APPLICATION NOTE

AN711
µC/OS for the Philips XA

Author: Jean Labrosse

1996 Sep 06

Philips
Semiconductors

PHILIPS

µC/OS for the Philips XA AN711

Author: Jean J. Labrosse

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 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.

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.

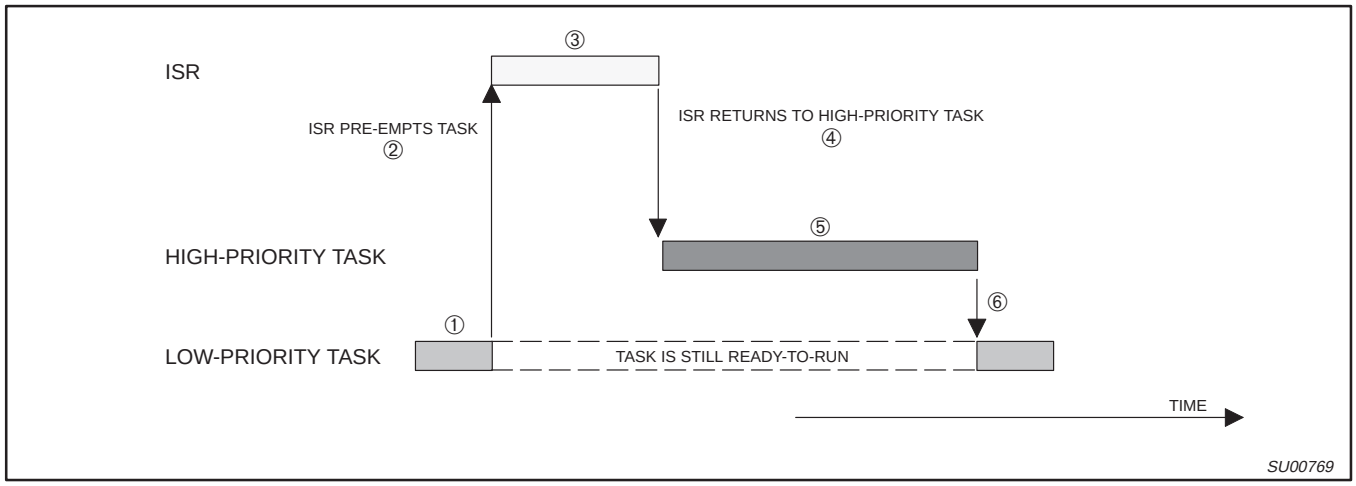

Figure 1.

THE PHILIPS XA AND REAL–TIME KERNELS

The XA has a number of interesting 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

µC/OS (pronounced *micro C OS*) is a portable, ROMable, preemptive, real–time, multitasking kernel and can manage up to 63 tasks. The internals of µC/OS are described in my book called: *µC/OS, The Real–Time Kernel* [1]. The book also includes a floppy disk containing all the source code. µC/OS 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 is comparable in performance with many commercially available kernels. The execution time for every service provided by µC/OS (except one) is both deterministic and constant. µC/OS allows you to:

- Create and manage up to 63 tasks,
- Create and manage binary or counting semaphores,
- Delay tasks for integral number of ticks,
- Lock/Unlock the scheduler,
- Change the priority of tasks,
- Delete tasks,
- Suspend and resume tasks and,
- Send messages from an ISR or a task to other tasks.

USING µC/OS

µC/OS requires that you call `OSInit()` before you start using any of the other services provided by µC/OS. 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 needs to be **created**. You create a task by simply calling a service provided by µC/OS (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 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 Figure 2.

A task under µC/OS must always be written as an infinite loop as shown in Figure 3. 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 explicitly resumed by another task or an ISR. µC/OS provides services to accomplish this.

A task is created by calling the `OSTaskCreate()` function. `OSTaskCreate()` requires four arguments as shown in the function prototype of Figure 4.

`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. *`pstk`* is a pointer to the task's top–of–stack. Finally, *`prio`* is the task's priority. With µC/OS, each task must have a unique priority. The smaller 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, 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 Figure 5.

Here, I located *task1*'s stack at the top of page 0 while *task2*'s stack will start at offset 0xF700 of page 7 and grow downwards from there. When locating stacks using constants, you must be careful that the linker does not locate data at these memory locations. If needed, you can also locate the stacks of multiple tasks in the same page using the same technique.

`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 assigns a *Task Control Block (TCB)* to the task. The TCB is used by µC/OS 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.

