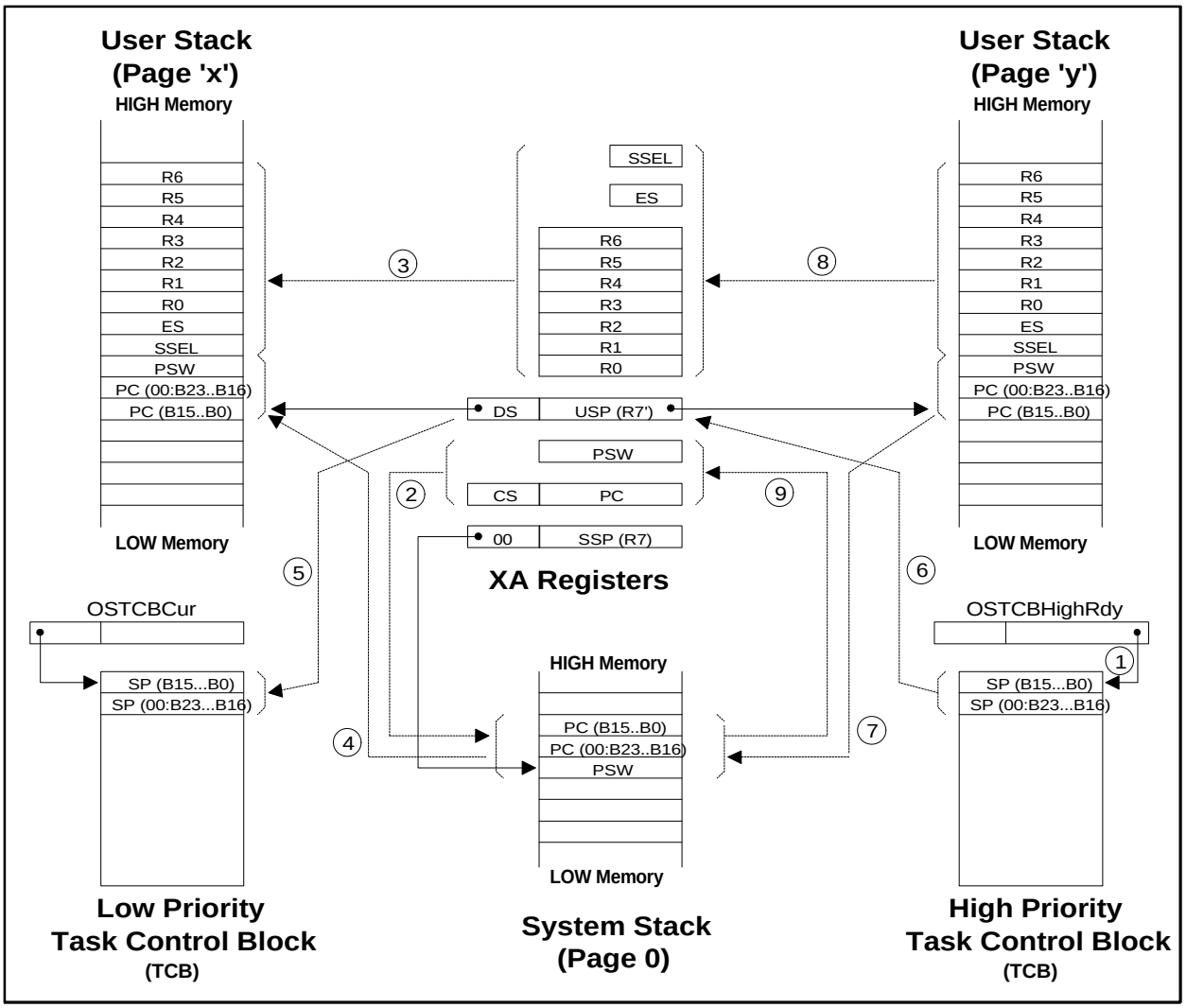

Figure 5

Interrupt Service Routines (ISRs) and µC/OS-II

Under µC/OS-II, you must write your ISRs as shown in listing 7. This code assumes you are using the Tasking C compiler and assembler. As you can see, you must prefix the function name and return type with three attributes: `_interrupt()`, `_using()` and `_frame()`.

