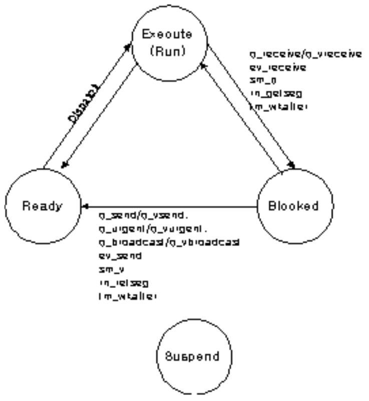
pSOS System Call

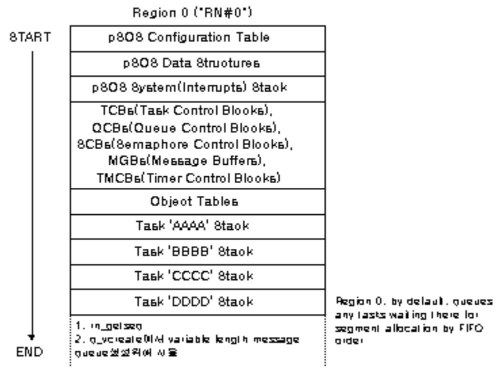
1. 紐� �ㅈㅐㅊ

pSOS Real Time Kernel � ㅂㅓㅊ�ㄴㅣ ㄹ瑜� �ㅈㅓㅌ由ㅂㅂㅓㄹ�ㄸㅐㄹ.

2. Task State Transitions



3. Region 0


Region 0 ("RN#0")
START
END
pSOS Configuration Table
pSOS Data Structures
pSOS System(Interrupts) Stack
TCBs(Task Control Blocks),
QCBs(Queue Control Blocks),
SCBs(Semaphore Control Blocks),
MGBs(Message Buffers),
TMCBs(Timer Control Blocks)
Object Tables
Task 'AAAA' Stack
Task 'BBBB' Stack
Task 'CCCC' Stack
Task 'DDDD' Stack
1. rn_getseg
2. q_vcreate에서 variable length message queue생성위해 사용
Region 0, by default, queues any tasks waiting there for segment allocation by FIFO order

4. pSOS+ Configuration

pSOSConfigTable

	void (*kc_psoscode)();	/* start address of pSOS+ */
	void *kc_rn0sadr;	/* region 0 start address */
	unsigned long kc_rn0len;	/* region 0 length */
	unsigned long kc_rn0usize;	/* region 0 unit size */
Object Count	unsigned long kc_ntask;	/* max number of tasks */
	unsigned long kc_nqueue;	/* max number of message queues */
	unsigned long kc_nsema4;	/* max number of semaphores */
	unsigned long kc_nmsgbuf;	/* max number of message buffers */
	unsigned long kc_ntimer;	/* max number of timers */
	unsigned long kc_nlocobj;	/* max number of local objects */
Clock Ticks	unsigned long kc_ticks2sec;	/* clock tick interrupt frequency */
	unsigned long kc_ticks2slice;	/* time slice quantum, in ticks */
I/O Devices	unsigned long kc_nio;	/* num of I/O devices in system */
	struct pSOS_IO_Jump_Table *kc_iojtable;	/* addr of I/O switch table */
	unsigned long kc_sysstk;	/* pSOS+ system stack size (bytes) */
Root Task	void (*kc_rootsadr)();	/* ROOT start address */
	unsigned long kc_rootsstk;	/* ROOT supervisor stack size */
	unsigned long kc_rootustk;	/* ROOT user stack size */

	unsigned long kc_rootmode;	/* ROOT initial mode */

Callouts	void (*kc_startco)();
	void (*kc_deleteco)();
	void (*kc_switchco)();
		/* callout at task activation */
		/* callout at task deletion */
		/* callout at task switch */

	void (*kc_fatal)();
	unsigned long kc_rootpri;
		/* fatal error handler address */
		/* ROOT task priority */

5. pSOS+ Real Time Kernel

5.1. Task Management

t_create Creates a task.

```
unsigned long t_create(
    char name[4],           /* task name */
    unsigned long prio,     /* task priority */
    unsigned long sstack,  /* task supervisor stack size */
    unsigned long ustack,  /* task user stack size */
    unsigned long flags,    /* task attributes */
    unsigned long *tid      /* task identifier */
)
```

� ㅃ ㅔ ㅀ sstack : t_create() internally calls rn_getseg() to allocate a segment from Region 0 to hold the task���stack and the user stack, if any.

� ㅃ ㅔ ㅀ Ustack : ustack may be 0 if the task executes only in supervisor mode

� ㅃ ㅔ ㅀ flags

T_GLOBAL /T_LOCAL

Makes the task global: external tasks on other nodes can address it / restricts the task to the local node.

