Designing an Embedded Operating System with the TMS320 Family of DSPs

APPLICATION BRIEF: SPRA296

Astro Wu
DSP Applications – TI Asia

Digital Signal Processing Solutions
January 1998

TEXAS INSTRUMENTS

IMPORTANT NOTICE

Texas Instruments (TI) reserves the right to make changes to its products or to discontinue any semiconductor product or service without notice, and advises its customers to obtain the latest version of relevant information to verify, before placing orders, that the information being relied on is current.

TI warrants performance of its semiconductor products and related software to the specifications applicable at the time of sale in accordance with TI's standard warranty. Testing and other quality control techniques are utilized to the extent TI deems necessary to support this warranty. Specific testing of all parameters of each device is not necessarily performed, except those mandated by government requirements.

Certain application using semiconductor products may involve potential risks of death, personal injury, or severe property or environmental damage ("Critical Applications").

TI SEMICONDUCTOR PRODUCTS ARE NOT DESIGNED, INTENDED, AUTHORIZED, OR WARRANTED TO BE SUITABLE FOR USE IN LIFE-SUPPORT APPLICATIONS, DEVICES OR SYSTEMS OR OTHER CRITICAL APPLICATIONS.

Inclusion of TI products in such applications is understood to be fully at the risk of the customer. Use of TI products in such applications requires the written approval of an appropriate TI officer. Questions concerning potential risk applications should be directed to TI through a local SC sales office.

In order to minimize risks associated with the customer's applications, adequate design and operating safeguards should be provided by the customer to minimize inherent or procedural hazards.

TI assumes no liability for applications assistance, customer product design, software performance, or infringement of patents or services described herein. Nor does TI warrant or represent that any license, either express or implied, is granted under any patent right, copyright, mask work right, or other intellectual property right of TI covering or relating to any combination, machine, or process in which such semiconductor products or services might be or are used.

TRADEMARKS

TI is a trademark of Texas Instruments Incorporated.

Other brands and names are the property of their respective owners.

CONTACT INFORMATION

US TMS320 HOTLINE	(281) 274-2320
US TMS320 FAX	(281) 274-2324
US TMS320 BBS	(281) 274-2323
US TMS320 email	dsph@ti.com

Contents

Figures

Examples

Designing an Embedded Operating System with the TMS320 Family of DSPs

Abstract

Application software targeted for today's digital signal processors (DSP) is becoming more complex. DSPs are now incorporated with numerically intensive algorithms and must perform complex system control and communication protocols previously relegated to general-purpose microprocessors. When a complicated control is mixed with DSP software, the problem arises of how to implement a real-time kernel.

The Texas Instruments (TI™) TMS320 family of DSPs has evolved over the years from a simple attached numbers cruncher to a system on a chip. Sophisticated telecommunication systems are developed with TI DSPs such as the TMS320C5x, TMS320C54x, and TMS320C6x, which have MIPS greater than 50. As a result, more and more engineers face the problem of combining a previously implemented microcontroller base and their DSP code to implement a real-time operating system (OS).

This application note previews some of the problems facing designers of real time operating systems, and discusses how future applications with real time kernels will be implemented with high-speed DSP chips such as the Texas Instruments TMS320 series.

Product Support on the World Wide Web

Our World Wide Web site at www.ti.com contains the most up to date product information, revisions, and additions. Users registering with TI&ME can build custom information pages and receive new product updates automatically via email.

Performance Consideration

A number of factors affect a real-time kernel's performance:

- Task scheduler
- Interrupt response
- Determinism

Task Scheduler

The task scheduler is invoked before the real-time kernel completes service to an event from an application or when the interrupt service is completed. Therefore, the task scheduler could be the most frequently activated procedure in the activities of the kernel operations. It includes two timing factors:

- Scheduling
- Context switching

These factors must be considered when reducing the time consumed by the task scheduler.

Scheduling

In a preemptive kernel such as used in our design, scheduling decides the highest priority task to run. Context scheduling is performed whenever the highest priority task in ready has higher priority than the current running task. Traditionally, the Ready List is designed as a Linked List (single or multiple) data structure that maintains a list of Task Control Block (TCB) pointers. This means we must search the list for the pointer of the highest priority TCB. The following method uses the ReadyTable to determine the highest priority TCB.

A TCB enters the Virtual Ready List by mapping its priority to the ReadyGroup and ReadyTable[] base on the functions below which can support 8x8 tasks (in C code):

```
ReadyGroup  |= 8 >> (pTCB->Priority >> 3);
ReadyTable  |= 8 >> (pTCB->Priority & 0x07);
```

For design flexibility, we can rewrite the above code as:

```
#define MAX_NTASK       64                .
#define COL_MASK        7
#define MAX_COL_READY   8
#define MAX_ROW_READY   8
#define MAX_NCOL           3
ReadyGroup  |= MAX_ROW_READY >> (pTCB->Priority >> MAX_NCOL);
ReadyTable   |= MAX_COL_READY >> (pTCB->Priority & COL_MASK);
```

When fetching the highest priority pointer of the TCB (TCBHighReady) in the Ready List (TCBPriTbl[]), we use the functions below (in C code) :

```
P              = HighPriTCBIndexTbl[ReadyGroup];
TCBHighRdy     = TCBPriTbl[(P << 3) +
                 HighPriTCBIndexTbl[ReadyTbl[P]]];
```

HighPriTCBIndexTbl[] is generated into the program according to the following rules. For simplicity we use the maximum tasks 8x8 case:

HighPriTCBIndexTbl[N] = P, where N is the table index and P is its container.

