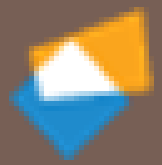

LECTURE 06

REAL TIME LINUX

CE 705

Realtime Systems

Dr. Salem Al-Agtash

Outline

- Commercial real time OS
- Real Time Linux
- RT Linux Modules
- Advance APIs

Commercial Real-Time Operating Systems

- There are currently about 100 RTOS vendors.
- Products are available for 8, 16 and 32 bit microprocessors. Some of these packages are complete operating systems and include a real-time kernel, an input/output manager, windowing systems, a file system, networking, language interface libraries, debuggers and cross-platform compilers.
- The cost of a RTOS varies between $100 to well over $10,000.
- With so many vendors, the difficulty is in the selection process.
- Some well known RTOS are:

  uC/OS-II, VxWorks, pSOS, OSEK, iRMX, QNX, and RTLinux

RTOS for small embedded systems.

- Many small embedded systems such as engine controls, intelligent instruments, robots, computer peripherals and telecommunications equipment can benefit from the use of a RTOS. Such systems are generally designed around an 8-bit microprocessor. With a 64 KB address space, most 8-bit microprocessors cannot afford a memory hungry RTOS.
- Other features for a small RTOS are:
  - Low cost.
  - Have minimum interrupt latency.
  - Deterministic execution time for all kernel services.
  - Be able to manage at least 20 tasks.
  - Provide at least the following services:
    - Allow tasks to be dynamically created and deleted.
    - Provide semaphore management services.
    - Allow time delays and timeouts on kernel services.

Capabilities of Commercial RTOS

- **Conformance to Standards:** The OS is compliant or partially compliant to Real-Time POSIX API standard, preemptive fixed-priority scheduling, standard synchronization primitives.
- **Modularity and Scalability:** Small and configurable. The OS can be scaled down to fit with application in ROM in small embedded system. It can be configured to provide I/O, file and networking.
- **Speed and Efficiency:** Micro-kernel based. Timing figures as context switch time, interrupt latency, semaphore get/release are typically small (one to few microseconds).
- **System calls:** Nonpreemptable portion of kernel functions short and deterministic.

- **Split interrupt handling:** Nonpreemptable portion of interrupt handling and the execution times of immediate interrupt handling routines are kept small.
- **Scheduling:** All OS offer at least 32 priority levels, some offer 128, 256. They have FIFO or RR policy for scheduling equal priority threads.
- **Priority Inversion Control:** The OS provides priority inheritance but may allow to disable it to save the overhead of this mechanism.
- **Clock and Timer Resolution:** The OS provides a nominal timer resolution to nanoseconds.
- **Memory management:** OS may provide virtual-to-physical address mapping but does not do paging.
- **Networking:** The OS can be configured to support TCP/IP, stream etc.

- Networking support
  - BSD 4.4 TCP/IP networking
  - IP, IGMP, CIDR, TCP, UDP, ARP, PPP, SLIP/CSLIP
  - Standard Berkeley Sockets, Berkeley packet filter
  - RIPv1/v2 and routing sockets
  - TFTP, BOOTP, DHCP, DNS, FTP, rlogin, telnet, rsh
  - NFS client and server, ONC RPC, SNTP
  - SNMPv1/v2c/v3
  - Network Protocol Toolkit
  - OSPFv2, BGP- 4, Ipsec/IKE
  - NAT, L2TP, MPLS
  - ATM, Frame relay, ISDN, SS7

. . .

- Fast, flexible I/O and local file system
  - POSIX asynchronous I/O and directory handling
  - SCSI support
  - MS-DOS compatible file system
  - TrueFFS flash file system (optional)
  - ISO 9660 CD-ROM file system
  - PCMCIA support

- Support platforms
  - Motorola/IBM PowerPC architecture family
  - Intel® Pentium, ® Pentium II, and Pentium III architecture families
  - Intel™ Celeron architecture family
  - ARM/strong ARM architecture family
  - MIPS architecture family
  - Average latency 1.7 microseconds, maximum latency 6.8 microseconds on a 200MHz Pentium machine.

RT Linux

- Linux provides most of the real-time POSIX(1003.1b) API functions, but has many shortcomings when used for real-time applications.
- One of the most serious problem is: disabling of interrupts by subsystems when they are in critical sections.
- Scheduling: Linux provides individual processes with the choice of SCHED_FIFO, SCHED_RR, or SCHED_OTHER policies.
- There are 100 priority levels.
- Clock and Timer Resolutions: Linux updates the system clock and checks for timer expirations periodically. The actual resolution of timers is 10 milliseconds.