Table 1 shows the function prototypes of the services provided by µC/OS, V1.09. The prototypes are shown in tabular form for sake of discussion. The actual prototype of `OSTimeDly()` for example is actually:

```
void OSTimeDly(UWORD ticks);
```

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: UBYTE, UWORD, ULONG and OS_EVENT. UBYTE, UWORD and ULONG represent an *unsigned–byte* (8–bit), an *unsigned–word* (16–bit), and an *unsigned–long* (32–bit), respectively. OS_EVENT is a typedef'ed data structure declared in UCOS.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:

```
far OS_EVENT *MySem;
```

The 'far' attribute is specific to the HI–TECH compiler (described later) and 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.

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

SU00770

Figure 2.

```
void UserTask(far void *pdata)
{
    pdata = pdata;
    /* User task initialization                            */
    while (1) {
        /* User code goes here                             */
        /* You MUST invoke a service provided by µC/OS 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           */
    }
}
```

SU00771

Figure 3.

```
UBYTE OSTaskCreate(void (*task)(far void *pd),
                   far void *pdata,
                   far void *pstk,
                   UBYTE prio);
```

SU00772

Figure 4.

```
UBYTE OSTaskCreate(task1, pdata1, (far void *)0x00FFFE, prio1);
UBYTE OSTaskCreate(task2, pdata2, (far void *)0x07F700, prio2);
```

SU00773

Figure 5.

Table 1. µC/OS V1.09
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()
Task Management						
UBYTE	OSTaskCreate	void (task)(far void *pd)	far void *pdata	far void *pstk	UBYTE prio	main() or Task
UBYTE	OSTaskDel	UBYTE prio	–	–	–	Task
UBYTE	OSTaskDelReq	UBYTE prio	–	–	–	Task
UBYTE	OSTaskChangePrio			–	–	Task
UBYTE	OSTaskSuspend	UBYTE prio	–	–	–	Task or ISR
UBYTE	OSTaskResume	UBYTE prio	–	–	–	Task or ISR
void	OSSchedLock	void	–	–	–	Task or ISR
void	OSSchedUnlock	void	–	–	–	Task or ISR
Time Management						
void	OSTimeDly	UWORD ticks	–	–	–	Task
UBYTE	OSTimeDlyResume	UBYTE prio	–	–	–	Task
void	OSTimeSet	ULONG ticks	–	–	–	Task or ISR
ULONG	OSTimeGet	void	–	–	–	Task or ISR
Semaphore Management						
far OS_EVENT *	OSSemCreate	UWORD value	–	–	–	Task
UWORD	OSSemAccept	far OS_EVENT *pevent	–	–	–	Task or ISR
UBYTE	OSSemPost	far OS_EVENT *pevent	–	–	–	Task or ISR
void	OSSemPend	far OS_EVENT *pevent	UWORD timeout	UBYTE *err		Task
Message Mailbox Management						
far OS_EVENT *	OSMboxCreate	far void *msg	–	–	–	Task
far void *	OSMboxAccept	far OS_EVENT *pevent	–	–	–	Task or ISR
UBYTE	OSMboxPost	far OS_EVENT *pevent	far void *msg	–	–	Task or ISR
far void *	OSMboxPend	far OS_EVENT *pevent	UWORD timeout	UBYTE *err	–	Task
Message Queue Management						
far OS_EVENT *	OSQCreate	far void **start	UBYTE size	–	–	Task
far void *	OSQAccept	far OS_EVENT *pevent	–	–	–	Task or ISR
UBYTE	OSQPost	far OS_EVENT *pevent	far void *msg	–	–	Task or ISR
far void *	OSQPend	far OS_EVENT *pevent	UWORD timeout	UBYTE *err	–	Task
Interrupt Management						
void	OSIntEnter	void	–	–	–	ISR
void	OSIntExit	void	–	–	–	ISR

µC/OS AND THE PHILIPS XA

µC/OS (V1.09) was ported to the XA using the *HI–TECH C XA* tool chain and the complete source code for both µC/OS and the port to the XA's *Large Memory Model* are available from Philips Semiconductors, Inc. 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 uses the XA. µC/OS 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 services will be executing in *User* mode. The XA will automatically be placed in *System* mode when either an interrupt or an exception occurs or, when µC/OS 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 allows you to specify a different stack size for each task. In other words, µC/OS doesn't require that the stack for each task be the same size. This feature prevents you from wasting valuable RAM when the stack requirements for each task varies.

µC/OS 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's services.

µC/OS 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 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 6. 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. `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 6 is reversed because `OSTaskCreate()` simulates a move of these registers from the system stack to the user stack ⑤.