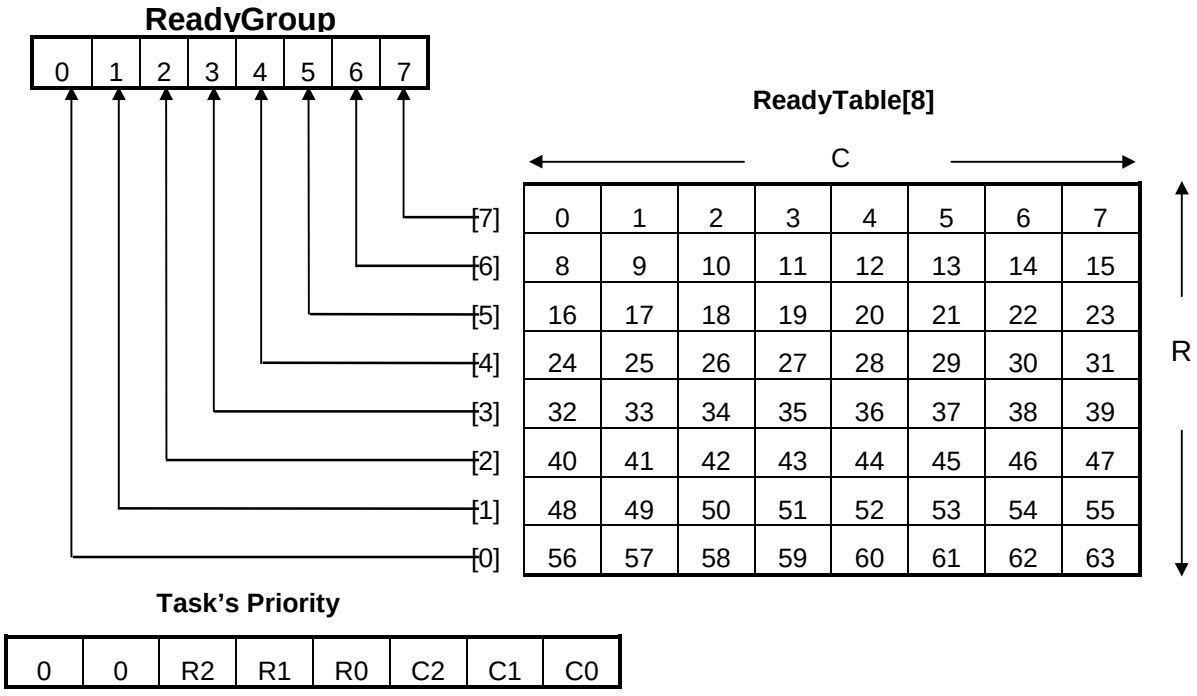
N = 2^K(2L+1) and P = 7-K

where

K= 0,1,…,8
L is integer ≥ 0
N = 0,1,…,255

The above formula is obtained by transforming the bit flag of ReadyGroup as well as ReadyTable[]. For example, the bit flag of ReadyGroup 11101110 will be 10000000 since only one bit (bit 7) denotes the highest priority group.

Figure 1. Mapping the Structure of the Ready List



Where C0-2 denotes the column index of bit position in ReadyTable[8] and R0-2 denotes the row index of bit position in ReadyGroup.

Note: The numbers in the table denote lowest to highest priority from 0 to 63. Each cell contains a bit I/0 to indicate setting/resetting for Task with a specific priority. (This is just an example of the kernel supporting 64 tasks.)

For portability and maintainability of the kernel system, most code is written in C except for those procedures that are time critical to the kernel's performance. Actually, we can write all of the codes in C except for the Context Switch codes written in Assembly. Therefore, the code's effort is reduced when utilizing the codes from C5x for other DSPs, such as C6x. With support from the Assembler tools, we can also rewrite Scheduler code in TMS320 Assembly without losing much of the maintainability (see Example 1).

Example 1. Task Scheduler Codes Written in C and C5x Assembly

```
;void Scheduler(void)
;{
;    UBYTE P;


;    EnterCritical();
;    if ((LockNesting | IntNesting) == 0)
;    {
;        /*
;         * Task scheduling must be enabled and not ISR level
;         */
;        P  = HighPriTCBIndexTbl[ReadyGroup];    /* Get PTR to highest pri task
ready to run  */
;        TCBHighRdy  = TCBPriTbl[(P << 3) + HighPriTCBIndexTbl[ReadyTbl[P]]];
;        if (TCBHighRdy->Priority > TCBCurrent->Priority)
;        {
;            /*
;             * Make sure this is not the current task running
;             */
;            OSCtxSwCtr++;                       /* Increment context switch counter
*/
;            ContextSwitch();                    /* Perform a context switch  */
;        }
;    }
;    ExitCritical();
;} Scheduler
        EnterCritical
        lar         AR4,#os_mem_start+OS_CONTROL.IntNesting
        lacl        *
        lar         AR4,#os_mem_start+OS_CONTROL.LockNesting
        or          *
        bcnd        ExitScheduler,NEQ
        lacl        #HighPriTCBIndexTbl
```

```
        samm    indx
        lar     AR4,#os_mem_start+OS_CONTROL.ReadyGroup
        lacl    *0+
        samm    indx            ; indx= y =HighPriTCBIndexTbl[ReadyGroup]
        sacb                    ; accb=y
        lar     ar4,#os_mem_start+OS_CONTROL.ReadyTbl
        lacl    *0+             ; acc=z= ReadyGroup[y]
        samm    indx            ; indx= x
        rpt     #MAX_NCOL-1
        sflb                    ; accb= y << MAX_NCOL
        lar     ar4,#HighPriTCBIndexTbl
        lacl    *0+             ; acc=z= HighPriTCBIndexTbl[x]
        addb                    ; acc=p= y<<MAX_NCOL+ z
        samm    indx            ; indx=p
        lar     AR4,#os_mem_start+OS_CONTROL.TCBPriTbl
        lacl    *0+
        lar     AR4,#os_mem_start+OS_CONTROL.TCBHighReady
        sacl    *               ; TCBHighReady= TCBPriTbl[p]
        add     #OS_TCB.Priority
        samm    AR4
        lacl    *               ; acc= TCBPriTbl->Priority
        lar     AR4,#os_mem_start+OS_CONTROL.TCBCurrent
        lar     AR4,*
adrk    AR4,#OS_TCB.Priority
        clrc    sxm
        sub     *               ; Q: TCBHighReady->Priority <=
TCBCurrent->Priority
        bcnd    ExitSchedular,LEQ ; Y: Don't do task switching
        lar     AR4,#os_mem_start+OS_CONTROL.StatusBits
        opl     SCHED_RUNNING_BIT,*
        lar     AR4,#os_mem_start+OS_CONTROL.CtxSwCtr
        lacl    *
        add     #1              ; CtxSwCtr++
```

```
        sacl    *
        ExitCritical
        b       ContextSwitch
ExitScheduler
        ExitCritical
        lacl    *                       ; Return task handle
        b       KernelService_Exit
```

Context Switching

Context switching is invoked for:

- Non-interrupt mode when the kernel has completed service to the event coming from a task
- Interrupt mode when the interrupt service routine is completed.