....

- There are "spin-offs" of the standard Linux kernel that provide hard real-time performance, or that are targeted to embedded use.
- There are two methods to enhance the real-time nature of Linux:
  - Method 1: To modify the structure and processing on Linux
  - Method 2: To make RTOS to coexist with Linux.
- TimeSys Linux and MontaVista use the first method to provide hard real-time
- There are two major developments at the RTOS level: RTLinux and RTAI. RTAI forked off an earlier version of RTLinux.
- RTLinux and RTAI do basically the same thing. They make their sources available, they have partial POSIX compliance, but they don't use compatible APIs.

RTLinux Features

- RTLinux is a hard real time operating system that coexists with the Linux OS.
- It is possible to create POSIX threads that will run precisely specified moments of time.
- **Real time programs are executed in kernel space and have little or no protection**
- Allows **communication** between real time threads and Linux user processes (FIFOs and Shared Memory)
- **Task Synchronization through**: Wakeup/Suspend, Mutex and semaphore.
- **Real time device driver**: Accessing Physical memory and I/O ports. Soft and Hard interrupts.

Design goal of RTLinux

- It is not feasible to identify and eliminate all aspects of kernel operation that lead to unpredictability.
- The sources of unpredictability:
  - Linux scheduling algorithm (optimized to maximize throughput)
  - Device drivers
  - Uninterruptible system calls
  - Use of interrupt disabling
  - Virtual memory operations
- To avoid these problems construct a small predictable kernel.
- This approach has the added benefit of maintainability.

- The basic Linux kernel without hard real time support separates the hardware from the user level task.
- The kernel has ability to suspend any user level task, once that task has outrun the "slice of time" allotted to it by the CPU.
- In trying to be "fair" to all tasks, the kernel can prevent critical events from occurring.
- For example, that a user task controls a robotic arm, The standard Linux kernel could potentially preempt the task and give the CPU to one which is less critical (e.g. one that boots up Netscape).
- Consequently, the arm will not meet strict timing requirements.

Priorities

- The **priority** of Linux tasks and RTOS are defined as follows:
  - Interrupt handling on RTOS — High
  - Kernel, Scheduler on RTOS
  - Task on RTOS
  - Interrupt handling on Linux
  - Kernel, Bottom handler and scheduler on Linux
  - Processes on Linux — Low

Understanding RTLinux Program

- RTLinux programs are created as modules and loaded into the kernel space.
- A Linux module is nothing but an object file, usually created with the -c flag argument to gcc.
- The main() function is replaced by a pair of init/cleanup functions:

  init init_module();

  void cleanup_module();
- The init_module() is called when the module is first loaded into the kernel.
- The cleanup_module() is called when the module is unloaded.

• • •

- To run any RTLinux program, you must first insert scheduler and support modules.

rtl.o, rtl_time.o, rtl_sched.o, rtl_posixio.o, rtl_fifo.o and/or mbuff.o.

- Use any of the following:

  rtlinux

  insmod

  modprobe

- the **insrtl** script file can be used to load all rtlinux modules and removed using **rmrtl** script file.

rtl_printf – print formatted output

- The RTLinux programs use rtl_printf() to display the messages, instead of printf().
- The **rtl_printf** function converts and formats its arguments similar to printf and places output to the kernel message buffer. It is safe to use from RTLinux threads and interrupt handlers.

```
#include <rtl_printf.h>
int rtl_printf(const char *format, ...);
```

RT Linux tasks

- **Initial Design –** Each RT task executed in its own address space. Which has the following drawbacks:
  - High overhead of context switching as TLB had to be invalidated.
  - High overhead of system calls.
- All RT tasks run in the same address space (in the kernel space) and at the highest privilege level. But highly error prone as a bug in a single application can crash entire system.

. . .

- RT tasks run as kernel modules. Can be dynamically added.
- Tasks have integer context for faster context switching (unless FP context is explicitly requested).
- Hardware context switching provided by x86 is not used.
- Task resources should be statically allocated (kmalloc etc) and should not be used within an RT task).