As previously mentioned, multitasking starts when you call `OSStart()`. Figure 7 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.

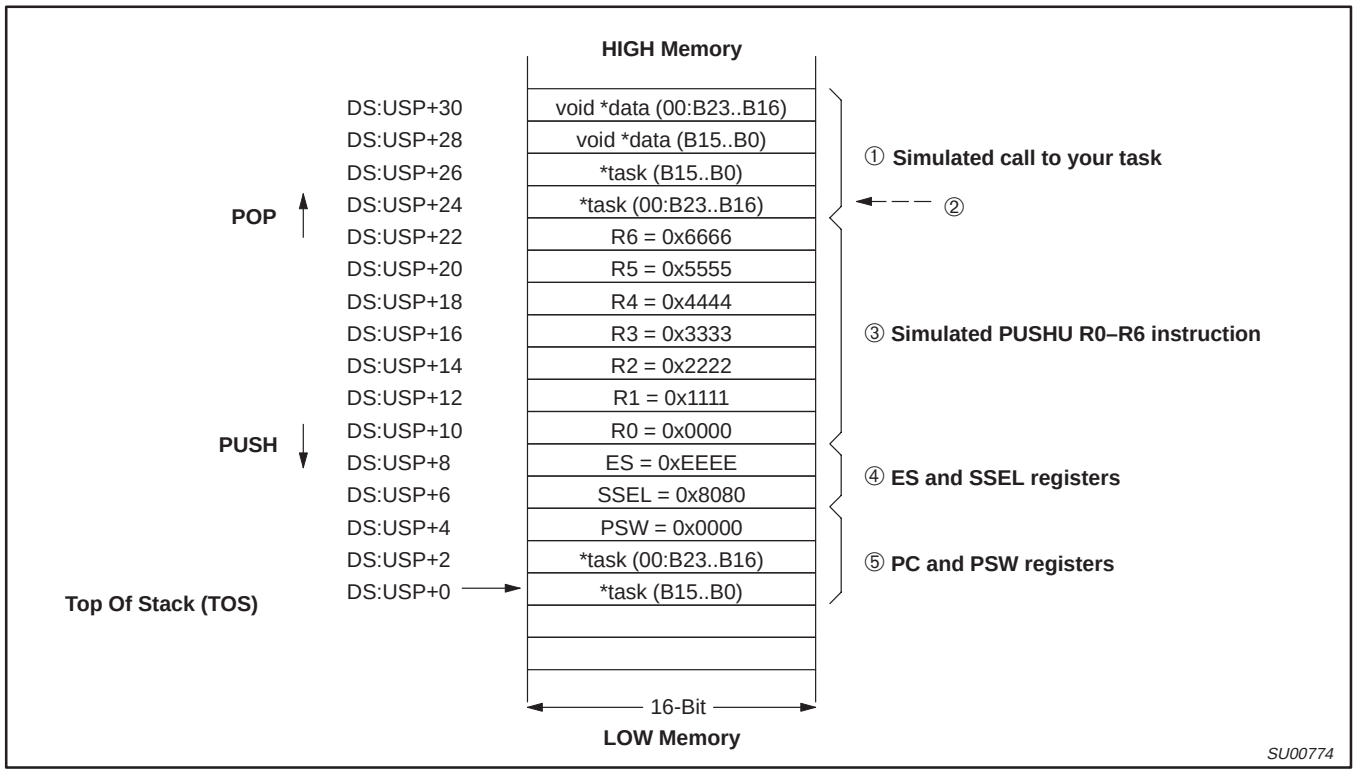
HIGH Memory
DS:USP+30
DS:USP+28
DS:USP+26
DS:USP+24
DS:USP+22
DS:USP+20
DS:USP+18
DS:USP+16
DS:USP+14
DS:USP+12
DS:USP+10
DS:USP+8
DS:USP+6
DS:USP+4
DS:USP+2
DS:USP+0
void *data (00:B23..B16)
void *data (B15..B0)
*task (B15..B0)
*task (00:B23..B16)
R6 = 0x6666
R5 = 0x5555
R4 = 0x4444
R3 = 0x3333
R2 = 0x2222
R1 = 0x1111
R0 = 0x0000
ES = 0xEEEE
SSEL = 0x8080
PSW = 0x0000
*task (00:B23..B16)
*task (B15..B0)
POP
PUSH
Top Of Stack (TOS)
① Simulated call to your task
②
③ Simulated PUSHU R0–R6 instruction
④ ES and SSEL registers
⑤ PC and PSW registers
16-Bit
LOW Memory
SU00774

Figure 6.