Because interrupt can happen anywhere in the code, we need to save/restore the whole context. In the non-interrupt mode, context needs to be saved/restored only when the task calls the operating system kernel service. Moreover, context switching is not allowed during nested interrupts because interrupt must have a higher priority than any task.

Because DSPs such as the C5x, C54x, and C6x provide many auxiliary registers, and the C compiler protects some registers before calling another subroutine in C or Assembly (see Appendix A), we do not need to save context for those registers when switching tasks in the non-interrupt mode.

Example 2. Contexts for Interrupt Mode (STACK_FRAME plus INT_SAVE) and Non-interrupt Mode (STACK_FRAME) in C5x Assembly

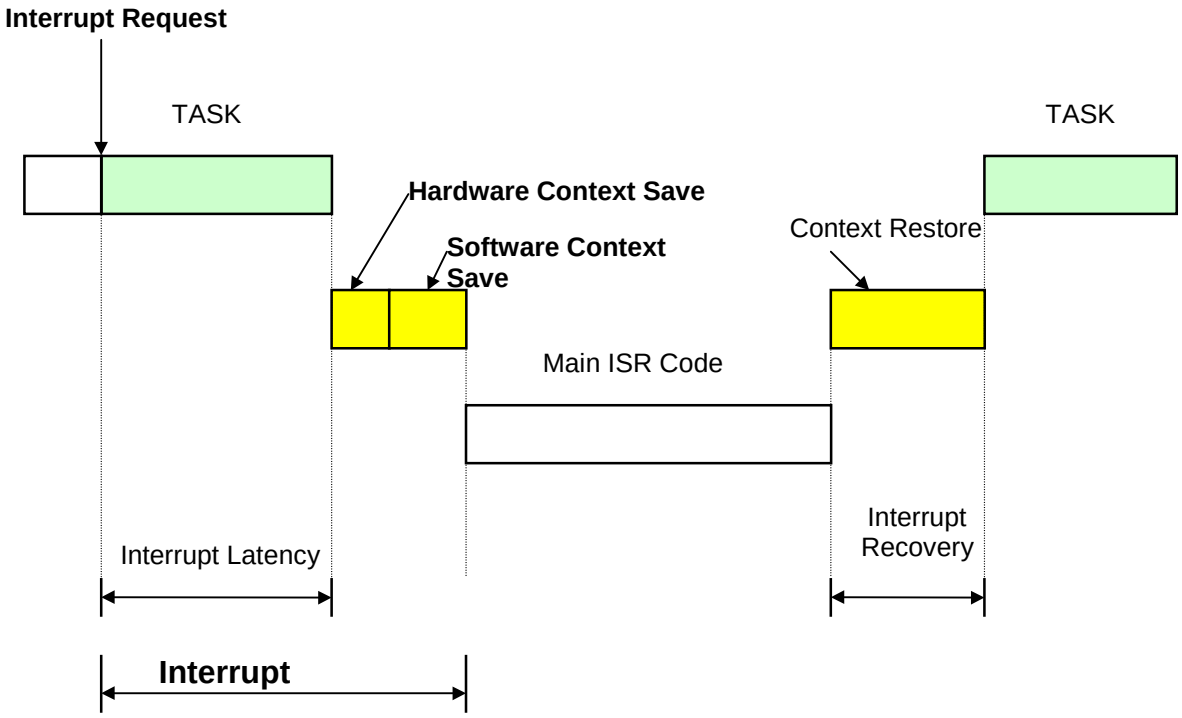
```
STACK_FRAME             .struct  ; Task Control Block
ST1             .word            ; ST0 register
ST0             .word            ; ST1 register
PMST            .word            ; PMST register
AR0             .word            ; AR0 register
AR1             .word            ; AR1 register
AR6             .word            ; AR6 register
AR7             .word            ; AR7 register
Page            .word            ; Current code Page
Return          .word            ; Hardware stack level 1 for C5x only
HardStack       .space  6*16     ; Hardware stack level 2-7 for C5x only
Stack8          .word            ; Hardware stack level 8 for C5x only
STACK_FRAME_LEN                  .endstruct

INT_SAVE        .struct          ; OS interrupt save area
ST1             .word            ; Interrupted code's ST1 register
ST0             .word            ; Interrupted code's ST0 register
Return          .word            ; Interrupted code's Return address
StackPtr        .word            ; Interrupted code's stack pointer
AR2             .word            ; Interrupted code's AR2 register
AR3             .word            ; Interrupted code's AR3 register
AR4             .word            ; Interrupted code's AR4 register
AR5             .word            ; Interrupted code's AR5 register
ACCL            .word            ; Interrupted code's accumulator
ACCH            .word            ; Interrupted code's accumulator
ACCBL           .word            ; Interrupted code's accumulator B
ACCBH           .word            ; Interrupted code's accumulator B
PMST            .word            ; Interrupted code's PMST register
RPTC            .word            ; Interrupted code's RPTC register
BRCR            .word            ; Interrupted code's BRCR register
PASR            .word            ; Interrupted code's PASR register
PAER            .word            ; Interrupted code's PAER register
TREG2           .word            ; Interrupted code's TREG2 register
TREG1           .word            ; Interrupted code's TREG1 register
TREG0           .word            ; Interrupted code's TREG0 register
PREGH           .word            ; Interrupted code's product register
PREGL           .word            ; Interrupted code's product register
DBMR            .word            ; Interrupted code's DBMR register
INDX            .word            ; Interrupted code's INDX register
ARCR            .word            ; Interrupted code's ARCR register
BMAR            .word            ; Interrupted code's BMAR register
INT_SAVE_LEN    .endstruct
```

Note: The ‘Page’ in STACK_FRAME is used for extended memory addressing. In C54x ‘page’ will be replaced with XPC.

Interrupt Response

The interrupt response time includes Interrupt Latency, Hardware Context Save, and Software Context Save. Improvements we can make are to Interrupt Latency by shortening the time in the critical section, and Software Context Save which is written in optimized Assembly code. We must disable all interrupts before entering the critical section and then enable the interrupts again after leaving the section.