Creating RTLinux POSIX Threads

- To create a new real time thread, use pthread_create() function.
- This function must be called from the Linux kernel thread (i.e. using init_module()):

```
#include <pthread.h>

int pthread_create(pthread_t *thread,
        pthread_attr_t *attr,
        void *(*start_routine)(void *), void *arg);
```

- The ID of the newly created thread is stored in the location pointed to by "thread". The function pointed to by start_routine is taken to be the thread code. It is passed the "arg" argument.
- The thread is created using the attributes specified in the "attr" thread attributes object. If attr is NULL, default attributes are used.

. . .

- To cancel a thread, use the POSIX function:

  pthread_cancel(pthread_t thread);

  from the cleanup_module().

- Use **pthread_delete_np()** instead of **pthread_cancel()/ pthread_join().**

Scheduling threads

- RTLinux provides scheduling, which allows thread code to run at specific times. It uses a priority driven scheduler, in which highest priority (thread) is always chosen to run.

int pthread_setschedparam(pthread_t thread, int policy,

const struct sched_param *param);

- The policy argument is currently not used in RTLinux, but should be specified as SCHED_FIFO for compatibility with future versions.
- The structure sched_param contains the sched_priority member.
- Higher values correspond to higher priorities. Use:
  - **sched_get_priority_max() , and**
  - **sched_get_priority_min()**

to determine possible values of sched_priority.

- To make a realtime thread execute periodically, users may use the non-portable function:

- **int pthread_make_periodic_np(pthread_t thread, hrtime_t *start_time*, hrtime_t *period*);**
- **int pthread_make_periodic_np(pthread_t thread, struct itimerspec *its);**

- which marks the thread as periodic. Timing is specified by the itimer structure its. The it_value member of the passed struct itimerspec specifies the time of the first invocation; the it_interval is the thread period.

```
struct itimerspec {
        struct timespec it_interval;    /* timer period */
        struct timespec it_value;       /* timer expiration */
};
```

- This function suspends the execution of the calling thread until the time period.

int pthread_wait_np(void);

A simple “Hello World” RTLinux program

- This small program uses all the basic APIs. It will execute two times per second, and during each iteration it will print the message:

  I’m here, my arg is 0

```
#include <rtl.h>
#include <time.h>
#include <pthread.h>
pthread_t thread;
void * start_routine(void *arg) {
        struct sched_param p;
        p . sched_priority = 1;
        pthread_setschedparam (pthread_self(), SCHED_FIFO, &p);
        pthread_make_periodic_np (pthread_self(), gethrtime(), 500000000);
        while (1) {
                pthread_wait_np();
                rtl_printf("I'm here; my arg is %x\n", (unsigned) arg);
        }
        return 0; }
```

```
int init_module(void) {
        return pthread_create (&thread, NULL, start_routine, 0);
}
void cleanup_module(void) {
        pthread_cancel (thread);
        pthread_join (thread, NULL);
}
```

Compiling and executing "Hello World"

- Creating the Makefile that will be used to compile hello.c program. Type the following into a file called Makefile and put it in the same directory as your hello.c program:

  hello.o: hello.c

  gcc $(CFLAGS) hello.c

- Locate the file rtl.mk and copy it into the same directory as hello.c and Makefile files. The rtl.mk file can usually be found at /usr/include/rtlinux/rtl.mk. cp /usr/include/rtlinux/rtl.mk. (Note the trailing dot (.).)
- Now, type the following:
  - make -f rtl.mk hello.o
- Now load the RTLinux modules:
  - rtlinux start hello
  - rtlinux status hello
  - rtlinux help
  - rtlinux stop hello

Simple Thread example:

Thread.c

```
#include <rtl.h>
#include <time.h>
#include <pthread.h>
pthread_t thread;
void * start_routine (void *arg) {
      rtl_printf (" ***I am  in Start Routine***\n");
  return 0;  }
int init_module (void) {
    int iRet;
     rtl_printf("<1>***I am in init Module***\n");
     iRet = pthread_create (&thread, NULL, start_routine,0);
          if( iRet == 0)
            rtl_printf("<1>***Thread Creation Success***\n");
           else
            rtl_printf("<1>***Thread Creation Failed***\n");
   return 0; }
void cleanup_module (void) {
  rtl_printf("<1>***I am in clean up module***\n");
  pthread_cancel(thread);
  pthread_join(thread,NULL);
  pthread_delete_np (thread);
  rtl_printf("<1>*****BYE*******\n");
}
```