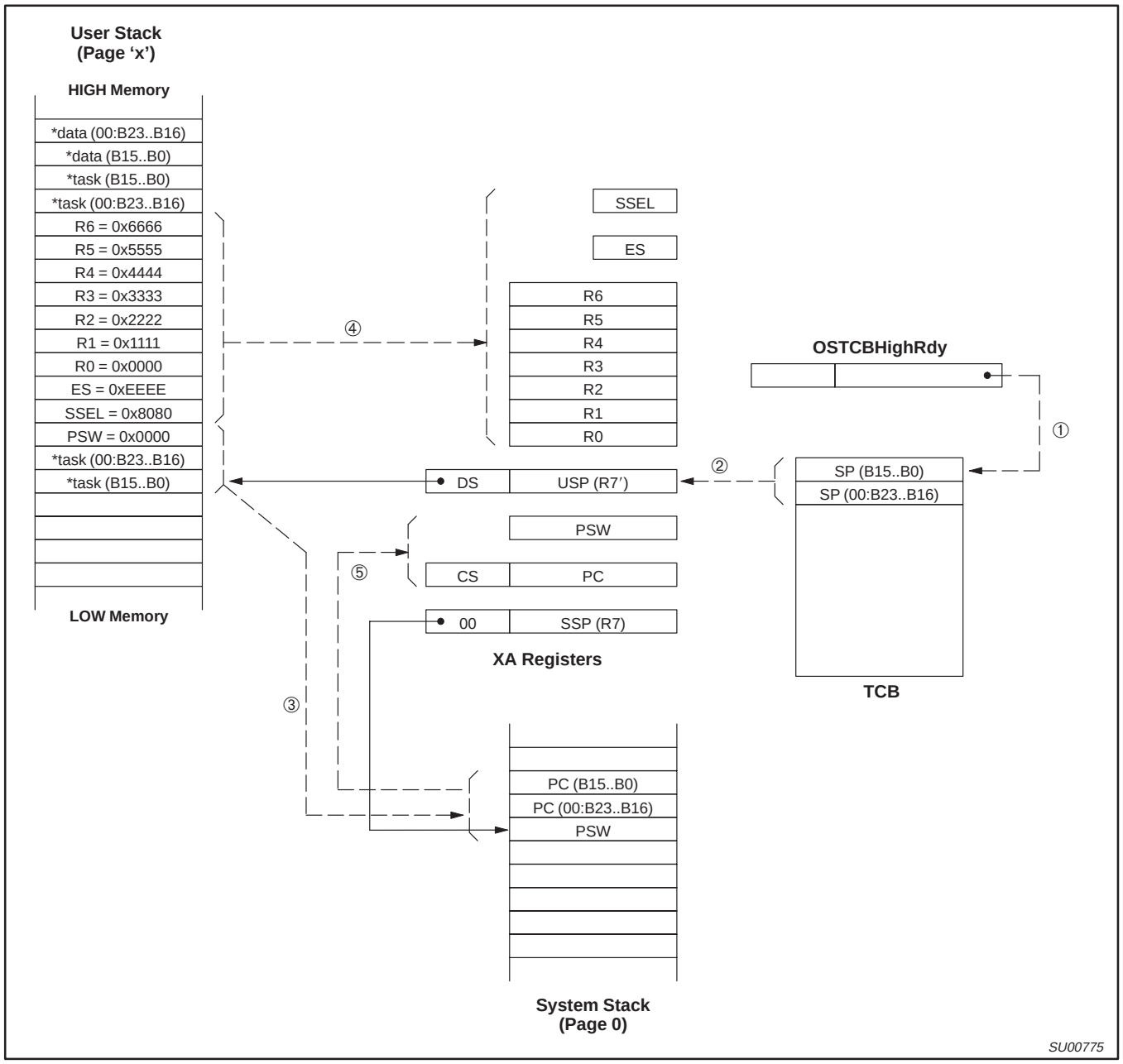
User Stack
(Page 'x')
HIGH Memory
*data (00:B23..B16)
*data (B15..B0)
*task (B15..B0)
*task (00:B23..B16)
R6 = 0x6666
R5 = 0x5555
R4 = 0x4444
R3 = 0x3333
R2 = 0x2222
R1 = 0x1111
R0 = 0x0000
ES = 0xEEEE
SSEL = 0x8080
PSW = 0x0000
*task (00:B23..B16)
*task (B15..B0)
LOW Memory
SSEL
ES
R6
R5
R4
R3
R2
R1
R0
④
OSTCBHighRdy
①
②
SP (B15..B0)
SP (00:B23..B16)
DS
USP (R7')
PSW
⑤
CS
PC
00
SSP (R7)
XA Registers
TCB
③
PC (B15..B0)
PC (00:B23..B16)
PSW
System Stack
(Page 0)
SU00775

Figure 7.