The T_GLOBAL attribute is ignored by the single-processor kernel.

T_FPU / T_NOFPU

Informs the pSOS+ kernel that the task uses /does not use the FPU coprocessor

t_start Starts a task.

```
unsigned long t_start(
    unsigned long tid,       /* task identifier */
    unsigned long mode,   /* initial task attributes */
    void (*start_addr)(),    /* task address */
    unsigned long targs[4] /* startup task arguments */
)
```

� ㅃ ㅔ ㅀ mode

T_PREEMPT /T_NOPREEMPT : Task is / is not preemptible.

T_TSLICE /T_NOTSLICE : Task can /cannot be time-sliced.

T_ASR /T_NOASR : Task's ASR is enabled / disabled.

T_USER /T_SUPV : Task runs in user / supervisor mode.

T_ISR /T_NOISR : Hardware interrupts are enabled / disabled while task runs.

T_LEVELMASK0 through T_LEVELMASK n : Certain hardware interrupts are disabled while the task runs. These options are available only on certain processors.

t_restart Forces a task to start over regardless of its current state.

```
unsigned long t_restart(
    unsigned long tid, /* task identifier */
    unsigned long targs[4] /* startup arguments */
)
```

This system call forces a task to resume execution at its original start address regardless of its current state or place of execution. If the task was blocked, the pSOS+ kernel forcibly unblocks it. The task's priority and stacks are set to the original values that t_create() specified. Its start address and execution mode are reset to the original values established by t_start(). Any pending events, signals, or armed timers are cleared.

t_delete Deletes a task.

```
unsigned long t_delete(
   unsigned long tid /* task identifier */
)
```

<u>Task's Notepad Register</u>

t_setreg Sets a task's notepad register.

```
unsigned long t_setreg(
    unsigned long tid,    /* task identifier */
    unsigned long regnum,  /* register number */
    unsigned long reg_value /* register value */
)
```

� ㅃ ㅔ ㅀ regnum : Specifies the register number.

t_getreg Gets a task���'s notepad register.

```
unsigned long t_getreg(
     unsigned long tid,     /* task identifier */
     unsigned long regnum,  /* register number */
     unsigned long *reg_value /* register contents */
)
```

This system call enables the caller to obtain the contents of a task's notepad register. Each task has 16 such software registers, held in the task's TCB.

� ㅃㅔㄹㅎ regnum : Specifies the register number. Registers numbered 0 through 7 are for application use,

� ㅃㅔㄹㅎ reg_value : Points to the variable where t_getreg() stores the registeri» contents.

Task Suspension & Resumption

t_suspend Suspends a task indefinitely.

```
unsigned long t_suspend(
    unsigned long tid /* task identifier */
)
```

t_resume Resumes a suspended task.

```
unsigned long t_resume(
    unsigned long tid /* task identifier */
)
```

Get/Change Task Information

t_ident Obtains the task identifier of a named task.

```
unsigned long t_ident(
     char name[4],         /* task name */
     unsigned long node,    /* node number */
     unsigned long *tid     /* task ID */
)
```

This system call enables the calling task to obtain the task ID of a task it knows only by name.

t_setpri Gets and optionally changes a task's priority.

```
     unsigned long t_setpri(
     unsigned long tid,    /* task identifier */
     unsigned long newprio, /* new priority */
     unsigned long *oldprio /* previous priority */
)
```

� ㅃㅔㄹㅎ oldprio : Points to the variable where t_setpri() stores the taski» previous priority.

t_mode Gets or changes the calling task's execution mode.

```
unsigned long t_mode(
      unsigned long mask,   /* attributes to be changed */
      unsigned long new_mode, /* new attributes */
      unsigned long *old_mode /* prior mode */
)
```

� ㅃㅔㄹㅎ mask : Specifies all task attributes to be modified.

� ㅃㅔㄹㅎ new_mode : Specifies the new task attributes.

� ㅃㅔㄹㅎ old_mode : Points to the variable where t_mode() stores the old value of the task?mode.

5.2. Storage Allocation

Regions	� ㄴ ㅣ ㄺ "Malloc"-style Heap of Variable Size **Segments** �ㄴ ㅣ ㄺ No "Garbage Collection" � ㄴ ㅣ ㄺ Danger of fragmentations
Partitions	� ㄴ ㅣ ㄺ fixed-size **buffers** �ㄴ ㅣ ㄺ No danger of fragmentations � ㄴ ㅣ ㄺ Waste memory, unless you select buffer size carefully

rn_create Creates a memory region.