Figure 2. Interrupt Latency, Response, and Recovery Time



Note: The response time is one factor in operating system performance.

Determinism

Kernel service should be deterministic by specifying how long each service call will take to execute. Some kernel services (such as Semaphore Pend, Semaphore Post, Mailbox Pend, and Mailbox Post) are frequently invoked. Thus, more attention should be given to these services when doing the optimization for shortening the average of the kernel service time.

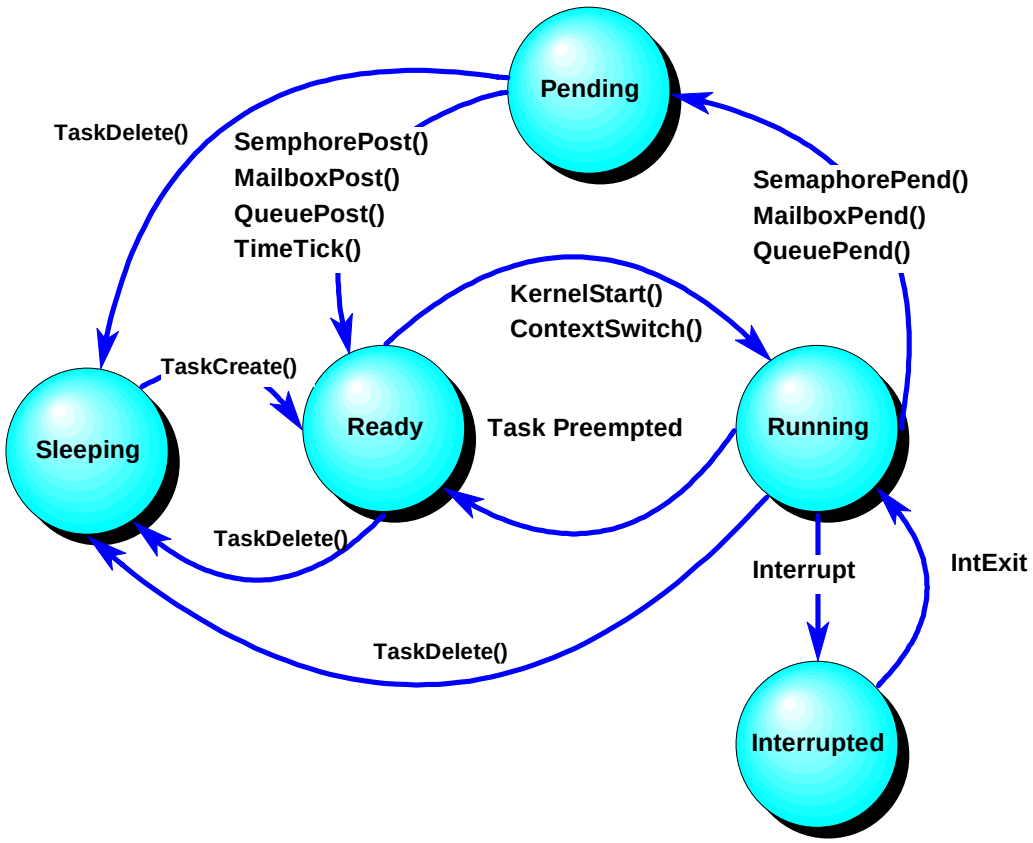
Task State Transition

The task can be designed in five states.

- Sleeping (Dormant)
- Ready
- Pending
- Running
- Interrupted

Figure 3 shows the task state transition diagram. (We can remove the Sleeping State if we don't need to delete a task when the OS kernel is running.)

Figure 3. Task State Transition Diagram



Event State Transition

When an application calls for kernel service, it sends an event parameter that arouses the OS kernel to perform appropriate processes. To more clearly understand our design for event processing of the OS kernel, we use the event state transition diagrams shown in Figure 4 through Figure 6 for the following events:

- TASK_CREATE
- SEMAPHORE_PEND
- SEMAPHORE_POST

Figure 4. Kernel Service State Diagram for TASK_CREATE

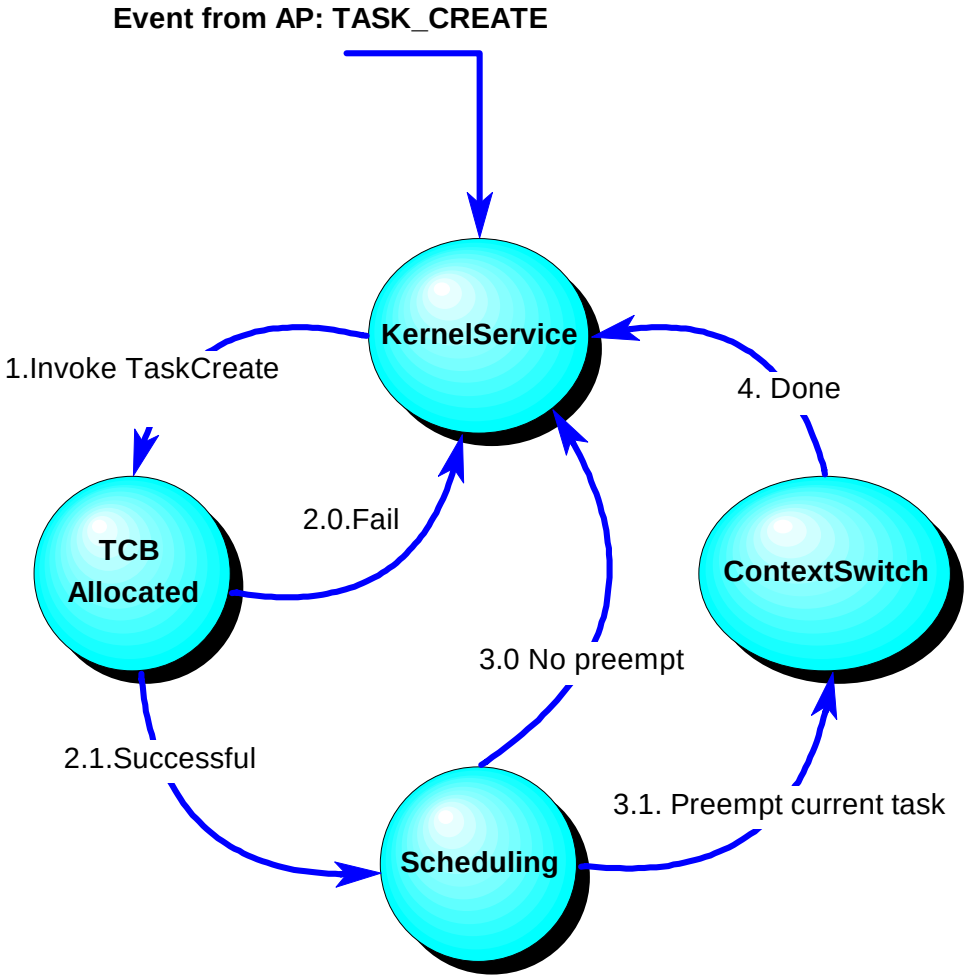


Figure 5. Kernel Service State Diagram for SEMAPHORE_PEND

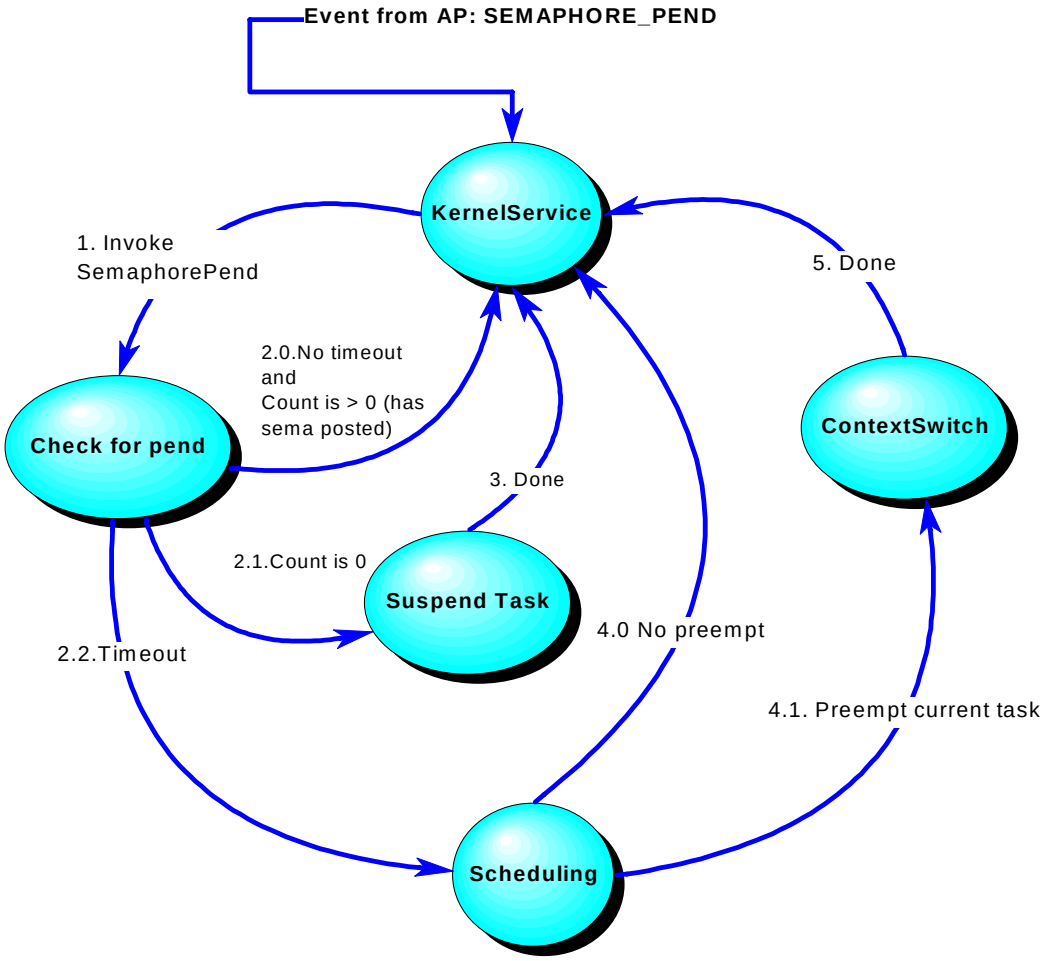
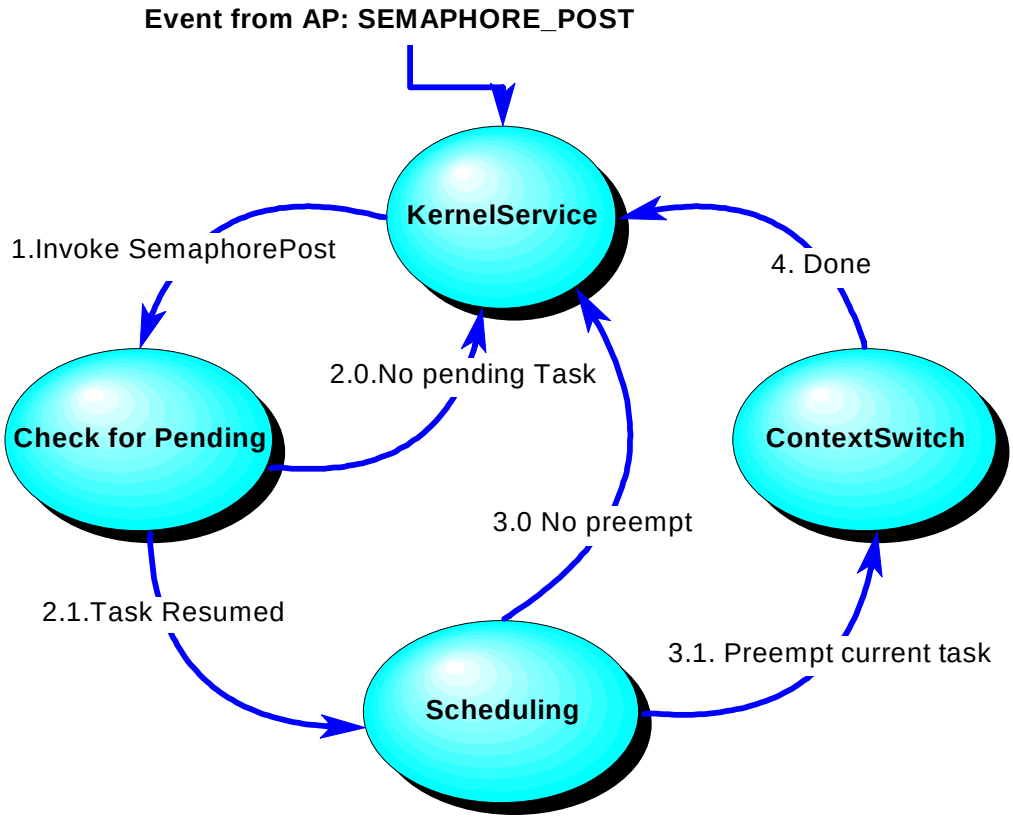