CONTEXT SWITCHING WITH µC/OS

Because µC/OS 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 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 8 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. `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.

The work done by `OSCtxSw()` is illustrated in Figure 9. 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 7, a context switch for the large model takes only about 10µs at 24MHz.

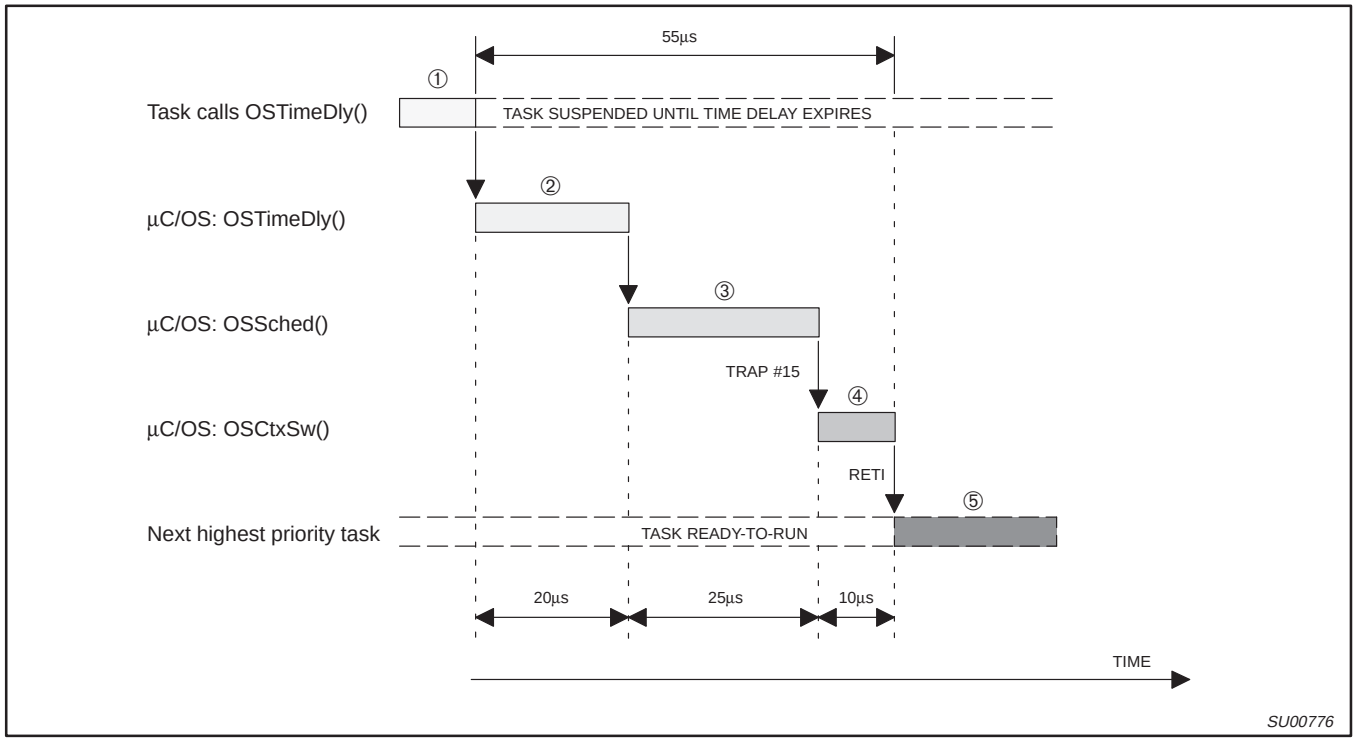

Figure 8.