� ㅃㅢㄺ�ㅈㅔㄷ (pdemo): rn_create("RMEM", seg_ptr, RNSIZE, 128, 0, &rnid, &rsize);

```
unsigned long rn_create(
     char name[4],          /* region name */
     void *saddr,           /* starting address */
     unsigned long length,   /* region's size in bytes */
     unsigned long unit_size, /* region's unit of allocation */
     unsigned long flags,    /* region attributes */
     unsigned long *rnid,    /* region ID */
     unsigned long *asiz     /* allocatable size */
)
```

� ㅃㅔㄹㅎ flag

RN_PRIOR(0x2) /RN_FIFO(0x0) : Tasks are queued by priority /FIFO order.

RN_DEL(0x4) /RN_NODEL(0x0) : Region can / cannot be deleted with segments outstanding.

rn_getseg Allocates a memory segment to the calling task.

� ㅃㅢㄺ�ㅈㅔㄷ (pdemo) : rn_getseg(0, RNSIZE + 4, RN_NOWAIT, 0, &seg_ptr);

```
unsigned long rn_getseg(
    unsigned long rnid,    /* region identifier */
    unsigned long size,    /* requested size, in bytes */
    unsigned long flags,   /* segment attributes */
    unsigned long timeout,  /* timeout in clock ticks */
    void **seg_addr       /* allocated segment address */
)
```

� ㅂㅓㄹㅎ�ㄸㅓㄴㅎ�ㄹㅣㅈ segment size : region�ㅆㅡㄹㅌ unit size� ㅆㅡㄹㅌ the nearest mutiple size

� ㅃㅔㄹㅎ flag

RN_NOWAIT Don't wait for a segment.

  : rn_getseq() returns unconditionally whether or not allocation successful

RN_WAIT Wait for a segment.   : segment 媛� �ㅂㅓㄹㅎ�ㄸㅓㄴㅎ�ㄹㅣㅍ �ㅂㅡㅊ源ㄸㅜㅅ�� block�ㅁㅏㄹㅍ.

rn_retseg Returns a memory segment to the region from which it was allocated.

```
unsigned long rn_retseg(
    unsigned long rnid, /* region identifier */
    void *seg_addr /* segment address */
)
```

rn_ident Obtains the region identifier of a named region.

```
unsigned long rn_ident(
     char name[4],          /* region name */
     unsigned long *rnid   /* region identifier */
)
```

rn_delete Deletes a memory region.

```
unsigned long rn_delete (
     unsigned long rnid    /* region ID */
)
```

pt_create Creates a memory partition of fixed-size buffers.

� ㅃㅓㄹㅌ�ㅈㅔㄷ) rc = pt_create("PTN1",part_base, (void & nbsp;*) 0, LENGTH, BLOCK_SIZE, PT_NODEL, &ptid, &nbufs);

```
unsigned long pt_create(
    char name[4],          /* partition name */
    void *paddr,          /* partition physical addr. */
    void *laddr,          /* partition logical address */
    unsigned long length,  /* partition length in bytes */
    unsigned long bsize,   /* buffer size in bytes */
    unsigned long flags,   /* buffer attributes */
    unsigned long *ptid,   /* partition identifier */
    unsigned long *nbuf    /* number of buffers created */
)
```

This service call enables a task to create a new memory partition, from which fixed-sized
memory buffers can be allocated for use by the application.

- *length Specifies the total partition length in bytes.*
- *bsize Specifies the size of the buffers. bsize must be a power of 2, and equal to or greater than 4.*
- flags

  PT_GLOBAL(0x1) /PT_LOCAL(0x0)

    Partition is globally addressable by other nodes / partition can be addressed only the by local node.

  PT_DEL(0x4) /PT_NODEL(0x0)

    Deletion of the partition with pt_delete() is enabled, even if one or more buffers are allocated./

    Deletion of the partition is prohibited unless all buffers have been freed.

pt_getbuf Gets a buffer from a partition.

```
unsigned long pt_getbuf(
     unsigned long ptid,    /* partition identifier */
     void **bufaddr        /* starting address of buffer */
)
```

� ㅃㅔㄹㅎ bufaddr : �ㅂㅓㄹㅎ�ㄸㅓㄴㅎ�ㄹㅣㅈ 踰ㄲㅛㄹㅍㄸㅡㅌ�ㅆㅡㄹㅌ �ㄸㅐㄲ�ㅇㅔㄱ 二쉽ㄴㅐㅁ瑜� 媛 �瑜댄ㄱㅕㄹㅂ�ㄸㅐㄹㅁ.