Figure 6. Kernel State Diagram for SEMAPHORE_POST



Timer Resource Consideration

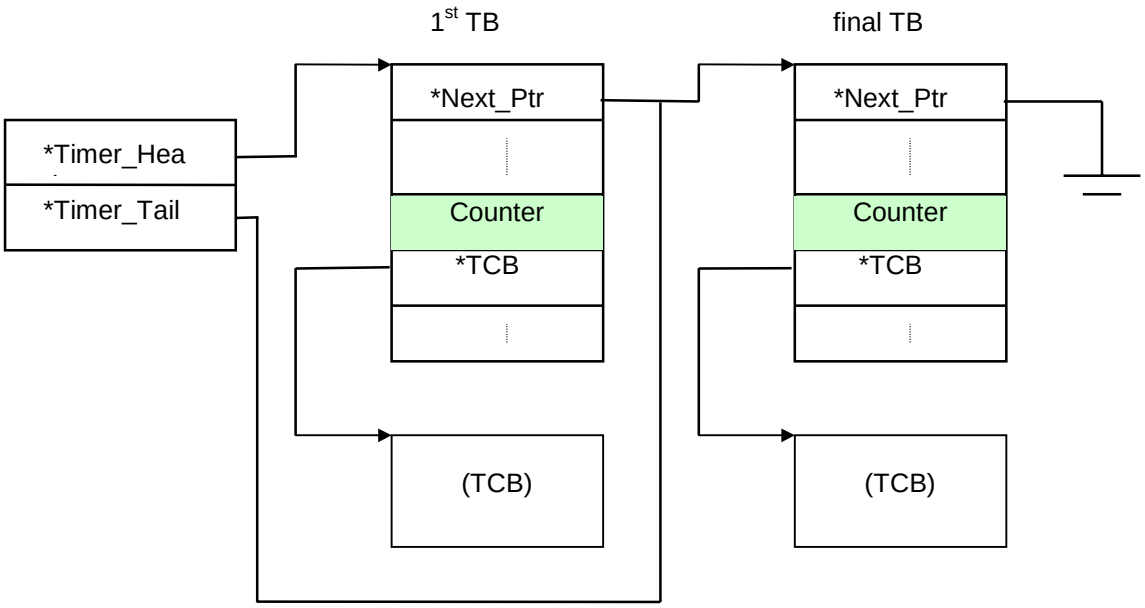
A multiple timer resource can be selected normally. The timer ticks can be internally selected from the DSP itself or from an external device such as the analog interface circuit (AIC). For example, if we need only one mini-second tick unit, we may not need the internal timer interrupt, which could frequently interrupt our tasks. Instead, we can use the AIC as an interrupt source and assign a counter inside the AIC interrupt service routine to count up to:

Sampling Rate (Hz) / Time of Tick (sec)

Timer Structure

The timer structure can be designed as a Linked List Timer Block (see Figure 7).

Figure 7. Data Structure for Timer Control List



If we design the counter in Timer Block (TB) as ticks for time expired, we should discount all TBs in the running list by one. When one tick unit is timed out and there is a new TB, always insert behind the tail. Instead of searching the whole list at that time, we modify both the TB insertion and the counter value assigned rules as follows:

- (Rule 1) For any Timer Block TBn in the Running List, its counter is TBn.Counter and:
- (Rule 2) Its time-to-expired of TBn = $\sum_{k=0}^{n} \mathrm{TBk.Counter}$

  where k=0 is 1st TB in List , etc.

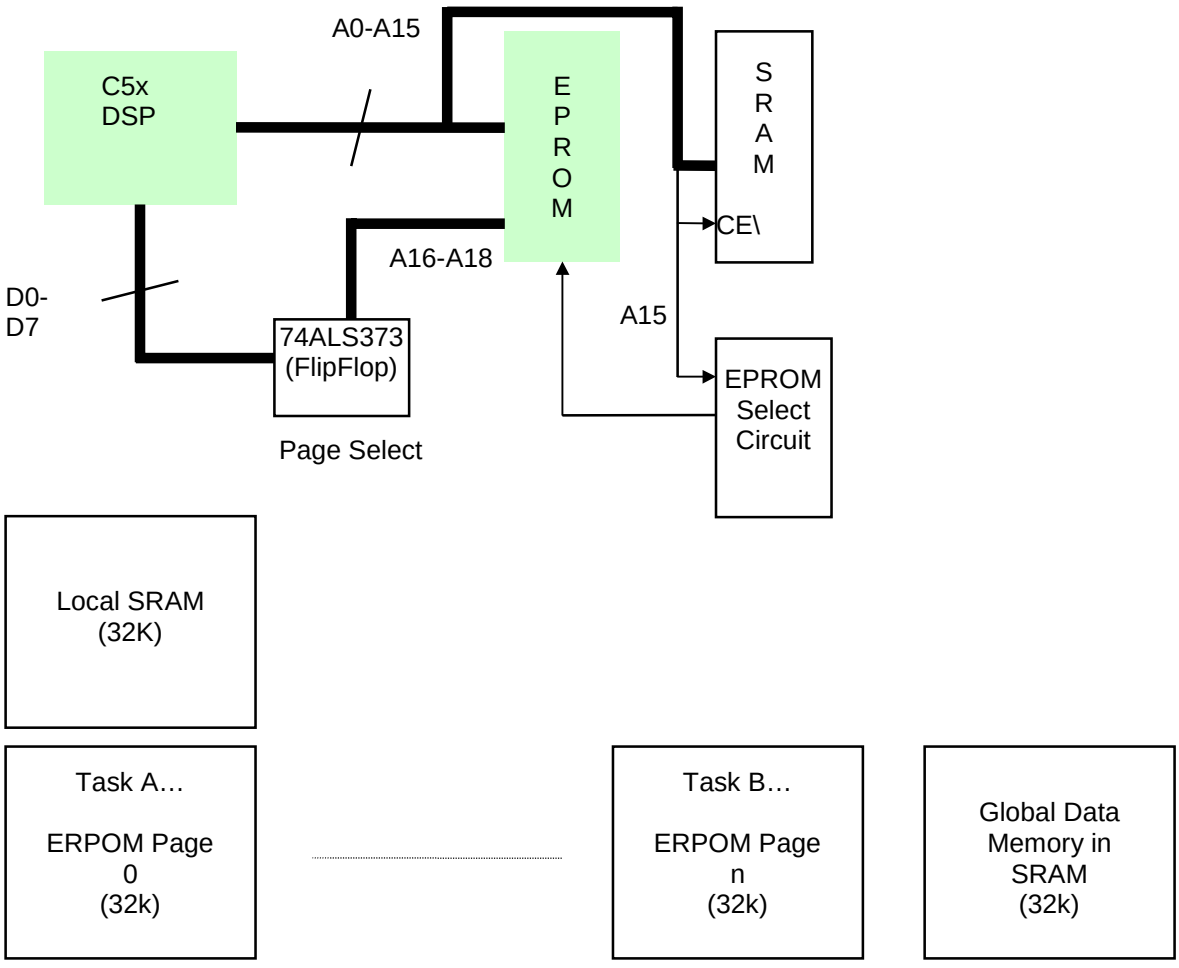
Based on rule 1, those TBs should be inserted in the list in an ascending order for its counter.