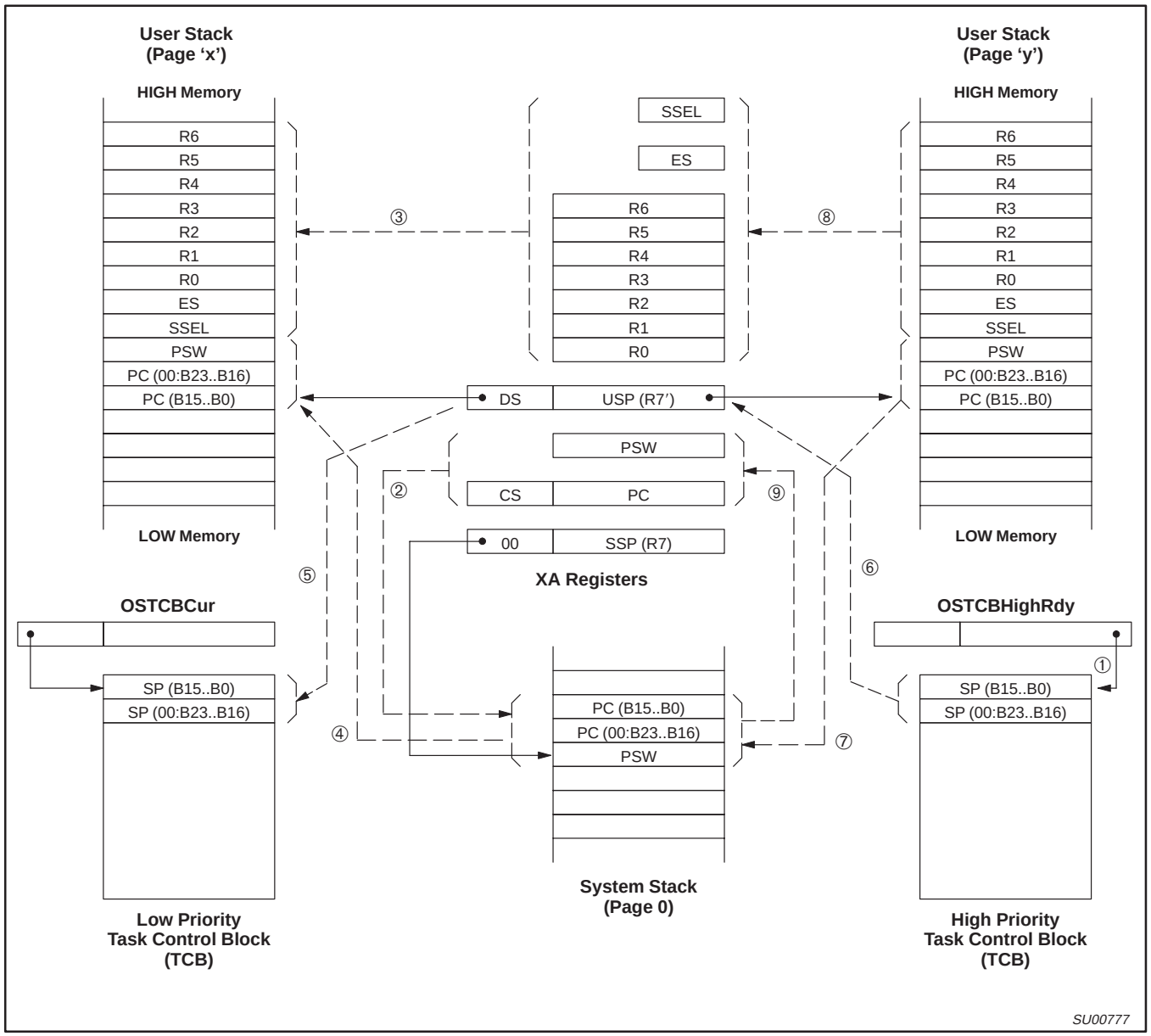
User Stack
(Page 'x')
HIGH Memory
R6
R5
R4
R3
R2
R1
R0
ES
SSEL
PSW
PC (00:B23..B16)
PC (B15..B0)
LOW Memory
User Stack
(Page 'y')
HIGH Memory
R6
R5
R4
R3
R2
R1
R0
ES
SSEL
PSW
PC (00:B23..B16)
PC (B15..B0)
LOW Memory
SSEL
ES
R6
R5
R4
R3
R2
R1
R0
DS
USP (R7')
PSW
CS
PC
00
SSP (R7)
XA Registers
①
②
③
④
⑤
⑥
⑦
⑧
⑨
OSTCBCur
SP (B15..B0)
SP (00:B23..B16)
OSTCBHighRdy
SP (B15..B0)
SP (00:B23..B16)
PC (B15..B0)
PC (00:B23..B16)
PSW
System Stack
(Page 0)
Low Priority
Task Control Block
(TCB)
High Priority
Task Control Block
(TCB)
SU00777

Figure 9.