Makefile

```
all: thread.o
include ./rtl.mk
test: all
        @echo "Type <return> to continue"
        @read junk
        -rmmod thread
        (cd ../../; scripts/rmrtl)
        @echo "Type <return> to continue"
        @read junk
        (cd ../../; scripts/insrtl)
        @echo "\nNow start the real-time tasks  module\n"
        @echo "Type <return> to continue"
        @read junk
        @insmod thread.o
stop_test:
        -rmmod thread
        -rmmod rtl_fifo
        -rmmod rtl_sched
        -rmmod rtl_time
clean:
        rm -f *.o
```

Steps to run

1. For compiling, Type make
2. For loading, Type /usr/rtlinux/bin/rtlinux start thread.o
3. For unloading, Type /usr/rtlinux/bin/rtlinux stop thread

RT FIFO example

rtfifo.c

```
#include <rtl.h>
#include <rtl_fifo.h>
#include <time.h>
#include <rtl_sched.h>
#include <rtl_sync.h>
#include <pthread.h>
#include <unistd.h>
#include <rtl_debug.h>
#include <errno.h>
#define IN_FIFO_ID  1
#define OUT_FIFO_ID 2
#define IN_FIFO_LENGTH  0x100
#define OUT_FIFO_LENGTH 0x100
pthread_t thread;
// RT FIFO invokes this function every time the user process writes
// something into /dev/rtf1
int pd_do_aout(unsigned int fifo) {
    int err;
    u32 ao_value;
    while ((err = rtf_get(IN_FIFO_ID, &ao_value, sizeof(u32)))
    ==  sizeof(u32)) {
        rtl_printf("%x\n", ao_value);
    //    pd_aout_write(board, ao_value);
```

...

```
    if (err != 0) return -EINVAL; else return 0;
}
void *pp_thread_ep(void *rate)  // our periodic thread
{
    int ret;
    u16 ain_data;
    pthread_make_periodic_np (pthread_self (), gethrtime (),
1000000000);
    ain_data = 0xffff;
    while (1)
    {
      //ret = pd_ain_read(board, &ain_data); // read value
from analog in
      // write to the output FIFO where user process can read it
from /dev/rtf2
      ret = rtf_put(OUT_FIFO_ID, &ain_data, sizeof(u16));
      ain_data--;
      pthread_wait_np ();
    }
}
```

RT FIFO example

...

```
int init_module (void) {
  int thread_status;
  int fifo_status;
  rtl_printf ("<1>***I am in init_module****\n");
  // free up the resource, just in case
  rtf_destroy(IN_FIFO_ID);
  rtf_destroy(OUT_FIFO_ID);
  // create fifos we can talk via /dev/rtf1 and /dev/rtf2
  rtf_create(IN_FIFO_ID, IN_FIFO_LENGTH);   // rt task <- user process
  rtf_create(OUT_FIFO_ID, OUT_FIFO_LENGTH); // rt task -> user
process
  rtf_create_handler(IN_FIFO_ID, &pd_do_aout);
  thread_status = pthread_create (&thread, NULL, pp_thread_ep, (void
*) 1);
  if (thread_status != 0) {
      rtl_printf ("<1>failed to create RT-thread: %d\n", thread_status);
      return -1;
  }
  else {
      rtl_printf ("<1>***created RT-thread***\n");
  }
return 0;
}
```

...

```
void cleanup_module (void) {
  rtl_printf ("<1>***I am in clean_up module***\n");
  pthread_cancel (thread);
  pthread_join (thread, NULL);
  rtf_destroy(IN_FIFO_ID);     // free up the resource, just
in case
  rtf_destroy(OUT_FIFO_ID);    // free up the resource,
just in case
  rtl_printf ("<1>*******BYE*******\n");
}
```

```
#include <stdio.h>
#include <errno.h>
#include <sys/time.h>
#include <sys/types.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/ioctl.h>
int main () {
  int fd1, fd2;
  int n, x = 0;
  unsigned short y;
  if ((fd1 = open ("/dev/rtf1", O_RDWR)) < 0) {
     fprintf (stderr, "Error opening /dev/rtf1\n");
     exit (1);
   }
  if ((fd2 = open ("/dev/rtf2", O_NONBLOCK)) < 0) {
     fprintf (stderr, "Error opening /dev/rtf2\n");
     exit (1);
   }
```

```
while (1) {
     n = write(fd1, &x, sizeof(x));
     x++;
     sleep (1);
     n = read(fd2, &y, sizeof(y));
     if (n != 0)
     {
       printf("User %x\n", y);
     }
  }

  return 0;
}
```

Using Shared Memory

For shared memory mbuff driver is provided. First, mbuff.o module must be loaded in the kernel. Two functions are used to allocate blocks of shared memory, connect to them and deallocates them.

```
#include <mbuff.h>
void *mbuff_alloc(const char *name, int size);
void mbuff_free(const char *name, void *mbuf);
```

The argument name is identifier to the shared memory.