So, when one tick unit is timed out, we only count down the 'TB0.Counter' by one and if it is zero, we invoke the TCB for task switching.

EPROM Addressing Considerations

Because code in SRAM runs much faster than code in EPROM, we usually copy the time critical codes (such as real-time signal processing and real-time kernel) to the SRAM. But for the embedded system and also for cost-effective considerations, we put most codes in EPROM and copy the critical portion to Local SRAM during run-time. The C5x DSP has limited addressing ability to 64k word, so code words over 64k when put into EPROM need a circuit for page selection, as shown in Figure 8.

Figure 8. Circuit and Memory Layout for C5x Extended Memory Addressing



Note: The C54x uses the XPC register for page selection.

This function has been added to expand the memory addressing of the C5x. Because the EPROM address is added internally to the C54x and C6x, we can make page selections by writing to the register directly.

The page selection information is part of the context and should be stored in the stack of TCBs (see Example 2). Context will be restored to the task located at a specific page of EPROM (see Figure 8). No single task can be addressed to more than one page space.

Application Interface Consideration

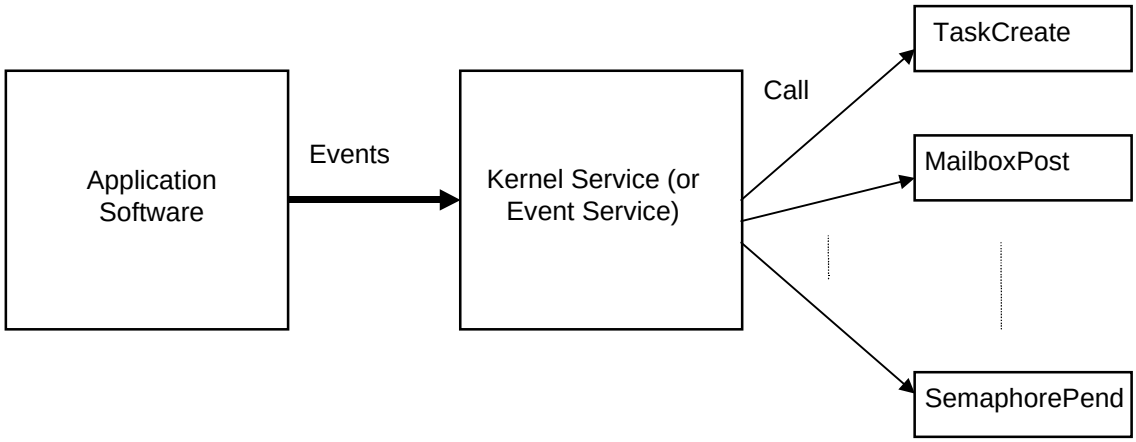
The application calls the kernel by feeding the proper event relevant information to the kernel service. The following factors should be addressed when designing the interface between the application software and the operating system kernel:

- Ease of maintenance
- Debugging user-friendliness
- Portability
- Performance

Single Entry Point

A single entry point API (application program interface) is better than multiple access points. It sets relevant service procedures public to the application on issues of maintenance, debugging, and user-friendliness just by increasing a bit of overhead.

Figure 9. Kernel API Block Diagram



When Application is written in C

For applications written in C, we must provide a C procedure interface between the user's application and the OS kernel. This section explains two methods that apply mainly to the C5x. It is assumed that only a small effort is required for future portings to the C54x and C6x.

Case I- Kernel Service as C Code

This method shown in Example 3 is based on the single entry point API explained previously (see the section, *Single Entry Point*). We use as little of the inline assembly code as possible. The local variable declared in the procedure *TSK_create* will be treated as an argument to the software interrupt vector called *KernelService*.

Example 3. Kernel Service as C Code (for Applications in C)

```
void TSK_create (int pTask, int Priority)
{
  Stack_Frame p;
  p.ST0 = ST0;
  p.ST1 = ST1;
  p.PMST = PMST;
  p.KSEventID = CREATE_TASK;
  p.Arg2 = pTask;
  p.Arg3 = Priority;
  KERNEL_SERVICE;
}
#define KS_VECTOR          10
#define  KERNEL_SERVICE \
       asm("  intr   KS_VECTOR"); \
       asm(" ret ");
int  KernelService (Stack_Frame  *pStackFrame)
{
  TCBCurrent.StackPointer = pStackFrame;
```

```
    if (pStackFrame->KSEventID > MAX_KS_EVENTS)
        return SYS_BAD_EVENT;
    return  (*EventTable[pStackFrame->KSEventID])(pStackFrame)
;
}
int (* EventTable[]) (Stack_Frame *) =
{
        CreateTask,
        DeleteTask,
        CreateSemaphore,
        PostSemaphore,
        PendSemaphore,
        •••
};
```

Case II- Kernel Service as Assembly Code

In this case, we don't need to design the interface in C. Because registers such as AR2 to AR5 were protected in the caller's local frame by the C compiler when they entered this Assembly interface, we don't need to save those contexts to TCB's stack frame (see the Appendix).