INTERRUPT SERVICE ROUTINES (ISRS) AND µC/OS

Under µC/OS, you must write your ISRs in assembly language as shown in Figure 10. This code assumes you are using the HI–TECH C compiler and assembler. You must always save all the registers at the beginning of the ISR and restore them at the completion of the ISR. You must also always notify µC/OS when you are starting to process an ISR by calling `OSIntEnter()`. `OSIntEnter()` simply increments an interrupt nesting counter and thus takes very little time to execute. You can either process the interrupting device direcly in assembly language or, call a C handler as shown in Figure 9. Note that you are responsible for clearing the interrupt. 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 will need to perform a context switch to return to the more important task.

The stack frames (system and user) during an interrupt is shown in Figure 11. 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. `OSIntExit()` creates a local variable on the stack by saving R4. 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 11, 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.

```
        signat      _YourISR,24
        global      _OSIntEnter
        global      _YourISRHandler
        global      _OSIntExit
_YourISR:
        pushu       r0,r1,r2,r3,r4,r5,r6  ; Save R0 through R6
        pushu.b     442h                  ; Save ES
        pushu.b     403h                  ; Save SSEL
        pop         r2                    ; Move PSW to user stack
        pushu       r2
        pop         r2                    ; Move PCH to user stack
        pushu       r2
        pop         r2                    ; Move PCL to user stack
        pushu       r2
;
        fcall       _OSIntEnter           ; Notify uC/OS of ISR begin
        mov.b       401h,#80h             ; Allow interrupt nesting
        fcall       _YourISRHandler       ; Call your C ISR handler
        fcall       _OSIntExit            ; Notify uC/OS of ISR end
;
        popu        r2                    ; Move PCL to system stack
        push        r2
        popu        r2                    ; Move PCH to system stack
        push        r2
        popu        r2                    ; Move PSW to system stack
        push        r2
        popu.b      403h                  ; Restore SSEL
        popu.b      442h                  ; Restore ES
        popu        r0,r1,r2,r3,r4,r5,r6  ; Restore R0 through R6
;
        reti                              ; Return from Interrupt
```

SU00778

Figure 10.

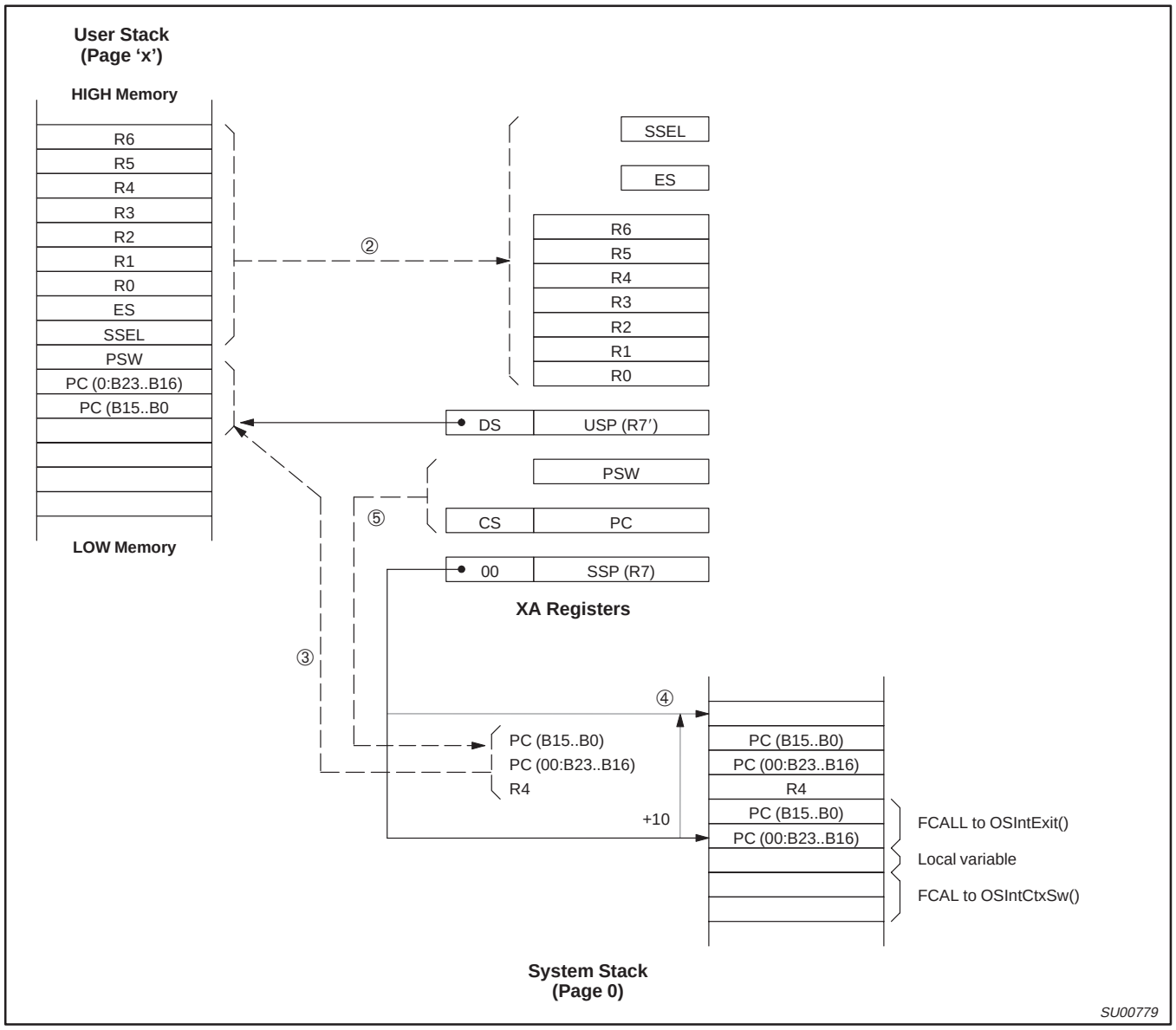
User Stack
(Page 'x')
HIGH Memory
R6
R5
R4
R3
R2
R1
R0
ES
SSEL
PSW
PC (0:B23..B16)
PC (B15..B0
LOW Memory
②
③
④
⑤
SSEL
ES
R6
R5
R4
R3
R2
R1
R0
DS
USP (R7')
PSW
CS
PC
00
SSP (R7)
XA Registers
PC (B15..B0)
PC (00:B23..B16)
R4
+10
PC (B15..B0)
PC (00:B23..B16)
R4
PC (B15..B0)
PC (00:B23..B16)
FCALL to OSIntExit()
Local variable
FCAL to OSIntCtxSw()
System Stack
(Page 0)
SU00779