pt_retbuf Returns a buffer to the partition from which it came.

```
unsigned long pt_retbuf(
     unsigned long ptid,    /* partition identifier */
     void *bufaddr          /* starting address of the buffer */
)
```

pt_delete Deletes a memory partition.

```
unsigned long pt_delete (
       unsigned long ptid /* partition identifier */
)
```

pt_ident Obtains the identifier of a named partition.

```
unsigned long pt_ident(
    char name[4],          /* partition name */
    unsigned long node,    /* node number */
    unsigned long *ptid    /* partition identifier */
)
```

pt_sgetbuf Gets a buffer from a partition.

```
unsigned long pt_sgetbuf(
    unsigned long ptid,    /* partition identifier */
    void **paddr,          /* physical address */
    void **laddr           /* logical address */
)
```

On MMU-based systems, both physical and logical addresses are returned to
simplify transfer of buffers between supervisor and user mode programs.
In non-MMU systems, the logical address is the same as the physical address,
and this call functions the same as the pt_getbuf() call.

5.3. The Message Queue

<u>ULONG q_create(char name[4], ULONG count, ULONG flags, ULONG *qid);</u>

```
unsigned long q_create(
    char name[4],          /* queue name */
    unsigned long count,   /* queue size */
    unsigned long flags,   /* queue attributes */
    unsigned long *qid     /* queue identifier */
)
```

� ㅃㅔㄹㅎ flags

Q_GLOBAL(0x1) /Q_LOCAL(0x0)

  Queue is globally addressable by other nodes/queue is addressable only by the local node.

Q_PRIOR(0x2) /Q_FIFO(0x0)

  Tasks are queued by priority / FIFO.

Q_LIMIT(0x4) /Q_NOLIMIT(0x0)

  Message queue size is limited to count / is unlimited.

Q_PRIBUF(0x8) /Q_SYSBUF(0x0)

  Private / system buffers are allocated for message storage.

q_receive Requests a message from an ordinary message queue.

```
unsigned long q_receive(
    unsigned long qid,           /* queue identifier */
    unsigned long flags,         /* queue attributes */
    unsigned long timeout,       /* timeout in clock ticks */
    unsigned long msg_buf[4]     /* message buffer */
)
```

� ㅃㅔㄹㅎ flags

Q_NOWAIT(0x1)/Q_WAIT(0x0) : Don't wait for message./ Wait for message.

q_send Posts a message to an ordinary message queue.

```
unsigned long q_send(
    unsigned long qid,           /* queue identifier */
    unsigned long msg_buf[4]     /* message buffer */
)
```

q_broadcast Broadcasts identical messages to an ordinary message queue.

```
unsigned long q_broadcast(
    unsigned long qid,           /* queue identifier */
    unsigned long msg_buf[4],    /* msg. of 4 long words */
    unsigned long *count         /* # tasks receiving msg. */
)
```

� ㅃㅔㄹㅎ Count : the number of tasks readied by the broadcast.

q_urgent Posts a message at the head of an ordinary message queue.

```
unsigned long q_urgent(
    unsigned long qid,           /* queue identifier */
    unsigned long msg_buf[4]     /* message buffer */
)
```

q_ident Obtains the queue ID of an ordinary message queue.

```
unsigned long q_ident(
    char name[4],          /* queue name */
    unsigned long node,    /* node number */
    unsigned long *qid     /* queue identifier */
)
```

q_delete Deletes an ordinary message queue.

```
unsigned long q_delete(
      unsigned long qid       /* queue identifier */
)
```

Variable Length Message Queue : Most useful for Multiprocessing System

q_vcreate Creates a variable-length message queue.

� ㅃㅡㅣㄹㅌ�ㅈㅔㄷ) rc=q_vcreate("MYVQ",Q_GLOBAL|Q_PRIOR, 5, manlen,& nbsp;&qid);

```
unsigned long q_vcreate(
    char name[4],          /* queue name */
    unsigned long flags,   /* queue characteristics */
    unsigned long maxnum,  /* maximum number of messages that can be pending at on time at the queue*/
    unsigned long maxlen,  /* maximum message length (in bytes) */
    unsigned long *qid     /* queue identifier */
)
```

� ㅃㅔㄹㅎ flags

Q_GLOBAL /Q_LOCAL

Queue is globally addressable by other nodes /queue is addressable only by the local node.