The reference count (how many are connecting on this buffer) for this block is set to 1. On success, the pointer to the newly allocated block is returned. NULL is returned on failure.

RT FIFO vs. Shared Memory

- Queue data, no protocol needed to prevent overwrites.
- Message boundaries not maintained.
- Support blocking for synchronization, no polling required.
- FIFOs are point-to-point channels.
- Maximum number of FIFOs statically defined.

- No queuing of data. Need to follow explicit protocol
- Can write structured data into memory.
- Need to poll for synchronization.
- Can have any number of tasks sharing memory.
- Physical memory is the only limit.

Mapping functions

- The C library provides three functions in the header file sys/man.h

```
caddr_t mmap(caddr_t addr, size_t len, int prot, int flags,
                        int fd, off_t off);
int   munmap(caddr_t addr, size_t len);
```

- The mmap() function makes use of mmap system call mmap, which in turn calls do_map(). The file descriptor fd must be opened before the call.
- The munmap() makes use of the system call munmap to remove memory areas mapped to the user segment.

Example

```
#include <rtl_fifo.h>
….
int init_module (void)
{
  char *rtshmbase;
  int I;
  rtl_printf ("\n START OF INIT MODULE\n");
  rtshmbase = (long*) ioremap(0x3f00000, 0x100000);
//64MB-1
  for (i = 0; i < 8; i++)
    *rtshmbase++ = i+0x41;
  return 0;
}
void cleanup_module (void)
{
  rtl_printf ("clean up module\n");
  iounmap(0x3f00000);
}
```

USER Process

```
#include <sys/mman.h> …..
int main() {
  char *rtshm_ptr;
  int fd, I;
  if ((fd = open("/dev/mem", O_RDWR)) < 0)  {
    printf("Open Error\n");
    exit(1);
  }
  rtshm_ptr = (char * ) mmap (0, 0x100000,
                  PROT_READ | PROT_WRITE,
                 MAP_SHARED, fd, 0x3F00000);
  if (rtshm_ptr == MAP_FAILED)  {
    printf("Open Error\n");
    exit(1);
  }
  else  {
     for (i = 0; i < 8; i++)
                printf("%c", *rtshm_ptr++);
  }
  printf("\n");
  munmap(0x3f00000, 0x100000);

  close(fd);
}
```

Example

```
pthread_t thread;
static char *ptr;
int init_module (void) {
  rtl_printf ("<1>***I am In init_module***\n");
  ptr = (char *) mbuff_alloc (“MGJU”, 13);
  if (ptr != NULL)
    rtl_printf ("<1>***shared memory created successesully***\n");
  rtl_printf ("<1>***String- HELLO WORLD is Written to Shared memory***\n");
  memcpy (ptr, "HELLO WORLD", 13);
  return 0;
}
void cleanup_module (void) {
  rtl_printf ("<1>***I am In cleanup_module***\n");
  rtl_printf ("<1>***Shared Memory Is Deallocated***\n");
  mbuff_free (“MGJU”, ptr);
  rtl_printf ("<1>---------BYE----------\n");
}
```

```
Include……
volatile char *ptr;
int main (void) {
  /*  attaching  shared memory  in user space */
  ptr = (volatile char *) mbuff_alloc (“MGJU”, 13);
  printf ("***I am in user space***\n");
  printf ("***Data read from kerel*** %s\n", ptr);
  /* releasing the memory */
  printf ("\n***Shared Memory is dettached from user space***\n");
  mbuff_free (“MGJU”, (volatile char *) ptr);
  return 0;
}
```

Mutual exclusion

- Mutual exclusion refers to the concept of allowing only one task at a time (out of many) to read from or write to a shared resource.
- RTLinux supports the POSIX pthread_mutex family of functions

(include/rtl_mutex.h).

pthread_mutex_init(pthread_mutex_t *mutex,

const pthread_mutex attr_t *mutexattr);

pthread_mutex_lock(pthread_mutex_t *mutex)

pthread_mutex_unlock(pthread_mutex_t *mutex)