Figure 11.

REFERENCES

[1] *µC/OS, The Real–Time Kernel*
Jean J. Labrosse
R&D Books, 1992
ISBN 0–87930–444–8

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

[3] *16–bit 80C51XA Microcontrollers (eXtended Architecture)*
Philips Semiconductors, Inc.
Data Book IC25, 1996

CONTACTS

HI–TECH Software
P.O. Box 103 Alderley
QLD 4051, Australia
+61 7 3300 5011
+61 7 3300 5246 (FAX)
WEB: http://www.hitech.com.au
e–mail: hitech@hitech.com.au

Jean J. Labrosse
9540 N.W. 9th Court
Plantation, FL 33324
954–472–5094
e–mail: 72644.3724@compuserve.com

Philips Semiconductors, Inc.
811 E. Arques Avenue
Sunnyvale, CA 94088–3409
(408) 991–2000
WEB: http://www.semiconductors.philips.com
FTP: ftp://ftp.ibsystems.com/pub/philips–mcu/bbs/xa/rtos
BBS: 800–451–6644 or 408–991–2406
or +31–40–2721102 (Europe)
9600 Baud, 8 bits, no–parity, 1 stop
Filename: UCOS_XA.EXE (MS–DOS self–extracting)
Instructions: From the root directory of the drive on which you want µC/OS, type: UC0S_XA

R&D Books, Inc.
1601 W. 23rd St., Suite 200
Lawrence, KS 66046–9950
(913) 841–1631
(913) 841–2624 (FAX)
WEB: http://www.rdbooks.com
e–mail: rdorders@rdpub.com

NOTES

µC/OS for the Philips XA

AN711

DEFINITIONS

Data Sheet Identification	Product Status	Definition
Objective Specification	Formative or in Design	This data sheet contains the design target or goal specifications for product development. Specifications may change in any manner without notice.
Preliminary Specification	Preproduction Product	This data sheet contains preliminary data, and supplementary data will be published at a later date. Philips Semiconductors reserves the right to make changes at any time without notice in order to improve design and supply the best possible product.
Product Specification	Full Production	This data sheet contains Final Specifications. Philips Semiconductors reserves the right to make changes at any time without notice, in order to improve design and supply the best possible product.

Philips Semiconductors
811 East Arques Avenue
P.O. Box 3409
Sunnyvale, California 94088–3409
Telephone 800-234-7381

457041/5M/FP/0996/pp16 Date of release: 09-96

Document order number: 9397 750 01178