Q_PRIOR /Q_FIFO

Tasks are queued by priority / FIFO.

q_vreceive Requests a message from a variable-length message queue.

```
unsigned long q_vreceive(
    unsigned long qid,     /* queue identifier */
    unsigned long flags,   /* queue attributes */
    unsigned long timeout,  /* timeout in clock ticks */
    void *msg_buf,         /* message buffer */
    unsigned long buf_len,  /* length of buffer */
    unsigned long *msg_len  /* length of message */
)
```

� ㅃㅔㄹㅎ flags :

Q_NOWAIT /Q_WAIT Don't wait / wait for message..

q_vsend Posts a message to a specified variable-length message queue.

```
unsigned long q_vsend(
    unsigned long qid,     /* queue identifier */
    void *msg_buf,         /* message buffer */
    unsigned long msg_len,  /* length of message */
)
```

q_vbroadcast Broadcasts identical variable-length messages to a message queue.

```
unsigned long q_vbroadcast(
    unsigned long qid,    /* queue identifier */
    void *msg_buf,        /* message buffer */
    unsigned long msg_len,  /* length of message */
    unsigned long *count   /* number of tasks */
)
```

q_vurgent Posts a message at the head of a variable-length message queue.

```
unsigned long q_vurgent(
    unsigned long qid,     /* queue identifier */
    void *msg_buf,         /* message buffer */
    unsigned long msg_len,  /* length of message */
)
```

q_vident Obtains the queue ID of a variable-length message queue.

```
unsigned long q_vident(
    char name[4],          /* queue name */
    unsigned long node,   /* node number */
    unsigned long *qid    /* queue identifier */
)
```

q_vdelete Deletes a variable-length message queue.

```
unsigned long q_vdelete(
  unsigned long qid      /* queue identifier */
)
```

```
�ㅃㅣㅆ� ㅆㅒㄸ 2:29 99-06-15

TestVqueue()
{
    unsigned long  tid, qid, rc, received, msg[5],args[4];

    if (rc = q_vcreate("SRVq", Q_LOCAL|Q_FIFO, 400, 20, &qid))
        printf("err");
    if (rc = t_create("SRVt", 150, 4096, 0, T_LOCAL | T_NOFPU, &tid))
          printf("err");
    if (rc = t_start(tid, T_PREEMPT|T_NOTSLICE|T_NOASR|T_SUPV|T_ISR, ServerTask, args))
        printf("err");

    if (rc = q_vident("SRVq", 0, &qid))
        printf("err");

    msg[0]=0xfff80001;
    msg[1]=0xfff80002;
    msg[2]=0xfff80003;
    msg[3]=0xfff80004;
    msg[4]=0xfff80005;
    if (rc = q_vsend(qid, &msg, sizeof(msg)))
         printf("err");
}

static void ServerTask(void)
{
    unsigned long rc, qid, tid, msg[5];
    unsigned long msglen,i;

    if (rc = q_vident("SRVq", 0, &qid))      printf("err");

    for (;;)
    {
        if (rc = q_vreceive(qid, Q_WAIT, 0, &msg, 20, &msglen))
            printf("err");

        printf("%d\n",msglen);
        printf("0x%x\n",msg[0]);
    }
}
```

5.4. Event - Synchronization by event facility

16bit system event flags	16bit user event flags

ev_receive Enables a task to wait for an event condition.

� ㅃㅢㄾ�ㅈㅔㄷ) errcode=ev_receive(0x9, EV_WAIT|EV_ANY, 100, & amp;events_r);

```
unsigned long ev_receive(
    unsigned long events,  /* bit-encoded events */
    unsigned long flags,   /* event processing attributes */
    unsigned long timeout,  /* timeout delay */
    unsigned long *events_r /* events received */
)
```

� ㅃㅔㅀ events : the set of events.

� ㅃㅔㅀ flag

EV_NOWAIT / EV_WAIT

Return if the event condition is unsatisfied /block until the event condition is satisfied.

EV_ANY /EV_ALL

Wait for ANY("OR") / ALL of the desired events.("AND")

� ㅃㅔㅀ timeout : If EV_WAIT is set, the timeout parameter specifies the timeout in units of clock ticks.

If the value of timeout is 0, ev_receive() waits indefinitely.

� ㅃㅔㅀ events_r : the actual events captured.

ev_send Sends events to a task.

```
unsigned long ev_send(
    unsigned long tid,     /* target task identifier */
    unsigned long events   /* bit-encoded events */
)
```

� ㅃㅢㄾ�ㅈㅔㄷ : Timer 遺�遺� 李몄“

5.5. Semaphore

P(S)	IF S>0
	then S := S - 1
	else (wait on S)

V(S)	If (one or more processes are waiting on S) then (let one of the these processes proceed) else S := S + 1

sm_create Creates a semaphore.

```
unsigned long sm_create(
    char name[4],         /* semaphore name */
    unsigned long count,   /* number of tokens */
    unsigned long flags,   /* semaphore attributes */
    unsigned long *smid    /* semaphore identifier */
)
```

� ㅃㅔㅀ flags

SM_GLOBAL /SM_LOCAL : Semaphore can be addressed by other nodes /local nodes only.

SM_PRIOR /SM_FIFO : Tasks are queued by priority / FIFO order.

sm_p Acquires a semaphore token.

```
unsigned long sm_p(
    unsigned long smid,    /* semaphore identifier */
    unsigned long flags,   /* attributes */
    unsigned long timeout  /* timeout */
)
```

� ㅃㅕㅀ flags

SM_WAIT/SM_NOWAIT

: Block until semaphore is available./ Return with error code if semaphore is unavailable.

sm_v Releases a semaphore token.(giVe up)

```
unsigned long sm_v(
    unsigned long smid     /* semaphore identifier */
)
```

� ㅃㅕㅄ Semaphore �ㄱㅝㅀ�ㅅㅠㅆ �ㅃㅢㄾ:

��� Critical Region

� ㄴㅣㄾ Initial count = 1

� ㄴㅣㄾ sm_p() : to Enter critical region

� ㄴㅣㄾ sm_v() : to Leave critical region

��� Resource Limit

� ㄴㅣㄾ Initial count = number of equivalent resource

� ㄴㅣㄾ sm_p() : to Gain Access to a Resource

� ㄴㅣㄾ sm_v() : to Release a Resource

sm_delete Deletes a semaphore.

```
unsigned long sm_delete(
    unsigned long smid    /* semaphore ID */
)
```

sm_ident Obtains the semaphore identifier of a named semaphore.

```
unsigned long sm_ident(
    char name[4],          /* semaphore name */
    unsigned long node,    /* node selector : single node� ㅆㅣㄲ 寃쏙ㅅㅡㄲ 0 */
    unsigned long *smid    /* semaphore ID */
)
```

	Pro's	Con's	Attributes
Events	Tasks can wait on combinations of conditions.	Can't count. Can't carry data. ISR can't receive events.	Created as part of each task. Events sent to specified task only. Timer services are available to send events to tasks.
Semaphores	Can count. Tasks and ISRs can use semaphores(ISR with SM_NOWAIT)	Can't carry data. Tasks can wait on only 1 semaphore at a time.	Must be created. Operations are on semaphores, not on task(s). Multiple tasks can use a semaphore.
Message queue	Can carry data. Can Count. (Duplicate message OK) Tasks and ISRs can receive messages.(ISR with Q_NOWAIT)	Services take more time. Tasks can wait on only 1 messae queue at a time.	Must be created. Messages sent to queue, not on task(s). Multiple tasks can use a queue.

5.6. . ASR (Asynchronous Signal Routine)

��� An ASR may be assigned a specific task.

��� Then a task has 2 parts: Main Body, and ASR

��� ASR may execute asynchhronously fro Main Body of Task.

��� ASR Code executes only when its Task would be the Running Task

��� Using signals, one task or ISR can selectively force another task out of its normal locus of execution -
that is, from the task's main body into its ASR.

��� Signals provide a "software interrupt" mechanism.

(interrupt��� �ㄸㅐㄻ瑜몃 ㅈㅓㅀ : �ㅆㅢㅋ�ㄲㅘㅎ�ㅇㅡㄽ�ㄷㅠㄳ��� �ㄸㅐㅄ由� ASR��� task�ㅆㅡㄾ �ㄱㅣㄾ�ㄲㅐㄵ� ㅃㅛㅌ 利ㄷㅚㅌㄸㅐㄲ �ㅃㅢㄶ�ㅃㅕ
ㅍ�ㅆㅠㅀ 誘몃ㅌㅟㄲ吏� �ㅂㅡㅇ�ㄷㅔㅀ�ㄸㅐㄻ.)

as_catch Specifies an ASR

```
unsigned long as_catch(
    void (* start_addr) (), /* ASR address */
    unsigned long mode /* ASR attributes */
)
```

� ㅃㅕㅀ Mode

T_PREEMPT /T_NOPREEMPT : ASR is / is not preemptible.

T_TSLICE /T_NOTSLICE : ASR can / cannot be time-sliced.

T_ASR /T_NOASR : ASR nesting enabled/disabled.

T_USER /T_SUPV : ASR runs in user mode / supervisor mode.

T_ISR /T_NOISR : Interrupts are enabled / disabled while ASR runs.

T_LEVELMASK0 through T_LEVELMASK : Certain interrupts are disabled while ASR runs.

as_send Sends asynchronous signals to a task.

```
    unsigned long as_send(
    unsigned long tid,     /* target task ID */
    unsigned long signals  /* bit-encoded signal list */
    )
```

The purpose of these signals is to force a task to break from its normal flow of execution and execute its Asynchronous Signal Routine (ASR).

as_return Returns from an ASR

unsigned long as_return();

This system call must be used by a task's ASR to exit and return to the original flow of execution of the task.