Semaphores

- Semaphores are counters for resources shared between threads.
- The basic operations on semaphores are: increment the counter atomically, and wait until the counter is non null and decrement it atomically.

```
#include < semaphore .h >

int sem_init ( sem_t *sem, int pshared, unsigned int val ) ;

int sem_wait ( sem_t *sem ) ;

int sem_post ( sem_t *sem ) ;

int sem_destroy ( sem_t *sem ) ;
```

Example

```
#define NTHREAD 3
static pthread_t threads[NTHREAD];
static sem_t sem;
static void *
start_routine (void *arg) {
  int ret;
  int taskno = (int) arg;
  rtl_printf ("<1>task %d;  waiting  on semaphore\n", taskno);
  ret = sem_wait (&sem);
  rtl_printf ("<1>task %d; entering critical section with  %d\n", taskno,
                        ret);
  if (ret < 0) {
      rtl_printf ("<1>task %d; errno=%d\n", taskno, errno);
    }
  ret = sem_post (&sem);
  rtl_printf ("<1>task %d; leaving the semaphore \n", taskno);
  return 0;
}
```

```
Int init_module (void) {
  int i;
  rtl_printf ("<1>***I AM IN INIT
MODULE***\n");
  sem_init (&sem, 1, 1);
  for (i = 0; i < NTHREAD; i++) {
      pthread_create (&threads[i], NULL,
start_routine, (void *) i);
    }
  return 0;
}
Void cleanup_module (void) {
  int i;
  rtl_printf ("<1>***I AM IN CLEAN UP
MODULE***\n");

for (i = 0; i < NTHREAD; i++) {

      pthread_cancel (threads[i]);
      pthread_join (threads[i], NULL);
    }
  sem_destroy (&sem);
  rtl_printf ("<1>*********BYE********\n");
}
```

Realtime device drivers

- An operating system must interface its peripheral devices to its Kernel software as well as to the user application software.
- This should be done in a modular and systematic way, such that all hardware "looks the same" to software applications. The software that takes care of this hardware-independent interfacing are *device drivers*.
- In the UNIX world, device drivers are visible through the

/dev/xyz "files" (where xyz stands for a particular device, such as, for example, hda for the first hard disk, ttyS0 for the first serial line, etc.).

Basic driver requirements

- Writing device drivers for an RTOS or an EOS is not so much different from writing them for a general-purpose OS.
- Basically, in an RTOS context, one should make sure that all timing delays in the drivers are both *short* and *deterministic*, and every DSR should be an appropriately prioritized thread or handler that waits on an event to become active.

COMEDI (Control and Measurement Device Interface)

- The Comedi project develops open-source drivers, tools, and libraries for data acquisition.
- **Comedi** is a collection of drivers for a variety of common data acquisition plug-in boards. The drivers are implemented as a core Linux kernel module providing common functionality and individual low-level driver modules.
- **Comedilib** is a user-space library that provides a developer-friendly interface to Comedi devices.
- **Kcomedilib** is a Linux kernel module (distributed with Comedi) that provides the same interface as Comedilib in kernel space, suitable for real-time tasks. It is effectively a "kernel library" for using Comedi from real-time tasks
- http://www.linux-usb-daq.co.uk/comedi/pdf/comedilib.pdf

Using Floating Point Operations

- By default RTL threads cannot use floating-point operations.
- This is because RTL threads have an integer context only.
- This has been done to facilitate fast context switches because switching FP context takes some time.
- To change default status of floating point operations the following function needs to be called:

  pthread_setfp_np(thread,flag)

- To enable set **flag to 1**, to disable **set flag to 0**.
- FP context is switched only if the new thread requires it.

RT Linux - dynamic memory management

- The use of kmalloc has been used for real time applications, but this call could block the kernel. So this can not be safely used from RT task.
- However, in versions of RTAI v1.3, the dynamic memory allocation module now allows memory allocation and deallocation calls to be used from real time tasks. The rt_mem_mgr package can be installed as a stand alone module or as a part of RTAI distribution.
- **The implementation of rt_mem_mgr is unclear in RT-Linux.**

- You can insert the module by entering:

  insmod <rtai>/rt_mem_mgr/rt_mem_mgr.o

  to allocate memory:

  addr = rt_malloc(size);

  And to de-allocate it:

  rt_free(addr);