The `_interrupt()` attribute specifies the interrupt vector which is assigned to the interrupt handler. The vector to use determines on the device and you should consult the specific XA processor you are using for additional details. The '??' in the `_interrupt()` attribute of listing 7 is the vector number of your interrupting device.

The `_using()` attribute specifies the contents of the `PSW` when your interrupt handler executes. You should always specify `0x8F00` to prevent other interrupts from interrupting the current interrupt handler until you have at least incremented `OSIntNesting`.

The `_frame()` attribute specifies which registers are pushed onto the stack by the compiler at the beginning of the ISR. You MUST specify an empty list because `CPUPushAllToUserStk()` (described next) takes care of this for you.

You must always save all the registers at the beginning of the ISR and restore them at the completion of the ISR. An 'in-line' function `CPU_PushAllToUserStk()` (see OS_CPU.H) is provided to help you do this. You must also always notify µC/OS-II when you are starting to process an ISR by either calling `OSIntEnter()` or incrementing the global variable `OSIntNesting`. You then need to provide code to process the interrupting device. Note that you are responsible for clearing the interrupting device (if needed). When you are done processing the interrupt, you must call `OSIntExit()`. `OSIntExit()` decrements the nesting counter and, when the nesting counter reaches `0`, all interrupts have nested and the scheduler is invoked to determine whether the ISR needs to return to the interrupted task or, whether a higher priority task has been made ready to run by one of the ISRs. If there is a higher priority task, µC/OS-II will need to perform a context switch to return to the more important task. In this case, `OSIntExit()` will not return to your ISR. If, however, `OSIntExit()` returns to your interrupt, it means that no other task has a higher priority than the interrupted task. In this case, your ISR code must call `CPU_PopAllFromUserStk()` (see OS_CPU.H) to restore the state of the registers from the interrupted task. You should note that you don't need to insert an `RETI` (Return From Interrupt) instruction after restoring the registers because the Tasking compiler takes care of this for you (because of the `_interrupt()` attribute).

```
_interrupt(??)  _using(0x8F00)  _frame()  void  YourTickISR (void)
{
    CPU_PushAllToUserStk();

    OSIntNesting++;

    /* Call function to handle the interrupting device */

    OSIntExit();

    CPU_PopAllFromUserStk();
}
```

Listing 7

The stack frames (system and user) during an interrupt is shown in figure 6. Items ①, ② and ③ are performed at the beginning of your ISR. When you call `OSIntExit()`, the return address is pushed onto the system stack. If a context switch is needed, `OSIntExit()` calls `OSIntCtxSw()` which also causes its return address to be pushed onto the stack. In order for `OSIntCtxSw()` to properly perform a context switch, the stack pointer (R7) needs to be adjusted so that it points as shown in figure 6, ④. The adjustment value of the stack pointer depends on the compiler model and the compiler options selected. The value is, however, at least 8 (for the large model) because of the two return addresses. If your application crashes you may want to make sure that you have the proper value for this constant. The rest of the context switch is exactly the same as previously discussed.