The purpose of this call is to enable the pSOS+ kernel to restore the task to its state before the ASR.

as_return() cannot be called except from an ASR.

This call is analogous to the i_return() call, which enables an Interrupt Service Routine (ISR) to return to the interrupted flow of execution properly.

5.7. Time

5.7.1. Announce a Clock Tick to pSOS+

tm_tick Announces a clock tick to the pSOS+ kernel.

unsigned long tm_tick()

��� clock tick frequency : pSOS+ Configuration Table� ㅆㅡㄹㅌ **kc_ticks2sec**瑜 � �ㄴㅔㅊ�ㅂㅣㄺ

If this value is specified as 100, the system time manager will interpret 100 tm_tick() system calls to be one second, real time.

��� BSP� ㅆㅡㄹㅌ timer 猷⑦ㄸㅒㅍ�ㅃㅛㅌ�ㄲㅘㄴ RtcIsr(void) �ㅂㅓㄹㅌ�ㅃㅛㅌ�ㄲㅘㄴ �ㄱㅝㄶ�ㅅㅠㅆ

5.7.2. Calendar Date and Time

tm_set Sets or resets the system�� ㅃㅡㅍ version of the date and time.

```
unsigned long tm_set(
    unsigned long date,    /* year/month/day */
    unsigned long time,    /* hour:minute:second */
    unsigned long ticks    /* clock ticks */
)
```

� ㅃㅖㅀ date : Year(16bits) + Month(8bits) + Day(8bits)

� ㅃㅖㅀ date : Hour(16bits) + Minute(8bits) + Second(8bits)

� ㅃㅖㅀ ticks : the number of ticks from the last second of the time arguement.

```
/*--------------------------------------------------------------------*/
/* Set date to May 1, 1995, time to 8:30 AM, and start the system      */
/* clock running.                                                  */
/*--------------------------------------------------------------------*/
date = (1995 << 16) + (5 << 8) + 1;
time = (8 << 16) + (30 << 8);
ticks = 0;
tm_set(date, time, ticks);
```

tm_get Obtains the system's current version of the date and time.

```
unsigned long tm_get(
    unsigned long *date,   /* year/month/day */
    unsigned long *time,   /* hour:minute:second */
    unsigned long *ticks   /* ticks */
)
```

5.7.3. Time-based Awakening a Task

tm_wkafter Blocks the calling task and wakes it after a specified interval.

```
unsigned long tm_wkafter(
    unsigned long ticks    /* clock ticks */
)
```

tm_wkwhen Blocks the calling task and wakes it at a specified time.

```
unsigned long tm_wkwhen(
    unsigned long date,    /* year/month/day */
    unsigned long time,    /* hour:minute:second */
    unsigned long ticks    /* clock ticks */
)
```

5.7.4. Send Events to Calling Task

tm_evafter Sends events to the calling task after a specified interval.

```
unsigned long tm_evafter(
    unsigned long ticks,   /* delay */
    unsigned long events,  /* event list */
    unsigned long *tmid    /* timer identifier */
)
```

tm_evevery Sends events to the calling task at periodic intervals.

```
unsigned long tm_evevery(
      unsigned long ticks,   /* delay */
      unsigned long events,  /* event list */
      unsigned long *tmid   /* timer identifier */
)
```

tm_evwhen Sends events to the calling task at a specified time.

```
unsigned long tm_evwhen(
     unsigned long date,    /* date of wakeup */
     unsigned long time,    /* time of wakeup */
     unsigned long ticks,   /* ticks at wakeup */
     unsigned long events,  /* event list */
     unsigned long *tmid    /* timer identifier */
)
```

5.7.5. Cancel an Armed Timer

tm_cancel Cancels an armed timer.

```
unsigned long tm_cancel(
     unsigned long tmid     /* timer identifier */
)
```

```
�ㅃㅣㅆ� ㅆㅐㄸ 3:05 99-06-25
/* How to use tm_evafter */

#define EV_TIMER         1
#define EV_START_TIMEOUT 2
#define EV_END_TIMEOUT           4

void timer_task(void);
TestTmEvafter()
{
        ULONG   timerTaskID,tmid;

    if (t_create("TIME", 100, 15000,15000,T_LOCAL|T_NOFPU, &timerTaskID) != 0)
          printf("Task creation error");

    if (t_start(timerTaskID,T_PREEMPT|T_NOTSLICE|T_NOASR|T_SUPV|T_ISR, timer_task, 0) != 0)
          printf("Task start error");

        ev_send(timerTaskID,EV_START_TIMEOUT);
}

void timer_task(void)
{
        unsigned long tmid;
        unsigned long waiton = EV_TIMER|EV_START_TIMEOUT|EV_END_TIMEOUT;
        /* any of the events */
        unsigned long ev_rcvd  = 0;

        /*-----------------------------------------------------------------------*/
        /* then update the system time every time timer goes off                */
        /*-----------------------------------------------------------------------*/
        while (1)
        {
                unsigned long events;

                if ((ev_receive(waiton, EV_WAIT|EV_ANY, 0, &events)) != 0)
        {
                perror("timer_task: ev_receive() error");
                continue;
        }

                if (events & EV_START_TIMEOUT)
                {
                        printf("**");
                        tm_evafter(600, EV_END_TIMEOUT, &tmid ); /* 6 sec later */
                        continue;
                }
                else if (events & EV_END_TIMEOUT)
                        printf("@");
  }
}
```

6. Device I/O

de_close Closes an I/O device.

```
unsigned long de_close(
     unsigned long dev,     /* major/minor device number */
     void *iopb,          /* I/O parameter block address */
     void *retval         /* return value */
)
```

The de_close() call invokes the device close routine of a pSOS+ device driver specified by the dev argument.

� ㅃㅔㄹㅎ dev : Specifies the major and minor device numbers,

� ㅃㅔㄹㅎ iopb : Points to an I/O parameter block,

� ㅃㅔㄹㅎ retval : Points to a variable that receives a driver-specific value returned by the driver.

de_cntrl Requests a special I/O device service.

```
unsigned long de_cntrl(
    unsigned long dev,    /* major/minor device number */
    void *iopb,           /* I/O parameter block address */
    void *retval          /* return value */
)
```

The de_cntrl() call invokes the device control routine of a pSOS+ device driver specified by the dev argument. The functionality of a device control routine depends entirely on the device driver implementation. It can include anything that cannot be categorized under the other five I/O services. de_cntrl() for a device can be used to perform multiple input and output subfunctions. In such cases, extra parameters in the I/O parameter block can designate the subfunction.

de_init Initializes an I/O device and its driver.

```
unsigned long de_init(
    unsigned long dev,     /* major/minor device number */
    void *iopb,            /* I/O parameter block */
    void *retval,          /* return value */
    void **data_area       /* device data area */
)
```

The de_init() call invokes the device initialization routine of the pSOS+ device driver specified by the dev argument.

The drive init routine can perform one-time device initialization functions such as:

��� Resetting the devices

��� Setting the necessary programmable registers

��� Allocating and/or initializing the driver's data area (for pointers, counters, and so on)

��� Creating the messages queues, semaphores, and so on, that are needed for communication and synchronization

��� Installing the interrupt vectors, if necessary

de_open Opens an I/O device.

```
unsigned long de_open(
    unsigned long dev,     /* major/minor device number */
    void *iopb,            /* I/O parameter block address */
    void *retval           /* return value */
)
```

The de_open() call invokes the device open routine of a pSOS+ device driver specified by the dev argument.

The device open routine can be used to perform functions that need to be done before the I/O operations can be performed on the device. For example, an asynchronous serial device driver can reset communication parameters (such as baud rate and parity) to a known state for the channel being opened.

A device driver can also assign specific duties to the open routine that are not directly related to data transfer or device operations. For example, a device driver can use de_open() to enforce exclusive use of the device during several read and/ or write operations.

de_read Reads from an I/O device.

```
unsigned long de_read(
    unsigned long dev,     /* major/minor device number */
    void *iopb,            /* I/O parameter block address */
    void *retval           /* return value */
)
```

The de_read() call is used to read data from a device. It invokes the device read routine of a pSOS+ device driver specified by the dev argument. This service normally requires additional parameters contained in the I/O parameter block, such as the address of a data area to hold the data and the number of data units to read.

de_write Writes to an I/O device.

```
unsigned long de_write(
    unsigned long dev,     /* major/minor device number */
    void *iopb,            /* I/O parameter block address */
    void *retval           /* return value */
)
```

The de_write() call is used to write to a device. It invokes the device write routine of a pSOS+ device driver specified by the dev argument. This service normally

requires the additional parameters contained in the I/O parameter block, such as
the address of the user's output data and the number of data units to write.

7. Tip

```
#define START_CRITICAL   { ULONG oldMode; t_mode (1, T_NOPREEMPT, &oldMode);}
#define END_CRITICAL    { ULONG oldMode; t_mode (1, T_PREEMPT, &oldMode); }
```