Example 4. Kernel Service as Assembly Code (for Applications in C)

```
_TSK_create   .def  _TSK_create       ; TSK_create (pTask,  Priority)
* ar2-ar5 have been protected by caller in C
* we don't need local variables
* on entry ARP = 1
      sar       ar1,*                 ; save AR1 (SP) to Stack
      lar       ar2,*,ar2             ; AR2 = AR1
      sbrk       1
      lacl      *-                    ; get ARG1
      samm     AR3                    ; AR3 = pTask
```

```
        lacl    *                   ; get ARG2
        samm    AR4                 ; AR4= Priority
        lar     AR2, #CREATE_TASK
        call    KernelService
```

When the Application is Written in Assembly

Case I- Kernel Service is C Code

This case will not be discussed, since it's rarely happened.

Case II- Kernel Service is Assembly Code

In this case, we can design the interface as a Marco and use AR2 to AR5 and parameters for calling the kernel service. Since AR2 to AR5 are known to Application as Assembly in such usage, we can use them freely in Kernel without involving them in the context switching.

Example 5. Kernel Service as Assembly Code (for Applications in Assembly)

```
TSK_create      .macro  pTask,  Priority, StackFrame
        lar             AR2,#CREATE_TASK ; Event ID
        lar             AR3, pTask              ; new Task functional entry pointer
        lar             AR4, Priority           ; Priority of new Task
        lar             AR5, StackFrame         ; Stack Frame Pointer
        call            KernelService
       .endm ; TSK_create
KernelService
        lamm            AR2                     ; get EventID
        sub             #EventTableEnd-EventTable ; Q: Is EventID out of range ?
        bcnd            bad_event,GT            ; Y:  return  SYS_BAD_EVENT
        add             #EventTable             ;
        sacb                                    ; accb= event service function ptr
        EnterCritical
        mar             *,AR5                   ; AR5 : stack frame pointer
```

```
        sst             #0,*+                   ; push ST0 to software stack
        sst             #1,*+                   ; push ST1 to software stack
        ExitCritical
        lamm            PMST                    ; get PMST register
        sacl            *+                      ; push  PMST to software stack
        popd            *+                      ; push return address to software
stack
        lacb                                    ; restore event service function ptr
        tblr            temp
        lacl            temp                    ;
        bacc                                    ; Execute kernel service function
*****************************************************************************************************
                        Kernel Service Event Function Table
*****************************************************************************************************
EventTable
        .word   CreateTask
        .word   create_a_sem
        .word   pend_on_sem
        .word   PostSemaphore
        .word   create_a_mbx
        .word   pend_on_mbx
        ( others …)
EventTableEnd
```

Appendix A. Calling Convention for C5x and C54x

TMS320C5x

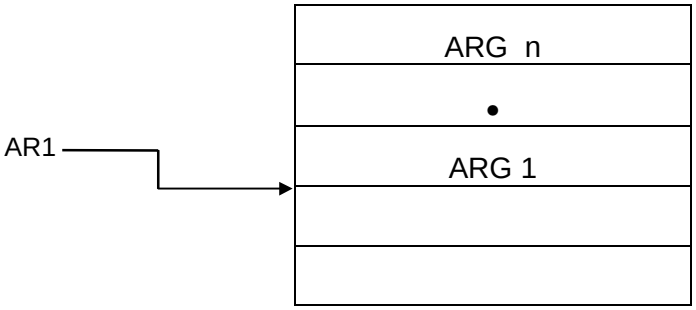
The C Compiler does **not** preserve the following registers over function boundaries.

Registers	Used by C
FP (AR0)	Long Frame Pointer
SP (AR1)	Stack Pointer
AR6,AR7	Register Variables

The C Compiler preserves the following registers over function boundaries.

Registers	Used by C
ACC	Accumulator
ACCB	Accumulator Buffer
P	Product Register
T	Temporary Register
AR2-AR5	Auxiliary Registers 2 to 5
PMST	Status Register
ST0, ST1	Status Register

Stack When Entering Subroutine



Note: The return address is in Hardware Stack.

TMS320C54x

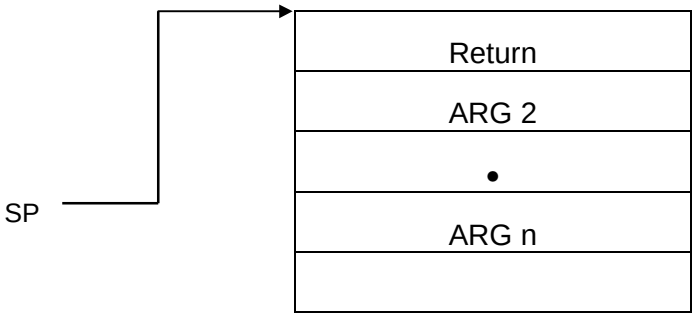
The C Compiler does **not** preserve the following registers over function boundaries.

Registers	Used by C
AR7	Long Frame Pointer
SP	Stack Pointer
AR1,AR6	Register Variables
A	1ST Argument or Return Address

The C Compiler preserves the following registers over function boundaries:

Registers	Used by C
B	Expression Analysis
T	Expression Analysis
AR0	Pointers and expressions
AR2-AR5	Expression Analysis
BRC	Loop registers (RSA, REA)

Stack When Entering Subroutine



Note: ARG1 is in ACC.

References

Jean J. Labrosse, *µC/OS The Real-Time Kernel*

James L. Perterson, *Operating System Concepts*, Addsion-Wesley Publishing Company

Andrew S. Tanenbaum, *Modern Operating System*, Prentice-Hall

TMS320C5x Assembly Language Tools, Literature number

TMS320C5x User's Guide, Literature number SPRU056C