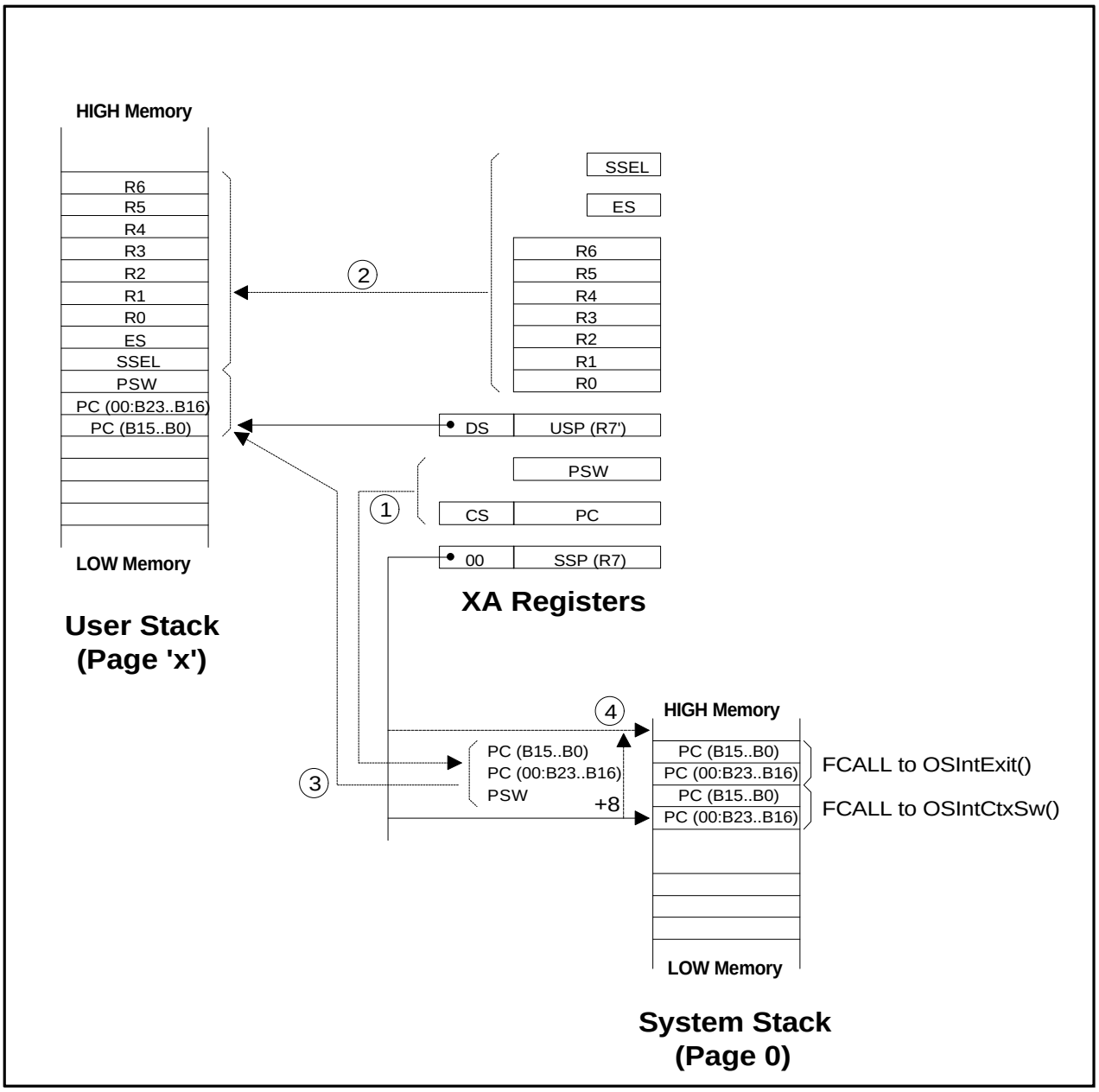
HIGH Memory
R6
R5
R4
R3
R2
R1
R0
ES
SSEL
PSW
PC (00:B23..B16)
PC (B15..B0)
LOW Memory
User Stack
(Page 'x')
SSEL
ES
R6
R5
R4
R3
R2
R1
R0
DS
USP (R7')
PSW
CS
PC
00
SSP (R7)
XA Registers
1
2
3
4
HIGH Memory
PC (B15..B0)
PC (00:B23..B16)
PSW
+8
PC (B15..B0)
PC (00:B23..B16)
PC (B15..B0)
PC (00:B23..B16)
FCALL to OSIntExit()
FCALL to OSIntCtxSw()
LOW Memory
System Stack
(Page 0)

Figure 6

References:

µC/OS-II, The Real-Time Kernel
Jean J. Labrosse
R&D Technical Books, 1998
ISBN 0-87930-543-6

Embedded Systems Building Blocks, Complete and Ready-to-Use Modules in C
Jean J. Labrosse
R&D Books, 1995
ISBN 0-87930-440-5

16-bit 80C51XA Microcontrollers (eXtended Architecture)
Philips Semiconductors, Inc.
Data Book IC25, 1996

Contacts:

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

Philips Semiconductors, Inc.
811 E. Arques Avenue
Sunnyvale, CA 94088-3409
(408) 991-2000
WEB: http://www.semiconductors.philips.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

Tasking Inc.
333 Elm Street
Dedham, MA 02026-4530
USA
800-458-8276
781-320-9400
781-320-9212 (FAX)
WEB: http://www.tasking.com
e-mail: support_us@tasking.com

Notes: