6. Device control functions for any peripheral present into a system (computer), 7. Graphic editors for kernel configuration, 8. Soft real time scheduling. The following are three important classes of devices in embedded Linux.

Real-time Linux schedular has all functions as O (1) functions, which means that size of input to a function and function-run time are linearly correlated. The task schedular supports spin lock (Section 7.11), and it is a preemptive scheduler. Linux supports character and block devices. A device type can be a char device for example, a parallel port, LCD matrix or serial port or keypad or mice. The character access is byte-by-byte and analogous to the access from and to a printer device. System call functions are also like an IO device open ( ), close ( ), write ( ) and read ( ) functions (Section 8.6.1).

The Linux kernel maps a device to a node of a file system. The network device drivers for the network devices also support the address resolution protocol (ARP) and reverse address resolution support (RARP). Recall Section 4.9.4 for the description of Linux device drivers. The important aspects of Linux are explained next.

Linux has the following features useful for embedded system design.

1. Linux is multi-user OS and supports user groups ($2^{32}$).
2. Linux has root directory (represented by / sign) and all files are subdirectories to it. For example, a device module file in the subdirectory is /dev. User files directory is /user.
3. Linux has large number of editor, file, directory, IO commands. Each command can consist of a number of words, first word is command and remaining words are arguments. A word may be a composite word for several functions and actions.
4. Linux has number of interfaces for user. For example, X-Windows for GUI and csh (for C shell). [Shell runs a parent process. Shell is an interface for a user to enable entering the commands for the OS. Shell reads, interprets, checks for errors and executes the command(s). For executing a command, a child process is created. (this action is called fork). After execution, there is return to the parent process, which issues a prompt sign $. The shell waits for another command from user.]
5. Linux uses POSIX processes and threads (Sections 8.12). Linux header files Linux/ types.h and Linux/ shm.h, if included support the system programming for forking processes and shared memory functions in the kernel. For shared memory functions the POSIX map are used in Linux.
6. A process always creates as child process in the process that uses fork ( ) function. The child creates as the copy of the parent with a new process ID (Section 7.1). There can be $2^{30}$ process IDs. The fork () returns a different process structure *pid* for the parent and *pid* number of child becomes = 0. The child process is made to perform different functions than the parent by overloading function of the parent process using execv ( ) function. The function has two arguments, the *function_name* for the child function and *function_arguments* for the child functions. Each *process* has its own memory and cannot directly call another *process.* There is no lightweight process as in UNIX.
7. Linux has modules for the character, block, socket, network device drivers and others (Section 4.9.4) and Linux 2.6.x supports 4095 device-types. Each type of device can have $2^{30}$ device addresses. A *character device* (for example, parallel port, LCD matrix port, keypad or mice) receives or sends a sequential access byte stream. A block device (for example, file system or RAM dish) is a device that handles a block or part of a block of data. For example, 1 kB data handled at a time. Each block device receives or sends through a file system node (Section 8.6.2). A net device (for example, Telnet, FTP, TCP, IP or UDP protocol stacks networking device) is a device that handles network interface device (card or adapter) using a line protocols, for example, tty or PPP or SLIP. A *network interface* receives or sends packets using a protocol and sockets, and the kernel uses the modules related to packet transmission. An input device is a device that handles inputs from a device, for example, keyboard. An *input* device driver has functions for the standard input devices. The *media* device drivers have functions for the voice and video input devices. Examples are video-frame grabber device, teletext device, radio device (actually a streaming voice, music or speech device). A video device is a device that handles the frame buffer from the system to other systems as a char device does or UDP network packet-sending device

The video driver has functions for the standard video output devices. A sound device driver has functions for the standard audio devices. A sound device is a device that handles audio in standard format.

8. Linux supports a module initialization, handling of the errors, prevention of unauthorized port accesses, usage counts, root-level security and clean up. A module creates by compiling without main ( ). A module is an object file. For example, object module1.o creates from module1.c file by command $ gcc –c {flags} module1.c. The Linux OS supports registering of the driver configurations and functions. Table 10.8 gives these functions. Linux scheduler not only schedules the tasks and device-driving ISRs but all other device modules also.

A Linux kernel can insert a module by registering and removing it by de-registering. (Registering means that it is scheduled later when its turn comes. Deregistering means the module will be ignored. It is similar to task spawning and deleting.) MUCOS (Section 10.2) registers and deregisters the tasks and ISR interrupts the tasks (for example, by a software interrupt instruction) and passes messages to the tasks. VxWorks (Section 9.3) kernel registers and deregisters the tasks (pre-empt mode scheduling) as well as nested ISRs. Linux kernel provides registering and deregistering of the device-driver modules as well. Thus, the Linux kernel permits the scheduling of device drivers and modules. Therefore, the different tasks or programs can send the bytes concurrently or sequentially to a device through its driver registered at the kernel. Further, the Linux kernel enforces the use of sequential accesses, an to specific memory addresses only, by the registering and de-registering mechanism.

**Table 10.8** Registering and De-Registering and Related Functions of Linux Modules

| *Function* | *Action(s)* |
|---|---|
| init | The init_module( )is called before the module is inserted into the kernel. The function returns 0 if initialization succeeds and negative value if does not. The function registers a handler for something with the kernel. Alternatively it replaces one of the kernel functions by overloading. |
| insmod | Inserts module into the Linux kernel. The object file module1.o, inserts by command $ insmod module1.o. |
| rmmod | A module file module1.o is deleted from the kernel by command $ rmmod module1. |
| cleanup | A kernel-level void function, which performs the action on an rmmod call from the execution of the module. The cleanup_module( ) is called just before the module is removed. The cleanup_module() function negates whatever init_module( ) did and the module unloads safely. |
| register_capability | A kernel-level function for registering. |
| unregister_capability | A kernel-level function for deregistering. |
| register_symtab | A symbol table function support, which exists as an alternative to declaring functions and variables static. |

9. Linux header file Linux/time.h, if included supports the timing function in the kernel for scheduling. Linux header files Linux/delay.h and Linux/tqueue.h, if included support delay functions and task queues functions. The task queues supported are the timers, disk, immediate and scheduler.
10. Linux supports signals (Section 7.10) on an event. Linux header file Linux/signal.h is included which supports the signalling function. Linux supports multithreading (Section 7.2), semaphores, mutex. Linux header files Linux/pthread.h, Linux/ipc.h Linux/sem.h, Linux/msg.h are included to support

the POSIX (Sections 8.12) thread, IPC, semaphore (including mutex) and message queue functions, respectively (Sections 7.10 to 7.16). Table 10.9 gives the functions for the signal, multithreading and semaphore and message queue IPCs.

11. Linux 2.6.x all tasks are assigned static priorities between –19 to +20 (highest to lowest). Time slices are varied as per priority.
12. Real-time Linux 2.6.24—a latest release supports a new set of group-scheduling functions, which improves CPU load.
13. Linux 2.6.24 provides high resolution timers as wel as tick-less timers. (Tick-less mean does not result in clock-interrupts.)
14. Linux 2.6.24 supports functions to improve system performance by restricting the block device from taking larger CPU load.

Recently Wind River of VxWorks fame has optimized its Linux device software platform.

**Table 10.9** Linux Functions for the Signal, Multithreading and Semaphore and Message Queue Interprocess Communication

| *S.No* | *Feature* | *Description* |
|---|---|---|
| *1* | Thread properties | Threads of same process share the memory space and manage access permissions. The IPCs are used for synchronization objects (interprocess communication objects) and threads. |
| *2* | Signal functions | 1. struct sigaction signal_1; /* statement in C defines a structure signal_1. */<br>2. signal_1.sa_flags =0; /* Sets flags = 0*/<br>3. signal_1.sa_handler = handlingfunction_1; /* Defines signal handler as a user defined function handlefunction_1. */<br>4. sigemptyset (&signal_1.sa_mask); /* Defines signal as empty and sa_mask masks the execution of signal handler. */<br>5. sigaction (SIGINIT, &signal_1, 0); /* Initiates signal function handlingfunction_1() using the structure signal_1. */ |
| *3* | Thread functions | 1. struct pthread_t clientThd_1, clientThd_2, ServingThd; /* statement in C defines three structures for POSIX threads clientThd_1, clientThd_2, ServingThd. */<br>2. pthread_create (&clientThd_1, NULL, 0; thread_1, NULL) /* creates thread with structure clientThd_1 and calling function thread_1. NULL is the parameter passed. Similarly the threads clientThd_2, ServingThd and others can be created*/ The function returns an integer *createdstate* which is not 0 in case thread creation fails.<br>3. pthread_self () ; /* Returns (gets) ID of the function pthread_self calling thread (self) */<br>4. pthread_join (&clientThd_1, &threadNew) /* joins thread with structure clientThd_1 and threadNew. */ The function returns an integer *createdstate* which is not 0 in case thread creation fails.<br>5. pthread_exit ("text") /* exits the thread under execution and text displays on console on exit. */<br>6. sleep (*sleeptime*); /* The thread sleeps for a period = *sleeptime* nanoseconds. Other thread gets the CPU in sleep interval*/<br>7. pthread_kill (); /* Sends 'kill' signal to the thread */ |
| *4* | Semaphore functions | 1. struct sem_t sem1, sem2; /* statement in C defines three structures for semaphore structure (event control block), sem1 and sem2. */ |

(*Contd*)

| S.No. | Feature | Description |
|---|---|---|
| | | 2. sem_init (); The arguments are semaphore to be initiated and *options*. The function returns an integer *createdstate* which is not 0 in case semaphore initiation fails<br>3. sem_wait (&sem1); /* wait for the semaphore sem1 */<br>4. sem_post (&sem1); /* post the semaphore sem1 */<br>5. sem_destroy (&sem1); /* destroy the semaphore sem1 */ |
| *5* | Mutex functions | 1. struct pthread_mutex_t ms1, ms2; /* statement in C defines three structures for semaphore structure (event control block), ms1 and ms2. */<br>2. pthread_mutex_init (); Initiates a mutex. The arguments are semaphore to be initiated (ms1 or ms2 or other) and parameter *options*. The function returns an integer *createdstate* which is not 0 in case mutex initiation fails<br>3. pthread_mutex_lock (); The argument is mutex ms1 or ms2 or other*/<br>4. pthread_mutex_unlock (); The argument is mutex ms1 or ms2 or other*/<br>5. pthread_mutex_destroy (); The argument is mutex ms1 or ms2 or other*/ |
| *6* | Message queue functions | 1. pthread_mq_open ( ), mq_close ( ) and mq_unlink ( ) initialise, close and remove a named queue. unlink ( ) does not destroy the queue immediately but prevents the other tasks from using the queue. The queue will get destroyed only if the last task closes the queue. Destroy means to de-allocate the memory associated with queue ECB.<br>2. pthread_mq_setattr ( ) sets the attributes.<br>3. pthread_mq_lock ( ) and pthread_mq_unlock ( ) unlock and lock a queue.<br>4. pthread_mq_send ( ) and pthread_mq_receive ( ) to send and receive into a queue. mq_send four arguments are msgid (message queue ID), (void *) &qsendingdata (pointer to address of user data), sizeof (qsendingdata) and 0. mq_ receive five arguments are msgid (message queue ID), (void *) &qreceivedata (pointer to address of bufferdata), sizeof (qreceivingdata), 0 and 0.<br>5. pthread_mq_notify ( ) signals to a single waiting task that the message is now available. The notice is exclusive for a single task, which has been registered for a notification. (Registered means later on takes note of the pthread_mq_notify.)<br>6. The function pthread_mq_getattr ( ) retrieves the attribute of a POSIX queue. |
| *7* | Queues of functions | The queue of functions are equivalent to character devices, accessed via POSIX read/write/open/ioctl system calls. |
| *8* | Time functions | 1. nanosleep (sleeptime); /* Sleep for nanosecond period defined in argument sleeptime /*<br>2. clock_getthrtime ( ) ; /* getthrtime returns high resolution time (in ns) of the clock named clock. The clock is name of the clock (system clock) specified earlier. */<br>3. getthrtime ( ) ; /* getthrtime returns high resolution time (in ns) in a thread in which it is used as argument. */<br>4. clock_gettime ( ) ; /* gettime returns time of the clock named clock. */<br>5. clock_settime ( ) ; /* settime sets time of the clock named clock. */ |
| *9* | Shared memory functions | Used as an alternative to using the IPCs.<br>1. include <shm.h> /* Includes shared memory header*/.<br>2. include "common.h> /* Includes common memory file header*/. |

*(Contd)*

| S.No. | Feature | Description |
|---|---|---|
| | | 3. struc common_struct sharedblck; /* Specify a new structure sharedblck */.<br>4. int shmID = shmget ((key_t) num, sizeof (struc common_struct), 0666 I IPC_CREAT); /* shmget ( ) function arguments are (key_t) num (= 2377), common structure size, and a option for shared memory creation or use of IPC. Return ID of shared memory structure*/.<br>5. sharedMem1 = shmat (shmID, (void *)0, 0) /* Pointer to shared memory block sharedMem1 is found by function shmat. The arguments are ID of shared memory block, null pointer function and option 0. */.<br>6. int shmem_status = shmctl (shmID, IPC_RMID, 0); /* shmctl ( ) function arguments are The arguments are ID of shared memory block, IPC_RMID pointer and option 0. Return shared memory status shmem_status */<br>7. int shmemdetect_status = shmdt (sharedMem1); /* shmdt ( ) function argument is pointer to shared memory block sharedMem1. Returns –1 if not detected. */ |

## 10.3.2 RTLinux

An extension of Linux version in which earlier there was no real-time support is a POSIX hard real-time environment using a real-time core. The core is called RTLinuxFree and RTLinuxPro, freeware and commercial software respectively. V. Yodaiken developed RTLinux.

There are relatively simple modifications, which converts the non real time Linux kernel with round-robin scheduler into a hard real-time environment. The deterministic interrupt latency ISRs execute at RTLinux core and other in-deterministic processing tasks are transferred to Linux. The forwarded Linux functions are placed in FIFO with sharing of memory between RTLinux threads as highest priority and Linux functions running as low priority threads. Figure 10.3 shows RTLinux basic features.

Running the task in the following configuration gives hard real-time performance.

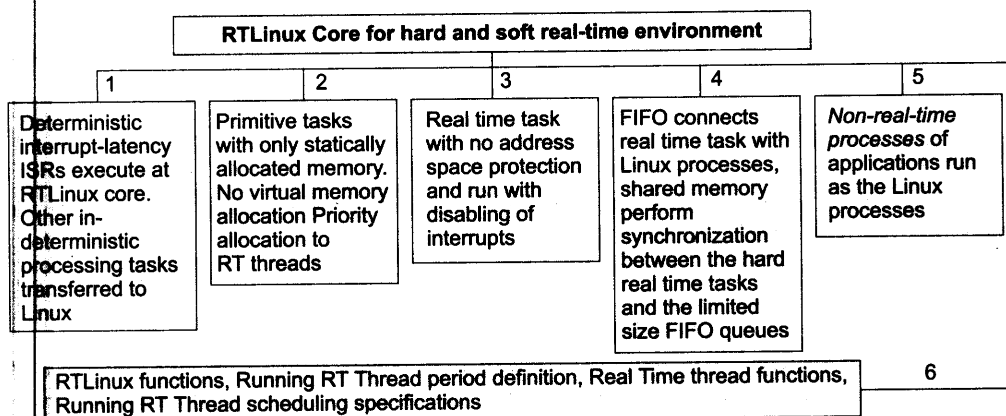



**Fig. 10.3** RTLinux basic features

- Run the primitive tasks with only statically allocated memory. The dynamic memory allocation or virtual memory allocation introduces unpredictable allocation and load timings.

- Run the real-time task with no address space protection. The memory address protection involves additional checks, which also introduce the unpredictable allocation and load timings.
- Run with disabling of interrupts so that other interrupts do not introduce the unpredictability.
- Run a simple fixed priority scheduler.
- When the FIFO connects real-time task with Linux processes, perform the synchronization between hard real-time tasks and limited-size FIFO-queues that is achieved through use of shared memory (not through IPCs).

*Soft real time-task* of applications can be configured to run differently. This is because the RTLinux allows flexibility in defining real-time task behaviour, synchronization and communication because the RTLinux kernel has been designed with modules, which can be replaced to make the behaviour flexible wherever possible.

*Non-real-time processes* of applications run as the Linux (before 2.6.x version arrived) processes.

RTLinux has the following features useful for real-time embedded system design.

1. Small footprint core. The core is like a virtual machine layer. The RTLinux core provides kernel as preempting kernel as well as non-preempting kernel.
2. RTLinux executive cannot be preempted.
3. Real-time tasks run in privileged mode and therefore directly access the hardware and do not use virtual memory. Real-time tasks are written as special Linux modules that can be dynamically loaded into memory.
4. Deterministic worst-case latency for number of processors, x86, ARM, MIPS tested for extended periods of extreme loads is very small.
5. CPU can be assigned to specific real-time threads, for example, during media processing. For a task, priority is introduced and set, and repetition rate and scheduling algorithm can be selected.
6. RTLinux gives high average case performance and the real-time threads meet the strict worst-case limited deadlines.
7. POSIX threads APIs has POSIX 1003.13 PSE51 specification. The latter is a minimal real-time system model specification. Linux or Unix functions are the lowest priority threads with POSIX IPCs using the shared memory and lock-free queues (queuing even when interrupts disabled) and shared memory.

***Programming with RTLinux*** Steps in running a program is to first boot RTLinux. Following examples gives a way to compile the module in RTLinux. It is similar to the one in Linux.

### Example 10.2

**1. include rtl.mk /* Include RTLinux make file. The rtl.mk file is an include file which contains all the flags needed to compile the code. */**
**2. all: module1.o /* Object file at module1.o */**
**3. clean: rm -f .o /* Remove using function rm object files inserted before this file */**
**4. module1.0: module1.c /* module1.0 is object file of source file module1.c */**
**5. $(cc) ${include} ${cflags} -c module1.c /* Compile, include, Cflags C module module1.c */**

**Now command $ compiles the files. Command $ rtlinux start module1 inserts (start) object binary file module1.o. Command $ dmesg displays the messages. Command $ rtlinux stop module1 removes (stops) object binary file module1.o.**

RTLinux supports a module as well as threads initialization, handling the errors, prevention of unauthorized port accesses, usage counts, root-level security and clean up. RTLinux scheduler not only schedules the

threads and modules. The RTLinux supports the registering of the modules, threads, configurations, functions, thread priority allocation and scheduling, Table 10.10 gives the registering, de-registering, priority and scheduling of real-time thread and FIFO-related functions in RTLinux.

**Table 10.10** Registering, De-Registering, Priority and Scheduling of Real-Time Thread and FIFO-Related Functions in RTLinux

| *Function* | *Action(s)* |
|---|---|
| init insmod, claenup, and rmmod | The init_module(), insmod_module( ), cleanup ()and rmmod_module() functions same as Linux |
| RT Linux functions | 1. rtl_hard_enable_irq ( );/* Enables hard real time interrupts */<br>2. rtl_hard_disable_irq ( );/* Gets current clock schedule */<br>3. rtlinux_sigaction ( ); /* to RTLinux signal handling function of the application */<br>4. rtl_getschedclcok ( );/* Gets current clock schedule */<br>5. rtl_request_irq ( ); /*to install real time interrupt handler */<br>6. rtl_restore_interrupts ( ); /* Restores the CPU interrupts state */<br>7. rtl_ stop_interrupts ( ); /* Stops the CPU interrupts */<br>8. rtl_printf ( ); /*to print text from real time thread */<br>9. rtl_no_interrupts ( ); /* /* No CPU interrupts permitted*/ |
| Thread creation and getting thread ID | Creation and getting ID is same as Linux threads pthread_create ( ) and pthread_self ( ). |
| Semaphore and mutex functions | Same as in Linux. |
| Thread wait | pthread_wait_np(); /*thread waits for other thread to complete*/. |
| Running RT thread period definition | pthread_make_periodic_np (pthread_self(), gethrtime(), 800000000); /* defines periodicity of thread with a thread self. First argument is reference to self thread (thread itself). Second argument is gethrtime() is a function to get thread time. Third argument is to define period of thread run = 800000000 ns. Any other period can be defined in third argument.*/ |
| RT thread deletion | pthread_delete_np (thread_1); /* Delete the thread thread_1*/ . |
| Thread priority | 1. struct sched_param thrd1, thrd2; /* Defines two structures for threads thrd1, thrd2 assignment parameters. */<br>2. thrd1.sched_priority = 1; /* Defines thrd1 scheduled priority parameter = 1 (=high). Similarly thrd2 priority can be defined */<br>3. pthread_setschedparam (pthread_self (), SCHED_FIFO, & thrd1;/* Function setschedparam arguments are self function, schedule defines as FIFO and thread structure thrd1. Similarly thrd2 arguments for Function setschedparam can be defined */ |
| Put into the FIFO from thread | thrd1_put(fid, &queuebuff, 1); ;/* Function thrd1_put arguments are fid (= an integer for FIFO descriptor ID, say, 1, 2, 3, ..), address of queue buffer queuebuff and an option = 1 */ |
| Real-time thread functions | 1. pthread_attr_setcpu_ np ( ); /* Set CPU pthread attributes. */<br>2. pthread_attr_getcpu_ np ( ); /* Get CPU pthread attributes. */<br>3. pthread_attr_setfp_ np ( ); /* Set CPU pthread floating point enable attributes.*/<br>4. pthread_ suspend_ np ( ); /* Suspends thread run.*/ |

(*Contd*)

| | |
|---|---|
| | 5. pthread_ wakeup_ np ( ); /* Wakes up the thread .*/<br>6. pthread_ wait_ np ( ); /* Wait for start (message or signal) the thread .*/<br>7. pthread_ setfp_ np ( ); /* Allows floating point arithmetic.*/<br>8. pthread_delete_ np ( ); /* Delete the thread.*/ |
| Real-time FIFO functions | 1. rtf_create ( ), rtf_create_rt_handler ( ), rtf_open ( ), rtf_close ( ) and rtf_unlink ( ) are the functions to createFIFO device, create real-time handler, initialize, close and remove a named RT FIFIO. unlink ( ) does not destroy the queue immediately but prevents the other tasks from using the queue. The queue will get destroyed only if the last task closes the queue. Destroy means to de-allocate the memory associated with queue ECB.<br>2. rtf_setattr ( ) sets the attributes.<br>3. rtf_lock ( ) and rtf_unlock ( ) lock and unlock a queue.<br>4. rtf_put (fid, &queuebuff, 1); ;/* Function rtf_put arguments are fid (= an integer for FIFO descriptor ID, say, 1, 2, 3, ..), address of queue buffer queuebuff and an option = 1 */.<br>5. rtf_get (fid, &queuebuff, 1); ;/* Function rtf_get arguments are fid (= an integer for FIFO descriptor ID, say, 1, 2, 3, ..), address of queue buffer queuebuff and an option = 1 */.<br>6. rtf_send ( ) and rtf_receive ( ) to send and receive into a queue. rtf_send four arguments are fid (FIFO device ID), (void *) &qsendingdata (pointer to address of user data), sizeof (qsendingdata) and 0. mq_ receive five arguments are msgid (message queue ID), (void *) &qreceivedata (pointer to address of bufferdata), sizeof (qreceivingdata), 0 and 0.<br>7. rtf_notify ( ) signals to a single waiting task that the message is now available. The notice is exclusive for a single task, which has been registered for a notification (registered means later on takes note of the rtf_notify).<br>8. rtf_getattr ( ) retrieves the attribute of a RT FIFO.<br>9. rtf_flush ( ) flushes the data of a RT FIFO.<br>10. rtf_allow_interrupts ; ( ) /* Controls real time interrupt handler */.<br>11. rtf_free_irq ( ); /*Frees real time interrupt handler */. |

IPC is performed by using a FIFO operation as follows. Real-time FIFOs use one source process (a real-time thread) and one end process (user space process). As one source process sends into the FIFO, another process at the other end of the FIFO receives. RTLinux FIFOs are character devices, (/dev/rtf*). Real-time threads use integers to refer to each FIFO. Integers 1, 2, 3 for FIFO 1, 2 and 3, respectively. There is a limit to the number of FIFOs (150). The RTF (RTLinux-FIFO) devices are rtf1 rtf2, rtf3 and so on. Linux character devices use create(), put(), get () and destroy() functions to device create, send from thread and get into process and device destroy. The RTF device uses rtf_create (), rtf_put(), rtf_get () and rtf_destroy() functions to RTF device create, send from thread and get into process and device destroy. Following is the code example.

### Example 10.3

This example shows how to create a thread and defines its real-time periodicity and use RT thread FIFO device to put into the buffer.

```
1. include rtl.h          /* Include RTLinux header file.*/
2. include pthread.h      /* Include POSIX thread header file.* /
3. include rtl_fifo.h     /* Include RTLinux FIFO header file.*/
4. include time.h         /* Include timer header file.* /
```

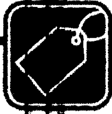
```
5. define fid 5 /* Define FIFO ID = 5 */
6. int open ("/dev/rtf5", O_WRONLY); /* Open FIFO device rtf5*/
7. sruct pthread_t clientThd_1 /* Create a client thread 1*/
8. int init_module (void) {
9. return pthread_create(&clientThd_1, NULL, thread_1, NULL); /* creates thread with structure
   clientThd_1 and calling function thread_1. Returns the thread ID return. */
10. }
11. void cleanup_module (void)
12. { pthread_delete_np (clientThd_1); /* Delete the thread function */
13. }
14. void * thread_1 (int fid) { /* Thread1 function */
15. static char queuebuff = 0; /* Define FIFO buffer address */
16. pthread_make_periodic_np (pthread_self(), gethrtime(), 800000000); /* Define thread period
    as 800000000 ns */
17. while (1)
18. {pthread_wait_np(); /* wait for RT thread */
19. queuebuff = (int) queuebuff ^0xff; /* Define new FIFO buffer address */
20. rtf_put(fid, & queuebuff, 1); /* Put into the queuebuff FIFO defined by fid */
21. } return 0; }
```

## Summary

The following is the summary of what we learnt in this chapter.

- Windows CE (WCE) is an operating system for systems, handheld and mobile devices, which have resource constraints of power, memory, touch screen or display screen size and processing speeds. It has extensions for pocket PCs and automotives.
- WCE is an open, scalable and small-footprint 32-bit OS.
- An enhancement of WCE is Windows CE.NET. [.NET framework provides for compiling the managed code. Managed code is one that is compiled in CIL (common intermediate language). It gives platform-independent CPU neutral compilation as the byte codes.
- WCE provides a Windows platform for the systems and it gives a user to feel, look and interact with the system using GUIs in a manner similar to a PC running on Windows.
- 80x86 or SuperH or ARM, SH4 and MIPS-based processor architectures are supported by WCE and the WCE fine tunes the processor performance.
- There is *componentization* of OS. OS has two layers: one is the source code and the is the other is the shared code with the devices manufacturer.
- A Windows-based application program is written to respond or activate or changes from the current state on pushing off notification(s) from the OS. A notification occurs on an event. The notification sends the *message* to the Windows application program. Messages are placed in queue for the Windows of the application program.
- Win32 API subset is used for GUIs programming.
- WCE supports the ISRs which pass the messages to ISTs, which run as lower-priority threads than the ISRs and ISTs run as priority queue of threads.
- Windows uses Handle in many procedures (functions). WCE does not support inheritance of Handles.
- WCE thread assigned priorities to each one among the eight levels. System-level threads and device drivers (ISTs) use the upper 248 levels of priorities.

- Windows CE 6.0 supports $2^{16}$ number of processes (earlier 32) with each process hairing a virtual memory limit of 2 GB (earlier 32 MB) and up to lower 2 GB (earlier 32 MB). There is also upper 2 an GB (earlier 32 MB) on the kernel VM space.
- Windows Mobile 6 supports Offfice Mobile 2007, smartphone with touch screen, improved Bluetooth stack, VoIP with AEC support to encryption of data stored in external removable storage cards, uses smartfilter for fast files, e-mail, contacts and songs search and can be set as modem for laptop.
- WCE provides for system memory between 1 MB and 64 MB and OS needs minimum 512 kB of memory and 4 kB of RAM. WCE also provides for managing the low memory conditions.
- WCE considers the RAM in two sections: *program memory* (called system heap) and *object store*.
- WCE has file systems and databases. The database has a series of un-lockable records with saving of a property(ies) and data together in a record. No record contains another record within it. Each record has four-level indices for sorting.
- WCE is a multitasking multithreaded program. There is at least one thread which is created per process. The basic unit of execution under control of the kernel is the thread. Each thread has a separate context (for CPU register including PC and stack pointer) and stack for saving the context when thread blocks and for retrieving on activating the thread.
- WCE provides for exception handling signals, notification signals, event functions (for threads of a process), semaphores, message queues, wait single object, multiple object and multiple message object functions.
- WCE provisions for GUIs based on Windows, menus, dialog boxes, radio and check buttons.
- Subset of WIn32 APIs in WCE provision for inputs from keys, touch screen or mouse, communication with serial port, Bluetooth, IrDA, WiFi, networking, device-to-device socket and communication functions.
- OSEK is a structured and modular software implementation based on standardized interfaces and protocols for the automobile distributed ECUs (electronic control units).
- OSEK/VDX specifies three standards, which give the portability and extendibility features, and thus the reusability of the existing software. OSEK specified three standards: one for embedded OS using *MODISTARC* (methods and tools for the validation of OSEK/VDX-based distributed architectures and for conformance testing of the OSEK/VDX implementations). The second is for communication and the third is for network management.
- OSEK uses abstract interfaces and implementation can be as per hardware and network. OSEK has standardized interfacing features for control units with different architectural designs. OSEK has an efficient design of architecture. The functionalities in the OSEK standard shall be configurable and scalable, to enable optimal adjustment of the architecture to the application in question.
- OSEK defines four types of classes BCC1, ECC1, BCCC2, ECC2COM and conformance classes CCCA and CCCB for internal communication. It specifies that there should be no creation and deletion of tasks during run. Task Priority must be defined and task activates only once in the codes. There must not be use of message queues, and semaphores must be used as flag only.
- Linux uses POSIX processes, threads, shared memory, POSIX map and POSIX queues.
- Linux has a number of interfaces for the user. For example, Graphic editors for configuring the kernel, X-Windows for GUI and csh (for C shell). Linux has a number of characters, blocks and network interface devices and has device drivers for the user-programming environment.
- Real-time Linux besides support to the module initialization, handling the errors, prevention of unauthorized port accesses, usage counts, root-level security, insert, remove and clean-up functions, also supports preemptive scheduling. Latest version of real-time Linux is Linux 2.6.24.
- RTLinux has real-time thread wait, thread period definition, thread deletion, priority assignment and FIFO device functions. It provides for running of real-time tasks by RTLinux layer and no deterministic non-real-time tasks by Linux. A FIFO connects real-time tasks with Linux processes, performs the synchronization between hard real-time tasks and limited-size FIFO queues through the use of shared memory (not through IPCs).
- RTLinux provides hard real functionalities in a separate layer, which runs the primitive tasks (real-time threads) with only statically allocated memory, no dynamic memory allocation, no virtual memory allocation, no address space protection, runs with disabling of interrupts, runs a simple fixed priority scheduler.
- RTLinux has separate functions. rtl_hard_enable_irq ( ); rtl_hard_disable_irq ( );rtlinux_sigaction ( ); rtl_getschedclcok ( ); rtl_request_irq ( ); rtl_restore_interrupts ( ); rtl_ stop_interrupts ( ); rtl_printf ( ); and rtl_no_interrupts ( ) and has real-time FIFO functions.



## Keywords and their Definitions

| | | |
|---|---|---|
| *Bitmap* | : | A bitmap is a graphical object which can be drawn on screen and is used for creating, retrieving images, manipulating and drawing at the device context. |
| *Block device* | : | A device which is accessed by a file system (disk) like commands and in which a block is accessed at an instant. |
| *Button* | : | A button is a window wherein the bringing the mouse or stylus near it (in-focus) or taking it away (out-of-focus) or clicking or taping over it on the screen initiates a notification from the OS, which then notifies to an API for running. |
| *Character device* | : | A device which accesses byte-by-byte analogous to the access from and to a console or keyboard or printer device using byte streams. |
| *Child process* | : | A process created in Linux by fork ( ) function, which sends new process ID to parent and makes self ID = 0, and which is made to perform different functions than the parent by overloading function of the parent process using execv ( ) function. |
| *Console* | : | A video terminal or touch screen or display object for GUI and for displaying output of the programs in a computer. |
| *Control* | : | An object for controlling program flow, for example, a command, menu or toolbar bar object or a date and time picker control object or calendar picker control or edit control to auto-capitalize the first character of a word when keying in and virtual keyboard or organizer (e.g., task-to-do). |
| *Componentization* | : | Provisioning of shared source and source code accesses provided by Microsoft such that there are two software layers. One sublayer consists of Microsoft-developed source codes of the WCE kernel and is shared with the system or the device manufacturer. Then the manufacturer adds the remaining part of the kernel according to the system hardware. The remaining part is the hardware abstraction layer. |
| *Device context* | : | A device context specifies the application Window, which sends the pixels to the screen, which is a tool, which Windows use in managing the access to the display and printer, which has two attributes of device context colours for background and foreground, has font of the displayed text, text alignment attributes left, right, top, centre, bottom, no update and update of current point and baseline alignment. |
| *DLL* | : | Dynamic link library is file, which is linked at run time of .exe file and which loads on request from the .exe file or another DLL. |
| *ECU* | : | An electronic control unit for controlling the unit and its communication with other ECUs in an automobile or other systems. |
| *Exception* | : | A signal from API for initiating a Window notification when an exception condition is arrived at run time. |
| *Execute-in-place file* | : | A file in ROM for execution that cannot be opened and read by standard file functions for *open* and *read*. |
| *FIFO* | : | A device which creates like a thread or file and which is sent. |
| *GUIs* | : | Graphic user interfaces for facilitating interaction and inputs from the user after graphic screen displays of menus, buttons, dialog boxes, text fields, labels, check-box and radio buttons and others. |
| *Handle* | : | Windows uses Handle in many procedures (functions). The Handle provides reference to an interface, for example for a Window, file or thread or port. An example is INSTANCE, a Window object for use as Handle. An interface is an unimplemented |

procedure (function or method) the codes for which are defined in the class which uses that interface. Handle is also used as a pointer, called *option pointer*.

***IPC object*** : An object in a multitasking or multithreading system for communication events, semaphores including mutex semaphores and messages into queues without using shared memory and which provides for threads synchronization.

***IST*** : Interrupt service thread is a thread which is put in the priority queue of threads waiting for execution and which is a slow-level interrupt service thread of a fast-level ISR, which posts the message(s) to that.

***Linux*** : A freeware OS named Linux is after Linus Torvalds, father of the Linux OS.

***Managed code*** : The code that is compiled in CIL . It gives platform-independent CPU neutral compilation as the byte codes. A run-time environment converts the byte code instructions into the native machine and platform instruction.

***Network interface*** : An interface device, which accesses using a network protocol such as Telnet, TCP, UDP, SLIP, PPP, SMTP or Bluetooth.

***Notification*** : The OS monitors all input sources (e.g., stylus tap, virtual (on-touch screen) or physical key press). The OS notifies and sends notification when a key has been pressed or a button has been clicked or command has been received for redrawing the Windows screen. A notification as a Windows-based application program is written to respond or activate or changes from the current state on pushing of notification(s) from the OS. A notification occurs on an event. The notification sends the *message* to the Windows application program. Messages are placed in queue for the Windows of the application program.

***Object store*** : A virtual RAM disk is a permanent store for storing files, registry, PIM, and WCE databases and is protected from power turning off. Individual file can use up to 32 MB in case of RAM as *object store*. If object is not found in RAM, it is loaded from the *object store*.

***Option pointer*** : A pointer which points to a pointer of one of the several sets of the codes, which run on selecting the option.

***OSEK*** : OSEK/VDX is a body for defining and specifying three standards for an embedded OS, communication and network management for automotive applications and control of ECUs.

***Pthread*** : POSIX thread, a thread for which POSIX standard functions are used to create, send, receive, suspend and which uses POSIX IPC semaphores, mutex, message queues and FIFOs.

***PocketPC*** : Handheld PC for home as well as office systems and for cellular networks connectivity using Windows CE extension, Windows Mobile, for applications such as Outlook, Explorer, pocket versions of Word, Excel and pocket Power Point, slide shows, mailing, internet and office applications.

***Power manager*** : Software to reduce the power dissipation by reducing clock speed or running wait or stop instruction or optimizing use of caches or stopping screen or reduced intensity displays after limited wait for user input.

***Real time core*** : A layer of functions for real time threads which has higher priority than other kernel functions.

***Registry*** : Registry API for system database using standard file registry functions: RegCreateKeyEx, RegOpenKeyEx, RegSetValueEx, RegQueryValueEx, RegDeleteKeyEx, RegDeleteValueEx and RegCloseKey.

***Signal*** : An interrupt or notification object, which initiates a handler function (interrupt service routine), provided signal is not masked and interrupts are enabled.

***Socket*** : An API at the streaming sockets or datagram to send and receive between two specific addresses at two APIs at different devices.

***Stylus*** : A writing pencil-core-shaped object for a user of device or system as an alternative to the mice and keyboard. The user touches the stylus tip at the displayed menu or displayed keypad on the screen to enter the commands or text, respectively.

***System heap*** : The system APIs and OS in the *program memory* area of memory.

***Touch screen*** : A device for screen displays as well as for accepting inputs through a stylus.

***VUIs*** : Voice user interfaces which facilitate interaction and command inputs using stored voice or tunes, and voice command inputs from the user.

***Win32 APIs*** : APIs for 32-bit programming using Window classes, objects, controls and Handles and for managing, displaying and drawing the Windows for the user APIs.

***Window*** : An object INSTANCE, which defines a Window (object) to provide Handle(s) for the GUIs and APIs for running the application codes. The object Window has basic coordinates *x* and *y*, and *z*-parameters, specification for visibility (show or hide or no activate), specification for parent–child hierarchy and procedures to share the attributes and to respond the requests and all notifications sent to the Windows.

***Windows CE*** : A Win32 API subset-based OS for handheld computers and mobile systems, developed by Microsoft that provides a programming environment using a subset of Win32 APIs, Visual C++, Visual Basic, and that enables a user to feel, look and interact with the system using GUIs in a manner similar to a PC running on Windows.

***Windows CE.NET*** : An enhancement of Windows CE deploying .NET framework that provides for compiling the managed code.

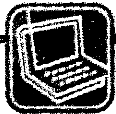
***Window Mobile*** : OS with extensions for Offfice Mobile 2007, smartphone with touch screen, improved Bluetooth stack, VoIP with AEC, support to encryption of data stored in external removable storage cards, uses smartfilter for fast files, e-mail, contacts and songs search, can be set as modem for laptop.

***WinSock*** : Streaming or datagram socket APIs in Windows, which has a feature of accessing personal area and network resources and that does not depend on platform and implementation of socket functions.

## Review Questions

1. Describe the features of Windows CE. What is the advantage in using .NET framework with Windows CE? Why does the Windows CE have low interrupt latencies?
2. Describe the additional features in Windows CE 6.0 and Windows CE extensions to Pocket PC, Windows Mobile 6 and Windows Automobile 5.0.
3. What are the differences in programming with Windows CE with respect to Windows? Explain meaning of Handle and Handle inheritance. What is the advantage of withdrawing reducing Handle inheritance in Windows CE?
4. What do you mean by Windows and relationship between the Windows? List the functions for management of Windows.
5. Describe memory management. Explain the similarity in file management and device management functions.
6. Describe the properties and Windows CE functions for the databases.
7. Windows CE and Linux are multitasking-multithreaded OSes, and MUCOS and VxWorks are multitasking OS. Explain the difference between two concepts. What is the basic unit of computations in each of the OSes? Describe the properties and Windows CE functions for threads and processes.

8. Explain the Windows CE exceptions, notifications and interprocess communication objects and their synchronization.
9. How do the inputs from keys, touch screen or mouse handled in Windows CE?
10. Describe Windows CE serial communication functions. Describe WNet API network connection functions.
11. List WinSock API subset in Windows CE for device-to-device socket-communication functions. How do communication functions for socket differ from the serial port communication?
12. How is Win32 API used for development of Windows for the GUIs in an application?
13. What are the embedded softwares in automotive system that need special features, standards and specifications in its operating system?
14. What is OSEK/VDX? What are three standards specified for an embedded operating system OS by OSEK/VDX?
15. List the OSEK basic features. How does the OSEK provide for functional extendibility and portability?
16. What do you mean by classes BCC1, ECC1, BCCC2 and ECC2? Why are the message queues not used, semaphores used as flags only and tasks are created and defined priorities at the beginning only in OSEK standard?
17. (a) List the features of real-time Linux. (b) Describe the functions for registering and de-registering related functions of Linux modules. (c) How does a child process creates in Linux by fork ( ) function? (d) What are the features added in Linux 2.6.x, which enables real-time system design? (e) What are the features added in Linux 2.6.24?
18. List the Linux functions for the signal, multithreading and semaphore and message queue for IPC.
19. How does RTLinux add a new core for real-time-tasks? How does hard real-time task executes in RTLinux? How does the shared memory help as a fast alternative to the use of IPC?
20. What are new functions added in RTLinux? Describe registering, de-registering, priority and scheduling of real-time thread and FIFO-related functions in RTLinux.



## Practice Exercises

21. Write the codes in Win32 API subset of Windows CE for displaying a contact name, telephone number, address and e-mailID at the Window after studying Listing 1-3 in Douglas Boling *Programming Microsoft WINDOWS CE.NET*, Microsoft, USA, 2003.
22. Write the codes for getting messages on Windows using touch screen after studying Listing 3-2 in Douglas Boling "*Programming Microsoft WINDOWS CE.NET*", Microsoft, USA, 2003.
23. Write the codes for viewing on a Window a file after studying Listing 8-1 in Douglas Boling *Programming Microsoft WINDOWS CE.NET*, Microsoft, USA, 2003.
24. Write the codes for querying a database in Windows CE.
25. Write the codes for creating two threads with highest and lowest priority, respectively, in Windows CE.
26. Write the codes for sending and receiving bytes between two Windows CE threads using message queues.
27. Write the codes for creating serial port and for sending and receiving bytes in Windows CE.
28. Write the codes for setting user notification and for acknowledging notification in Windows CE.
29. Write the codes for sending and receiving data between the sockets after creating sockets using SOCKET in Windows CE.
30. List the examples of using semaphore flags in an automobile.
31. Write the codes using fork ( ) for creating a child process in Linux.
32. Write the codes for creating the threads for sending and receiving PIM (personal information manager) data. PIM includes data of the contacts, calendar and task-to-do. A contact includes name, address, e-mail ID, phone numbers of home, office and mobile. The data are sent to another thread using synchronization by a POSIX semaphore.
33. Write the codes for creating the threads for sending and receiving Strings data through POSIX message queues in Linux.
34. Write the codes for displaying two texts alternately from two threads using RTLinux.
35. Write the codes for displaying two texts alternately every 8 s from two threads using RTLinux.
36. Write the codes for sending a byte stream into a FIFO in RTLinux.

# Design Examples and Case Studies of Program Modeling and Programming with RTOS-1

# 11

Recall

We have learnt in chapters 1 and 6, and in an online chapter on 'Software Engineering Approach for Embedded Systems Design' that the following steps are needed as per software engineering and UML modeling approaches.

- First, study the *requirements* and be clear of the required purpose, inputs, signals, events, notifications, outputs, functions of the system, design metrics, and test and validation conditions.
- Then make the *specifications*, in terms of users, objects, sequences, activity, state and class diagrams for abstracting the component, behavioral and events, and create description in terms of signals, states, and state machine transitions for each task.
- Next, define the *system architecture* for hardware and software, extra functionalities and related systems. The architecture specifies the appropriate decomposition of software into modules, components, appropriate protection strategies, and mapping of software.
- After defining the design, *implement and test* each component.
- Finally, *integrate* components in the *system*.

LEARNING OBJECTIVES

*We will learn embedded systems design from the three case studies discussed in this chapter and four in the next chapter. The first case study is of an **automatic chocolate-vending machine** (ACVM) which was introduced earlier in Section 1.10.2. The second study is of a digital camera, introduced earlier in Section 1.10.4. The third study is of a TCP/IP stack, which was introduced earlier in Section 3.11.*

*The objectives of these case studies are as follows:*

1. *To learn how the **requirements** are studied and **specifications** of a system are listed.*
2. *To learn how **UML modeling** is used to model the design of the system.*
3. *To learn to define **hardware architecture** using microprocessor or microcontroller or ASIPs or DSPs, and devices.*
4. *To learn how to define the **software architecture** for software, extrafunctionalities and related systems and define the decomposition of software into modules, components, appropriate protection strategies, and mapping of software.*
5. *To learn coding for design **implementation** using MUCOS and VxWorks RTOSes, and also the use of IPCs for task synchronization and concurrent processing.*

## 11.1 CASE STUDY OF EMBEDDED SYSTEM DESIGN AND CODING FOR AN AUTOMATIC CHOCOLATE VENDING MACHINE (ACVM) USING MUCOS RTOS

ACVM was introduced earlier in Section 1.10.2. It listed ACVM functions, hardware and software units. Figure 1.12 showed a diagrammatic representation of ACVM.

Sections 11.1.1 to 11.1.6 give the design steps of an ACVM. Section 9.2 described MUCOS RTOS. It has a portable, ROMable, scalable, preemptive, real time and multitasking kernel. Section 11.1.7 describes coding for ACVM using MUCOS RTOS (Section 9.2) programming environment.

### 11.1.1 Requirements

The requirements of the machine can be understood through a requirement table given in Table 11.1.

### 11.1.2 Specifications

The ACVM specifications in brief are as follows:

1. It has an *alphanumeric* keypad on the top of the machine. That enables a child to interact with it when buying a chocolate. The owner can also command and interact with the machine.

## Table 11.1 Requirements of an ACVM

| *Requirement* | *Description* |
|---|---|
| Purpose | To sell chocolate through an ACVM from which children can automatically purchase the chocolates. The payment is by inserting the coins of appropriate amount into a coin slot. [Adults are also welcome to use the machine!] |
| Inputs | 1. Coins of different denominations through a coin slot.<br>2. User commands. |
| Signals, events and notifications | 1. A mechanical system directs each coin to its appropriate port—Port_1, Port_2 or Port_5. Each port generates an interrupt on receiving a coin at input. Each port-interrupt starts an ISR, which increases value of *amount-collected* by 1 or 2 or 5 and posts an IPC to a waiting task.<br>2. Each selected menu choice sends a notification to the system. |
| Outputs | 1. Chocolate, and a signal to the system that subtracts the cost from the value of *amount-collected*.<br>2. Display of the menus for GUIs, time and date, advertisements, and welcome messages. |
| Functions of the system | A child sends commands to the system using a GUI (graphic user interface). The GUI consists of the LCD display and keypad units. A touchscreen is another alternative for LCD and keypad. The child inserts the coins for the cost of chocolate and the machine delivers the chocolate. If the coins are not inserted as per the cost of chocolate in reasonable times then all coins are refunded. If the coins inserted are of more amount than the cost of chocolate, the excess amount is refunded along with the chocolate. The coins for the chocolates purchased collect inside the machine in a collector channel, so that the owner can get the money through appropriate commands using a GUI. USB wireless modem enables communication to ACVM owner. |
| Design metrics | 1. *Power Dissipation*: As required by mechanical units, display units and computer system<br>2. *Performance*: One chocolate in two minutes and 256 chocolates before next filling of chocolates into the machine (assumed)<br>3. *Process Deadlines*: Machine waits for a maximum of 30 s for the coins and the machine should deliver the chocolate within 60 s.<br>4. *User Interfaces*: Graphic at LCD or touchscreen display on LCD and commands by children or machine owner through fingers on keypad or touch screen<br>5. *Engineering Cost*: US$ 50000 (assumed)<br>6. *Manufacturing Cost*: US$ 1000 (assumed) |
| Test and validation conditions | 1. All user commands must function correctly.<br>2. All graphic displays and menus should appear as per the program.<br>3. Each task should be tested with test inputs.<br>4. Tested for 60 users per hour. |

2. It has a *three-line* LCD display unit on the top of the machine. It displays menus, text entered into the ACVM, pictograms, and welcome, thank you and other messages. It enables a child as well as the ACVM owner to graphically interact with the machine. It also displays the time and date. [For graphic user interactions (GUIs), the keypad and LCD display units or touchscreen are basic units.]
3. It has a *coin-insertion slot* so that the child can insert the coins to buy a chocolate of his/her choice.
4. It has a *delivery slot* so that the child can collect the chocolate, and coins if refunded.
5. It has an *Internet connection port* so that the owner can know the status of the ACVM sales from a remote location.

***Specifications of the required functions in detail are as follows:***

1. The ACVM displays a pictogram for the chocolate vending company. It displays the GUI. The ACVM can also collect contact and birthday information. It can answer to FAQs (frequently asked questions).

It displays a welcome message in idle state. It also displays the time and date continuously at the right bottom corner of the display screen. It can also intermittently display advertisements or important information during the idle state.
2. When the first coin is inserted, a timer starts simultaneously. The child is expected to insert all the required coins in 30s.
3. After 30s the ACVM will display a query in case the child does not insert sufficient coins inside it. If query is not answered, the coins are refunded.
4. If within a specific time sufficient coins are collected, it displays a message, 'Thanks, wait for a few moments please!', delivers the chocolate through the delivery slot, and displays the message, 'Collect the chocolate and visit us again, please!'

Figure 11.1(a) shows the basic system of an 'Automatic Chocolate-Vending Machine System' (ACVM) machine. Figure 11.1(b) shows ports at the ACVM. The ports receive the inputs and generate events or notifications for the outputs—chocolate or display messages or coins.

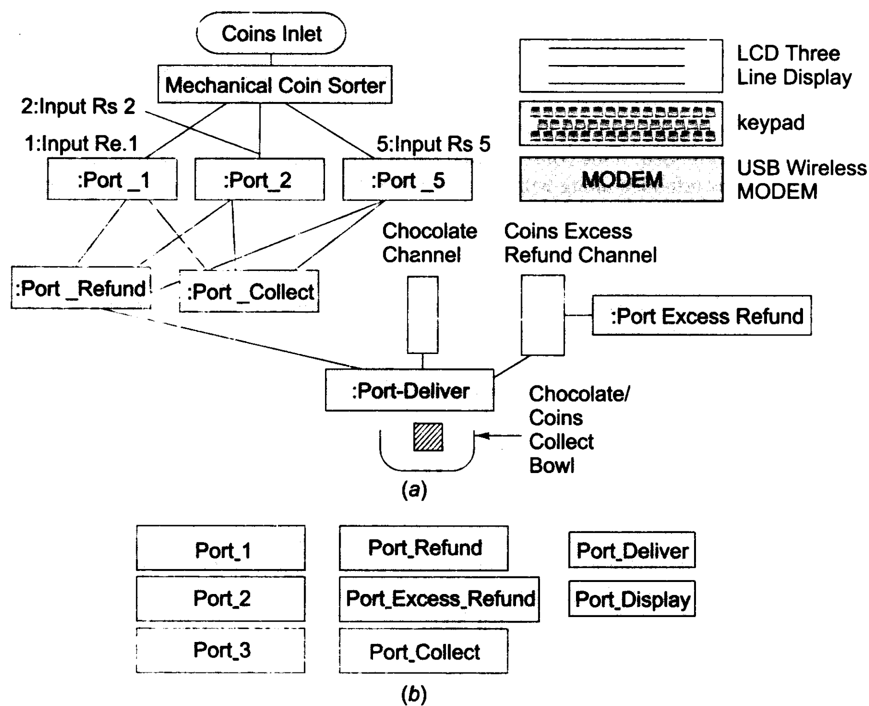



Fig. 11.1 (a) Basic System ACVM (b) Ports of the ACVM

***System specifications in detail are as follows:***

1. There is a slot into which a *child* (buyer) inserts the coins for buying a chocolate. Suppose the chocolate costs Rs 8. [*Children wish that chocolates should be so cheap*!] A coin can be in one of three possible denominations: Rs 1, 2 and 5. Whenever a coin is inserted, a mechanical system directs each coin of value Rs 1 or 2 or 5 to Port_1, Port_2 or Port_5, respectively. Each time a port receives a coin, it generates an interrupt, which posts a signal or semaphore to the corresponding task Task_ReadPorts for reading the coin value at the ports to increase the value of a parameter *amount_collected*.

2. The machine should have an LCD for dot matrix display or a touchscreen as 'User Interface'. Let the interface port be called Port_Display. It displays the message strings in three lines with the right-hand last line corner for display of *time* and *date*, and top line left-hand corner for pictogram. The pictogram displays the company emblem.
3. ACVM has a bowl from where the buyer collects the chocolate through a port for delivery. Let this port be called Port_Deliver. Port_Deliver connects to a chocolate channel. Whenever the channel is left with only a few chocolates, the owner of the machine fills chocolates into the channel. The buyer also collects the full refund or excess amount refund at the bowl through the ports Port_Refund (in the case of short amount) and Port_ExcessRefund (in the case of excess amount) respectively. All ports, Port_Deliver, Port_Refund and Port_ExcessRefund, communicate with Port_Collect by inter-process communication (IPC). Port_Collect is a common mechanical interface to Port_1, Port_2 and Port_5. Port_Deliver is a common mechanical interface to the bowl.
4. It should also be possible to reprogram the codes and relocation of the codes in the system ROM or flash or EPROM whenever the following happens: (i) the price of chocolate increases, (ii) the message lines or menus or advertisement or GUI or graphics need to be changed, or (iii) machine features change.
5. An RTOS schedules the buying tasks from start to finish. Let MUCOS be the RTOS used in the ACVM.

### 11.1.3 Specifications Modeling Using UML

Section 6.5 described that UML is a powerful modeling language for (i) software visualizing, (ii) data design(s), (iii) algorithms design, (iv) software design(s), (v) software specifications, and (vi) software development process. The ACVM specification can be modeled using UML.

***Class Diagram*** A class diagram shows how the classes and objects of a class relate, and also hierarchical associations and object interaction between the classes and objects. Rectangular boxes show the classes, and arrows with unfilled triangles at the end show the class hierarchy. Classes in the hierarchy can be joined using a line. Start and end numbers on a line shows how the objects of a class associate with objects of another [Table 6.3].

An ACVM system can be modeled by three class diagrams of abstract classes—ACVM_Devices, ACVM_Output_Ports and ACVM_Tasks. The tasks are the processes or threads that are scheduled by an operating system. Figure 11.2 shows three class diagrams of ACVM_Devices, ACVM_Output_Ports, and ACVM_Tasks. These demonstrate how the class diagrams are shown using UML.

1. *ACVM_Devices* is an abstract class from which the number of extended classes is derived for the devices to handle ACVM mechanism. The devices are keypad, display device, wireless USB modem and coins-input device. Therefore, abstract class *ACVM_Devices* has four extended classes—User_Keypad_Input, Display_Output, Coins_Input and Wireless_USB_Modem.
2. *ACVM_*Output*_Ports* is an abstract class from which the number of extended classes is derived for handling output ports at ACVM.
3. *ACVM_Tasks* extends the two classes, ACVM_System_Tasks and ACVM_System_ISRs. ACVM_System_Tasks is an abstract class from which we assume that *n* extended classes *Task1-Taskn* are derived. *Task1* to *Taskn* are the *n* tasks (processes or threads) at the ACVM. ACVM_System_ISR is an abstract class from which we assume that *m* classes are extended. ISR_Task1_ISR_TaskM are extended classes for ISR handling tasks at ACVM. [ISR_Task1 to ISR_TaskM are the *m* ISRs at ACVM. An ISR_Task initiates and runs on a signal or interrupt or event or notification or exception. A ISR_Task differs from a Task in the sense that ISR_Task has higher priority than the Task and an ISR_Task can only post the IPC object(s) (for example, signal, event semaphores, mailbox message

for notification, message-queue message) to a task(s) [or to an interrupt service thread in Windows CE], while the Task can wait for the IPC(s) as well as post IPC(s) to other tasks. The Task can also run as per preemptive or time slicing or any other scheduling [Section 7.6.2]. MUCOS environment schedules a task in preemptive mode.

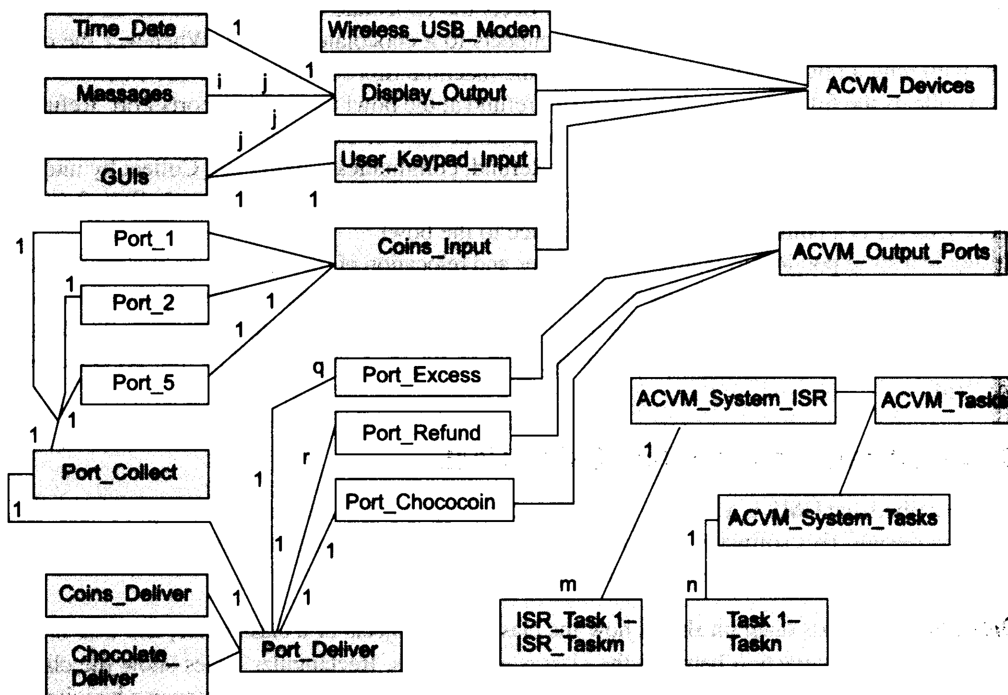



**Fig. 11.2** Class diagrams of ACVM devices, ports, and tasks

***Class*** Figure 11.3 shows examples of two classes, Display_Output and User_Keypad_Input, to demonstrate how the classes are shown using UML.

The Display_Output class has the following fields for the following: (i) *picture* for a pictogram, (ii) three strings, *Str1*, *Str2*, *Str3*, for text lines 1, 2 and 3 respectively, (iii) *menuChoiceOffered* string for the choice offered at an instant to graphic user, (iv) *time* that is an array of four integers for the current time in hour and minutes, (v) *amFlag* (= 1 when time is AM else 0), (vi) *pmFlag* (= 1 when *amFlag* = 0), and (vii) *date* is an array of four integers, for two digits each for month, day and year.

Display_Output has four methods (functions in C): displayPicture for showing an emblem or other picture, displayMessage ( ) for showing a message, displayTimeDate ( ) for showing the time and date, displayMenu ( ) for showing a menu and displayAdv ( ) for showing an advertisement.

The User_Keypad_Input class has the following fields: (i) inputline for the characters received for input line, for example, text during a child keying in the ID or name, address, date of birth, (ii) advertisement: array [ ] of inputChar (iii) inputChar: character, (iv) eight flags— choiceAcceptanceFlag, upKey, downKey, leftKey, rightKey, enterKey, YKey or NKey for notifying the acceptance of the selected choice or keying of up or down or left or right or enter or Y or N key, respectively. YKey notifies yes and NKey, no.

User_Keypad_Input has three methods: getFlag ( ), getString ( ) and getChar ( ). These are for getting a Boolean or string or character from the keypad.

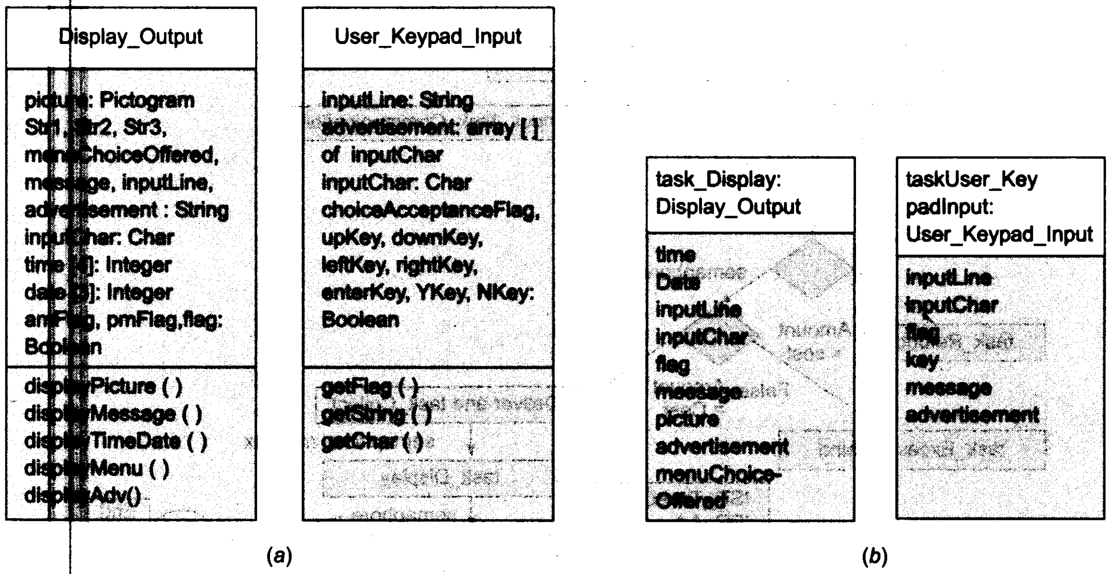



Fig. 11.3 (*a*) *Classes* Display_Output and User_Keypad_Input (*b*) *Objects* Task_Display and TaskUser_Keypad Input

***Object*** Figure 11.3(b) shows examples of two objects based on classes Display_Output and User_Keypad_Input. It demonstrates how the objects are shown using UML. An object is shown by a rectangular box with the object identity followed by a semicolon and then the class identity of which that object is an instance and functional entity. Task_Display and TaskUser_KeypadInput are the instances of Display_Output and User_Keypad_Input.

***State Diagram*** Figure 11.4 shows a state diagram for ACVM tasks. It demonstrates how the classes are shown using UML. A state diagram shows a model of a structure for its start, end, in-between associations through the transitions and shows event labels (or conditions) with associated transitions. A state diagram is represented as follows: A dark circular mark shows the starting point, and arrows show the transitions. A label over an arrow shows the condition or event, which fires that transition. A dark rectangular mark within a circle shows the end. [Table 6.3] The state transitions take place between the tasks Task_GUI, Task_Display, TaskUser_KeypadInput, Task_Communication, Task_ReadPorts, Task_Refund, Task_ExcessRefund, Task_Collect, Task_Deliver and Task_Display.

## 11.1.4 ACVM Hardware Architecture

Hardware architecture specifies the appropriate decomposition of hardware into processor(s), ASIPs, memory, ports, devices, and mechanical and electromechanical units. It also specifies interfacing and mapping of these components. Figure 11.5 shows a block diagram of ACVM hardware architecture. Following are the specifications:

1. Microcontroller 8051MX. This version enables use of RAM and ROM larger than 64 kB.
2. 8 MB *ROM* for application codes and RTOS codes for scheduling the tasks. 64 kB *RAM* for storing temporary variables and stack.

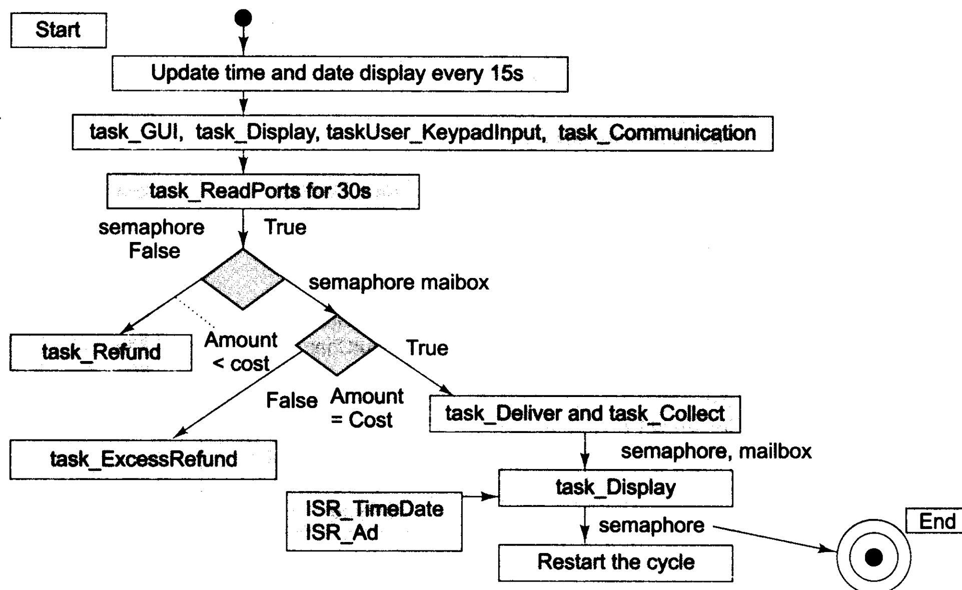



Fig. 11.4 State diagram for ACVM tasks

3. 64 kB flash memory part of the ROM stores user preferences, contact data, user address, user date of birth, user identification code, and answers of FAQs.
4. A 1 μs resolution timer is obtained by programming 8051 timer T0 interrupt service routine. Eight hardware-interrupts with 8 interrupt vectors are used for servicing the hardware interrupts.
5. A TCP/IP port provides Internet broadband connection through a wireless USB modem, for remotely controlling the ACVM and for retrieving the ACVM status reports by the owner. The ACVM can send warning and error reports to the owner. The Internet port helps in the future updating of the machine to install new applications. For example, an application which sends SMS messages upon arrival of fresh chocolates at the machine or e-mail birthday greetings to users.

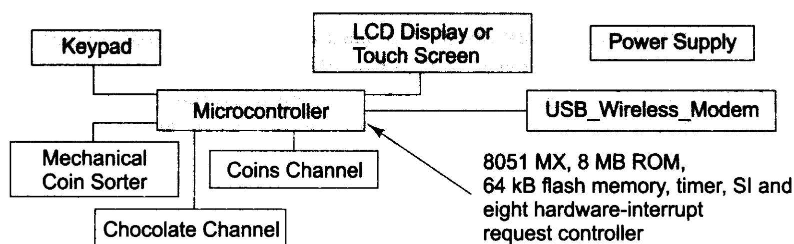



Fig. 11.5 Block diagram of ACVM hardware including Microcontroller

6. ACVM specific hardware to *sort the coins of different denominations.* Each denomination coin generates a set of status bits for the coin inputs and a port-interrupt request (notification for hardware event). Using an interrupt service routine for that port, the ACVM processor reads the port status and input bits. The bits give the information as to which type of coin has been inserted. After each read operation, the status bits are reset by the routine.

7. Main *power supply* of 220 V 50 Hz or 110 V 60 Hz. Internal circuits are driven by a supply of 5 V 50 mA for electronic and 12 V, 2 A for mechanical systems.

## 11.1.5 Software Architecture

Software architecture specifies the appropriate decomposition of software into modules, components, appropriate protection strategies, and mapping of software.

Figure 11.6 shows a design for the software architecture using the classes for Tasks and ISRs. ACVM_Tasks extends to the ACVM_System_ISRs and ACVM_System_Tasks. ACVM_System_ISRs extends to m ISRs, ISR_Task1, ..., ISR_TaskM. ACVM_System_ISRs extends to Task1,..., TaskN.

ISR_Task1, ..., ISR_TaskM and Task1,..., TaskN are as follows:

1. ISR_KeypadInut, ISR_TimeDate, ISR_Ad, ISR_Port1, and ISR_Port2 and ISR_Port5 are the ISRs, which service the device interrupts.

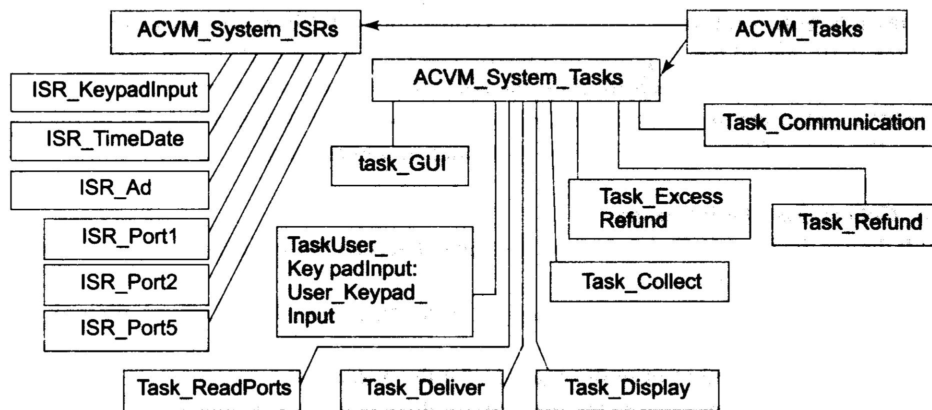



Fig. 11.6 Software architecture of ACVM

2. Task_GUI, Task_Display, TaskUser_KeypadInput, Task_Communication, Task_ReadPorts, Task_Refund, Task_ExcessRefund, Task_Collect, Task_Deliver and Task_Display are the tasks, which the kernel schedules.
3. ISR_KeypadInput for Keypad input read an interrupt on pressing a key. On getting an interrupt from the keypad, a message is read from the keypad device buffer. The message is as per the sought input at that instance. The message is for one of the following: input line (for the characters received for input line, for example, text during child keying in the ID or name, address, date of birth), input_character (for example, for the upKey or downKey or leftKey or rightKey or enterKey or YKey or NKey) or an input_choiceAcceptanceFlag. The flag = true (a Boolean when for acceptance of user for the choice marked and shown at display at that instance for one of the offered menu for selection on screen). The input_choiceAcceptanceFlag notifies the acceptance of selected choice. The message input line, or input_character or input_choiceAcceptanceFlag is notified to the Task User_Keypad_Input.
4. ISR_KeypadInput posts the message from the keyboard to the task object of User_Keypad_Input which runs one of the three methods: getFlag ( ), getString ( ) or getChar ( ). [Figure 11.3]
5. An ISR_TimeDate is for updating the time and date. ISR_TimeDate executes on a timer overflow interrupt, and saves the time and date information. A message is notified to Task_Display. Task_Display executes the method displayTimeDate ( ). [Figure 11.3]

6. A memory buffer saves the advertisements for display on the ACVM in its idle state at periodic intervals. An ISR_Ad is for selecting an advertisement for display. On a timer overflow interrupt, the ISR_ Ad is unmasked at periodic intervals and a software interrupt ISR_Ad is enabled for execution. ISR_Ad posts a message notification to Task_Display. One advertisement is picked at an instant from the memory buffer. The Task_Display executes the method displayAdv ( ).
7. Task_Display is an object of the class Display_Output [Figure 11.3].
8. Task_Display runs one of the following methods: displayMessage ( ) for messages in mailbox posted from the tasks, and displayTimeDate ( ) or displayAdv() or displayPicture ( ) or displayMenu ( ). The display method which runs on notification from ISR is according to the message notification posted from ISR_TimeDate or ISR_Ad or ISR_Picture or ISR_Menu.
9. Task_ReadPorts is an object of the class User_Keypad_Input [Figure 11.3]. Task_ReadPorts does the following. (i) It reads the byte (8 bits) at each of the above ports. (ii) If the coin status, as reflected by SemAmtCount, is as per the cost of the chocolate, it sends a flag to Task_Collect. The latter task initiates actions for collecting the coins into a collection unit. (iii) It also resets the bits after reading to keep the ports and all the 24 points in a ready state for the next cycle of the machine. (iv) It does the other actions described in Step 10, if (a) the coins do not accumulate as per the cost within a specific timeout period or are in excess of the cost, or (b) it does other Step 8 actions for displaying messages as per the state of the port just before switching to another task. It sends messages through three mailbox pointers.
10. Task_Collect does the following:
    (i) It directs the Port_Collect to act and the unit collects all the available coins from the ports, Port_1, Port_2 and Port_5.
    (ii) After collection is over, it sends an IPC to another task, Task_Deliver, through Port_Deliver. Port_Deliver, on receiving an IPC, initiates action for delivering the chocolate and also posts a signal or semaphore to reset the machine for the next cycle.
    (iii) Lastly, it sends a message in the mailbox for display in Task_Display.

The following are the details of multiple tasks:

1. Mechanical subsystems also provide a facility, which helps when there are two or more coins of the same type. Thus, there can be a maximum of eight total different points at each port. There are 24 bits, 8 at each port, Port_1, Port_2 and Port_5. These are in reset state (0s) or reset on power-up. The ACVM circuit design is such that at port $0^{th}$ bit, bit 0 is set (=1) when one coin is available and identified properly. Port bit 1 is set when two are available, bit 2 sets when three are available, and so on. There are 8 bits for 8 points at the port. The number of 1's at the port determines the number of coins at that port.
2. Besides being attached to mechanical subsystems, each of these ports sends 8 input bits for reading at ISR_Port1 or ISR_Port2 or ISR_Port5 using mailbox message pointers. The ISRs send the message pointer for the port bits.
3. Use of a mailbox MboxAmount is such that within a time-out period, a child can thus either insert the chocolate cost by 8 coins of Re 1 for 8 input points at Port_1, or insert a coin of Re 1 for Port_1 point, of Rs 2 for Port_2 point and Rs 5 for the Port_5 point. The child has options for several possible combinations for using the machine. The system recovers the cost of a chocolate before collecting coins and delivering the chocolate.
4. When a port Port_Collect receives a direction (a signal in the form of a flag) from the Task_Collect, then all 24 bits for the 24 points at the three ports release the coins using an electromechanical device. Coins collect at a collection unit. To recover the coins, the machine owner on a convenient time opens

the unit by a lock and withdraws the money. The machine owner also fills the coins in the unit at Port_ExcessRefund for refunding when the child inserts an extra amount or coin.

5. When a port, Port_Refund, receives a direction (a signal in the form of a flag) from a task, Task_Refund, it directs all the eight points at each of the ports to release the coins if any, using an electromechanical device. The coins drop in a bowl if the amount at the three ports is found to be less than the cost. When a port, Port_ExcessRefund, receives a direction (a signal in the form of a flag) from Task_ExcessRefund, it directs the eight points at another port to release the excess amount, using an electromechanical device, and the coins drop in the bowl, provided the amount at the three ports is found to be more than the cost. [To recover the coins, the child looks at a bowl to withdraw the refunded money.]
6. Task_Refund does the following: (i) It directs the Port_Refund to act and the unit sends the coins from three ports to a bowl when the amount is short. (ii) It does the other required actions. It sends a display message for the mailboxes, and the mails in which Task_Display waits.
7. Task_ExcessRefund does the following: (i) It directs the excess refunding on the unit to refund the excess amount. (ii) It does the other actions described in Step 8 and messages to the mailboxes.
8. At this step, an LCD port gets a message from a Task_Display. The display is as per the state of the machine or time-date mailed to the task waiting mailboxes. The displayed messages are as follows: (i) When the machine resets or on the start of a cycle, in the first line, a welcome message, "*Welcome to Sweet Memories Chocolates*" is displayed. The second line display is "*Insert amount, please*". The third line on the right corner displays "*time and date*" from a Task_TimeDateDisplay which sends the message every second. (ii) After a mail from Task_Collect, the first line shows a message, "*Wait for a moment*". "*Collect a nice chocolate soon*" is displayed in the second line. The message clears after a timeout. (iii) After a mail from Task_Deliver, the first line message is "*Collect the nice chocolate*". The second line message is "*Insert coins for more*". The message clears after a timeout. (iv) After an event flag from Task_Refund, the message in the first line is "*Sorry*"! The second line displays "*Please collect the refund*". The messages clear after a timeout. (iv) After the mails, Task_ExcessRefund displays in the first line, "*Collect the chocolate and money*". "*Do not forget to collect the excess*" is displayed in the second line. The message clears after a time out. (v) After a timeout or on machine reset for the next cycle, the display is repeated as in step (i).
9. OSSemPost and OSSemPend semaphore functions at MUCOS synchronize such that Task_ReadPorts waits for execution of the codes till the necessary amount, indicated by SemAmtCount, are collected within a specified timeout period. Task_Deliver sends a signal to Port_Deliver that delivers a chocolate from ACVM *and it must deliver only on* collection of specific coin combinations such that the amount received is equal or more than the chocolate cost.

It is a must that multiple tasks need to be synchronized with respect to each other, using the IPCs. An RTOS is thus required and the binary-semaphores, resource-key semaphores, counting semaphores and mailboxes are used for synchronizing and concurrent processing.

***Synchronization Diagram*** Let MUCOS RTOS be the choice for an embedded software development for ACVM. Figure 11.7 shows multiple tasks and their synchronization model. It demonstrates how to draw synchronization diagrams. The figure mentions synchronization objects near the line connecting a task with another task. The synchronization object(s) is a semaphore(s) or mailbox message(s) for notification. The steps for synchronization are as follows:

1. Task_ReadPorts starts action only when a semaphore SemFlag1 is posted from the ports. It also accept the semaphore timeout in case available. Stimeout semaphore is released by timer service routine for tick after 30s. It accepts amount messages from Port_1, Port_2 and Port_5. It posts message pointers for mailbox waiting for the mail at Task_Collect.

2. Task_Collect waits for taking SemFlag1 and MboxAmount. It releases SemFlag2 to let a Task_Deliver provide the chocolate. Task_Collect also releases SemFlag2 again which Task_ExcessRefund takes. This is because the child has already paid extra, so s/he must get a chocolate.

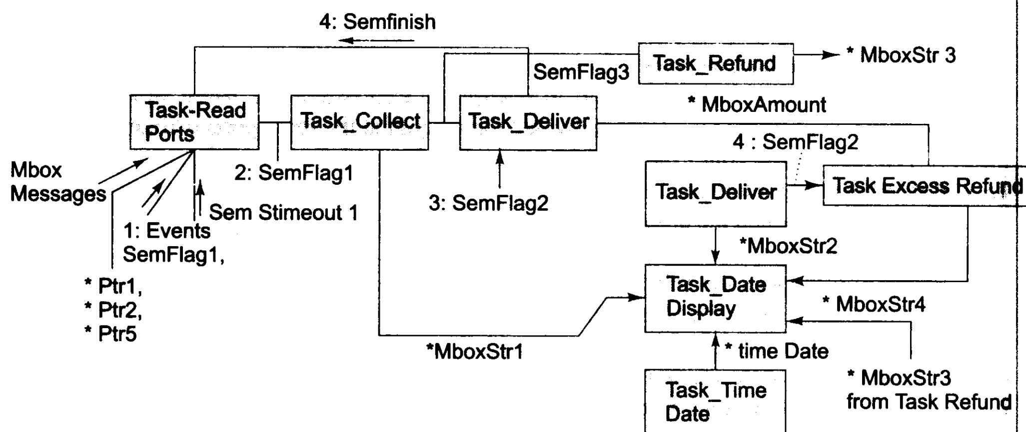



Fig. 11.7 Multiple tasks and their synchronization model using semaphores and mailbox messages

3. Task_Refund waits for taking SemFlag3. It sends message to *MboxStr3.
4. Task_ExcessRefund waits for taking SemFlag2 and posts mailbox message Str4 for display. Task_Display waits for taking SemFlag4. It takes mutex, SemMKey2 before passing the bytes to a stream for Port_Display and releases it after sending. It displays the mailbox messages at the message pointers, *Collect, *delivered, *refund, and *ExcessRefund.
5. Method displayTimeDate ( ) displays at Task_Display gets a timeout notification through a mailbox message for time and date. The time outs occur from ISR_TimeDate after every 1000 ticks of the system clock. A timeout updates the time and date values at a pointer *timeDate. It posts into the mailbox *timeDate to Task_Display and displayTimeDate ( ) uses it to display in the third line, at the right corner of the LCD.

## 11.1.6 Creating a List of Tasks, Functions and IPCs

For design implementation using a MUCOS RTOS environment, let us create a table. Table 11.2 gives the design table. Task synchronization model is used for creating the table. The table gives *name*, *priority* and *actions expected* from the task in columns 1, 2 and 3, respectively in each row. IPCs pending and IPCs posted are given in columns 4 and 5. Mechanical or other system inputs and outputs are given in the last columns, 6 and 7.

## 11.1.7 Exemplary Coding Steps

The task objects in Figure 11.6 can be implemented as C functions in a MUCOS environment with C as the programming language. MUCOS IPC functions can implement the synchronization model shown in Figure 11.7. The following are the MUCOS coding steps for the task objects listed in Table 11.2.

**Table 11.2** List of Six Tasks, Functions and IPCs

| *Task Function* | *Priority* | *Action* | *IPCs pending* | *IPCs posted* | *ACVM input* | *ACVM Output* |
|---|---|---|---|---|---|---|
| Task_ReadPorts | 9 | Waits for the coins and action as per coins collected | MboxPtr1Msg, MboxPtr2Msg, MboxPtr5Msg, SemFlag1 messages and Event signals from Port_1, Port_2 and Port_5; Stimeout from timer ISR | Message Pointer *MboxAmount | Coins at Port_1, Port_2 and Port_5. | — |
| Task_Collect | 11 | Waits for coins = or > cost till timeout and acts accordingly | SemFlag1, *MboxAmount | SemFlag2, SemFlag3, Message Pointers *MboxAmount, *Str1 | – | Coins at Port_Collect it amount >= cost |
| Task_Deliver | 12 | | SemFlag2 | SemFlag2, Message Pointer *Str2 | Chocolate from a channel | Delivers Chocolate into the bowl. |
| Task_Refund | 17 | Waits for refund event and refunds the amount | SemFlag3 | Message Pointer *Str3 | Coins at Port_1, Port_2 and Port_5 | Coins Flushed back to the bowl. |
| Task_Excess Refund | 13 | Refunds the excess amount | SemFlag2, *MboxAmount | Message Pointer *Str4 | Coins at the Ports Excess Refund channel | Excess Coins from Port_Excess Refund |
| Task_Display | 15 | Waits for the message mails | Message pointers: Collect, *delivered, *refund, *ExcessRefund (Str 2, Str3, Str4) and *timeDate | | Strings for line 1, 2 and 3 and time date. | Bytes for the LCD display to lines 1, 2 and 3. |

## Example 11.1

```
1. /* Define Boolean variable, define a NULLpointer to point in case mailbox is empty. */
typedef unsigned char int8bit;
#define int8bit boolean
#define false 0
```

```
#define true 1
/* Define a NULL pointer; */
#define NULL (void*) 0x0000
/* Preprocessor commands define OS tasks service and timing functions as enabled and their constants;
similar to Example 9.7. */
#define OS_MAX_TASKS 10
#define OS_LOWEST_PRIO 28 /* Let lowest priority task be 19. */
#define OS_TASK_CREATE_EN 1 /* Enable inclusion of OSTaskCreate ( ) function */
#define OS_TASK_DEL_EN 1 /* Enable inclusion of OSTaskDel ( ) function */
#define OS_TASK_SUSPEND_EN 1/* Enable inclusion of OSTaskSuspend ( ) function */
#define OS_TASK_RESUME_EN 1/* Enable inclusion of OSTaskResume ( ) function */
.
.
/* Specify all child prototype of the first task function that is called by the main function and is to be
scheduled by MUCOS at the start. In Step 11, we will be creating all other tasks within the first task */
/ * Remember: Static means permanent memory allocation */
static void FirstTask (void *taskPointer);
static OS_STK FirstTaskStack [FirstTask_StackSize];
/* Define public variables of the task service and timing functions */
#define OS_TASK_IDLE_STK_SIZE 100 /* Let memory allocation be for an idle state task stack size be
100*/
#define OS_TICKS_PER_SEC 1000 / * Let the number of ticks be 1000 per second. An RTC tick will
interrupt and thus tick every 1 ms to update counts. */
#define FirstTask_Priority 4 /* Define first task in main priority */
#define FirstTask_StackSize 100 /* Define first task in main stack size */
#define cost 8 /* Define cost of chocolate as Rs 8 */
2. /* Preprocessor definitions for maximum number of inter-process events to let the MUCOS allocate
memory for the Event Control Blocks */
#define OS_MAX_EVENTS 24/* Let maximum IPC events be 24 */
#define OS_SEM_EN 1/* Enable inclusion of semaphore functions. */
#define OS_MBOX_EN 1/* Enable inclusion of mailbox functions for the mailing of the
message pointers to Task_Display. */
#define OS_Q_EN 1/* Enable inclusion of queue functions for sending the string pointers
to LCD matrix Port_Display * /
/* End of preprocessor commands */
3. /* Prototype definitions for six tasks */
static void Task_ReadPorts (void *taskPointer);
static void Task_ExcessRefund (void *taskPointer);
static void Task_Deliver (void *taskPointer);
static void Task_Refund (void *taskPointer);
static void Task_Collect (*taskPointer);
static void Task_Display (void *taskPointer);
/* Definitions for six task stacks. */
static OS_STK Task_ReadPortsStack [Task_ReadPortsStackSize];
static OS_STK Task_ExcessRefundStack [Task_ExcessRefundStackSize];
```

```
static OS_STK Task_DeliverStack [Task_DeliverStackSize];
static OS_STK Task_RefundStack [Task_RefundStackSize];
static OS_STK Task_CollectStack [Task_CollectStackSize];
static OS_STK Task_DisplayStack [Task_DisplayStackSize];
/* Definitions for six task-stack sizes. */
#define Task_ReadPortsStackSize 100 /* Define task 1 stack size*/
#define Task_ExcessRefundStackSize 100 /* Define task 2 stack size*/
#define Task_DeliverStackSize 100 /* Define task 3 stack size*/
#define Task_RefundStackSize 100 /* Define task 4 stack size*/
#define Task_CollectStackSize 100 /* Define task 5 stack size*/
#define Task_DisplayStackSize 100 /* Define task 6 stack size*/
4. /* Definitions for six task-priorities. */
#define Task_ReadPortsPriority 9 /* Define task 1 priority */
#define Task_ExcessRefundPriority 13 /* Define task 5 priority */
#define Task_DeliverPriority 12 /* Define task 3 priority */
#define Task_RefundPriority 17 /* Define task 4 priority */
#define Task_CollectPriority 11 /* Define task 2 priority */
#define Task_DisplayPriority 15 /* Define task 6 priority */
5. /* Prototype definitions for the semaphores. */
OS_EVENT *SemFlag1; /* On interrupt signals from Port_1 to Port_5, Task_ReadPorts starts running.
This flag needed when using semaphore for inter-process communication between tasks reading amount
message from Port_1, Port_2 and Port_5. */
OS_EVENT *SemFlag2; /* Needed when using semaphore as flag for inter-process communication for
task_delivery over event notification and between Task_Collect and amount refunding task, Task_Delivery. */
OS_EVENT *SemFlag3; /* Needed when using semaphore as flag for inter-process communication
between Task_Collect and Task_ExcessRefund. */
OS_EVENT *Stimeout; /* Semaphore posted by a timer ISR after 30s wait */
OS_EVENT *SemVal; /* Needed when using semaphore for passing the 16-bit message between steps b
and c */
6. /* Prototype definitions for the mailboxes */
/* For using mailbox messages. These ISRS are for posting bits from Port_1, 2 and 5. */
OS_EVENT *MboxPtr1Msg;
OS_EVENT *MboxPtr2Msg; /*
OS_EVENT *MboxPtr3Msg;
OS_EVENT *MboxStr1Msg, *MboxStr2Msg, *MboxAmount, *MboxTimeDateStrMsg; /* Needed when
using mailbox message for sending display, amount and timedate pointers */
OS_EVENT *MboxStr3Msg, *MboxStr4Msg;
/*------------------------------------------------------------------------------------------*/
7. /* Define four Semaphores as event flags
SemFlag1 = OSSemCreate (0); /* Declare initial value of semaphore = 0 for using it as an event flag*/
SemFlag2 = OSSemCreate (0); /* Declare initial value of semaphore = 0 for using it as an event flag*/
SemFlag3 = OSSemCreate (0); /* Declare initial value of semaphore = 0 for using it as an event flag*/
SemStimeOut = OSSemCreate (0); /* declare initial value = 0 for timeout flag */
8. /* Create eight Mailboxes for the tasks. */
MboxAmount = OSMboxCreate (NULL);
```

```
MboxStr1Msg = OSMboxCreate (NULL);
MboxStr2Msg = OSMboxCreate (NULL);
MboxStr3Msg = OSMboxCreate (NULL);
MboxStr4Msg = OSMboxCreate (NULL);
MboxTimeDateStrMsg = OSMboxCreate (NULL); /* For message from Task_Display. */
9. /* Any other OS Events for the IPCs. */
OSMboxPtr1Msg = OSMboxCreate (NULL);
OSMboxPtr2Msg = OSMboxCreate (NULL);
OSMboxPtr5Msg = OSMboxCreate (NULL);
10. /* The codes are for reading from Port A and storing a character. Here, we have three ports, Port_1, Port_2 and Port_5 for Rs 1, 2 and 5 denomination coins. These are basically device driver codes for port_1, port_2 and port_5 and three status flags for resetting to the beginning. */
STAF _1 = 0;
STAF _2 = 0;
STAF _3 = 0;
.
.
11. /* Start of the codes of the application from Main.
Note: Code steps are similar to Steps 9 to 17 in Example 9.16 */
.
int port amount (int *amt); /* declare function to convert string from port to an integer value for amount */
void main (void) {
12. /* Initiate MUCOS RTOS to let us use the OS kernel functions */
OSInit ( );
13. /* Create first task, FirstTask that must execute once before any other. Task creates by defining its identity as FirstTask, stack size and other TCB parameters. */
OSTaskCreate (FirstTask, void (*) 0, (void *) &FirstTaskStack [FirstTask_StackSize], FirstTask_Priority);
/* Create other main tasks and inter-process communication variables if these must also execute at least once after the FirstTask. */
14. /* Start MUCOS RTOS to let us RTOS control and run the created tasks */
OSStart ( );
/* Infinite while-loop exits in each task. So there is no return from the RTOS function OSStart ( ). RTOS takes the control forever. */
}/ *** End of the Main function ***/
15. /* The codes of ISR_Keypad Input, ISR_Port 1, ISR_Port 2, ISR_Port 5 for ACVM_System_ISRs */
16. static void FirstTask (void *taskPointer){
17. /* Start Timer Ticks for using timer ticks later. */
OSTickInit ( ); /* Function for initiating RTCSWT that starts ticks at the configured time in the MUCOS configuration preprocessor commands in Step 1 */.
18. /* Create six Tasks defining by six task identities, Task_Display, Task_ReadPorts, Task_ExcessRefund, Task_Deliver and Task_Refund and the stack sizes, other TCB parameters.
*/
OSTaskCreate (Task_Display, void (*) 0, (void *) & Task_DisplayStack
[Task_DisplayStackSize], Task_DisplayPriority);
OSTaskCreate (Task_ReadPorts, void (*) 0, (void *) & Task_ReadPortsStack
[Task_ReadPortsStackSize], Task_ReadPortsPriority);
```

```
OSTaskCreate (Task_ExcessRefund, void (*) 0, (void *) & Task_ExcessRefundStack
[Task_ExcessRefundStackSize], Task_ExcessRefundPriority);
OSTaskCreate (Task_Deliver, void (*) 0, (void *) & Task_DeliverStack [Task_DeliverStackSize],
Task_DeliverPriority);
OSTaskCreate (Task_Refund, void (*) 0, (void *) & Task_RefundStack [Task_RefundStackSize],
Task_RefundPriority);
OSTaskCreate (Task_Collect, void (*) 0, (void *) & Task_CollectStack [Task_CollectStackSize],
Task_CollectPriority);
19. while (1) { /* Start of the while loop*/
20. /* Suspend with no resumption later the First task, as it must run once only for initiation of timer ticks
and for creating the tasks that the scheduler controls by preemption. */
OSTaskSuspend (FirstTask_Priority); /*Suspend First Task and control of the RTOS passes forever to
other tasks, waiting their execution*/
21. } /* End of while loop */
22. } /* End of FirstTask Codes */
/*****************************************************************************/
23. /* Function for finding amount value of the coins collected at ports */
Static int portamount (amt) {
int amount;
Switch (amt) {
case 0 : amount = 0; exit ( )
case 1 : amount = 1; exit ( )
case 2 : amount = 2; exit ( )
case 4 : amount = 3; exit ( )
case 8 : amount = 4; exit ( )
case 16 : amount = 5; exit ( )
case 32 : amount = 6; exit ( )
case 64 : amount = 7; exit ( )
case 128 : amount = 8; exit ( )
}
return amount;
}
24. /* The codes for Task_ReadPorts */
/*****************************************/
Static void Task_Read Ports (void * taskPointer)
{
25. /* initial declarations */
int amtport = 0; /* value of amount collected at ports */
int *amount1; /* Pointer to Port 1 message for amount */
int *amount2; /* Pointer to Port 2 message for amount */
int *amount5; /* Pointer to Port 5 message for amount */
26. while (1) { /* Start of while loop */
27. OSSemPend (SemFlag1, 0, *SemErrPointer); /* wait for semaphore from port ISRs */
28. while (Stimeout = = false || amtport < cost){ OSSemAccept (Stimeout, 0, *SemErrPointer);
29. *amount1 = OSMboxAccept (MboxPtr1Msg, 0, ErrPointer M1);
```

```
*amount2 = OSMboxAccept (MboxPtr2Msg, 0, ErrPointer M2);
*amount5 = OSMboxAccept (MboxPtr5Msg, 0, ErrPointer M5)
30. amtport = portamount (&amount1) + 2 * portamount (& amount2) + 5 *_portamount (& amount5)
} /* wait till amount at ports > = cost */
OSMboxPost (MboxAmount, &amtport); /*post the among value */
amtport = 0; / *set the amount at port = 0 */
OSTaskSuspend (Task_ReadPortsPriority);
/* The lower priority task_collect now runs */
31. }; /* End of while loop*/
}/ * End of the Task_ReadPorts function */
/*********************************************************************/
32. /* Start of Task_Collect codes */
static void Task_Collect (void *taskPointer) {
33. /* Initial Assignments of the variables and pre-infinite loop statements that execute once only*/
int *amount = 0;
.
34. while (1) { /* Start an infinite while-loop /
35. /* Take Mbox amount value */
*amount = OSMboxPend (MboxAmount, 0, MboxErrPointer);
36. if (*amount > = cost) {OSMboxPost (MboxAmount, *amount);
OSMboxPost (MboxStr1Msg, "wait for a moment collect a nice chocolate soon"); /* Send display
message */
OSSemPost (semFlag2, 0 * ErrPointer); /* Event notify to task_deliver */
37. /* codes for mechanical system to collect the coins from Port_1, Port_2 and Port_5 so that ports are
empty for new coins input *}
.
;}
else {OSSemPost (SemFlag3, 0, *ErrPointer)}
OSTaskSuspend (Task_CollectPriority); /* Task_Refund */
OSTaskResume (Task_ReadPorts);
}; End of while loop
38. } / * End of the Task_Collect function */
/*********************************************************************/
39. /* The codes for the Task_Deliver */
static void Task_Deliver (void *taskPointer) {
40. /* The initial assignments of the variables and pre-infinite loop statements that execute once only*/
.
.
41. while (1) { /* Start an infinite while-loop. */
42. /* Wait for flag SemFlag2 from Task_Collect */
OSSemPend (SemFlag2, 0, SemErrPointer);
43. /* Codes for device driver for Port_Deliver for delivering a chocolate into a bowl. */
.
.
44. /* Post two mails to waiting Task_Display through two message pointers for first line and second
line of LCD matrix at Port_Display. */
```

```
OSMboxPost (MboxStr2Msg, "Collect the nice chocolate. Thank you, Insert coins for more"); /*
45. /* Let delayed higher priority task err resume. */
46./* Post semaphore to flag that the chocolate delivery is over. */
OSSemPost (SemFlag2); /* SemFlag3 to task excess refund if any */
OSTaskSuspend (Task_DeliverPriority);
/* This enables low priority take to start */
OSTaskResume (Task_CollectPriority);
}; /* End of while loop*/
47. }/ * End of the Task_Deliver function */
/*****************************************************************************/
48. /* Start of Task_Refund codes */
static void Task_Refund (void *taskPointer) {
/* Initial assignments of the variables and pre-infinite loop statements that execute once only*/
.
.
49. while (1) { /* Start the infinite loop */
OSSemPend (SemFlag3, 0, SemErrPointer);
50. /* Code for the device driver to let Port_Exit release the coins as refund as within twenty cycles of clock ticks, if child fails to insert the coins of the required amount. */
.
.
51. /* Post mail to waiting Task_Display through message pointer for first line and second line of LCD matrix at Port_Display. */
OSMboxPost (MboxStr3Msg, "For chocolate,"Insert coins again collect Refund"); /*
OSTimeDlyResume (Task_Display);
OSTaskResume (Task_CollectPriority); /* Return to Task_Collect_Priority. */
}; /* End of while loop*/
52. } / * End of the Task_Refund function */
/*****************************************************************************/
53. /* Start of Task_ExcessRefund codes */
static void Task_ExcessRefund (void *taskPointer) {
54. /* Initial assignments of the variables assignments and pre-infinite loop statements that execute once only*/
.
.
55. while (1) { /* Start the infinite loop */
/* Wait for SemFlag2 from chocolate deliver task */
OSSemPend (SemFlag2, 0, SemErrPointer);
/* Wait for amount pointer in order to refund the excess amount */
*amount = OSMboxPend (MboxAmount, 0, *MboxErrPointer);
*amount = *amount_cost;
/* Reduce the amount by cost of chocolate */
56. /* Reset Amt */
57. /* Code for the device driver to let Port_ExcessRefund release the coins for balance amount from a channel as per the Amt value now. AVCS thus refund the coins, if child fails to insert the coins of the required amount. */
```

```
.
.
58. /* Post mail to waiting Task_Display through message-pointer for the first line and second line of LCD matrix at Port_Display. */
OSMboxPost (MboxStr4Msg, "Pl collect the excess Amount inserted Thank you, Visit Again" )/*
59. OSTaskSuspend (Task_Excess Refund Priority);
OSTask_Resume (Task_Deliver_Priority);
        }; /* End of while loop*/
60. }/ * End of the Task_ExcessRefund function */
/*********************************************************************/
61. /* The codes for the Task_Display */
static void Task_CharCheck (void *taskPointer) {
62. /* Declare string variables for the three lines and other initial assignments and pre-infinite loop statements that execute once only. */
unsigned char [ ] Str1, Str2, Str3, Str4, currentTimeDate; /* A variable to display at the right corner of line 3 of LCD Matrix */
.
.
63. /* Start an infinite while-loop. */
while (1) {
 /* Wait for Messages for line 1, line 2 and line 3. */
Str1= OSMboxAccept (MboxStr1Msg, 0, MboxErrPointer);
Str2 = OSMboxAccept (MboxStr1Msg, 0, MboxErrPointer);
Str3 = OSMboxAccept (MboxStr3Msg, 0, MboxErrPointer);
Str4 = OSMboxAccept (MboxStr4Msg, 0, MboxErrPointer);
64. currentTimeDate = OSMboxAccept (MboxTimeDateStrMsg, 0, MboxErrPointer);
displayTimeDate (currentTimeDate );
/* TimeDate display */
if (Str1 ! = NULL) displayStr1 ( );
if (Str2 ! = NULL) displayStr2 ( );
if (Str3 ! = NULL) displayStr3 ( )
if (Str4 ! = NULL ) displayStr4 ( );
65. /* Device driver Codes for sending the four strings to a byte stream from line 0 character first to last character line through Port_Display. */
.
.
OSTimeDly (200) (SemMKey2);
OSTaskResume (Task_ExcessRefundPriority);
66. }/ * End of While loop
}; /* End codes for the Task_Display */
/*********************************************************************/
67. /* Codes for function displayTimeDate */
static void displayTimeDate (unsigned char [ ] currentTimeDate ) {
68. /* Initial assignments of the variables */
unsigned char *timeDate;
```

```
.
.
.
}/* End of Codes for displayTimeDate function */
69. /* ISR codes for posting mailbox message to Task_Display */
ISR_TimeDate ( ) {
/* Codes for creating a message for time and date after each 1000th interrupt from the system
RTC tick. */
.
.
70. OSMboxPost (MboxTimeDateStrMsg, timeDate);
71. } /* End of ISR_TimeDate code*/
```

## 11.2 CASE STUDY OF DIGITAL CAMERA HARDWARE AND SOFWARE ARCHITECTURE

The digital camera was introduced earlier in Section 1.10.4. A digital camera is an example of SoC. [Section1.6.] Section 1.10.4 listed the camera functions, hardware and software units. Figures 1.14 showed the hardware and software components in a simple digital camera.

Sections 11.2.1 and 11.2.2 give the design steps of a digital camera. Sections 11.2.3 and 11.2.4 describe hardware and software architecture.

### 11.2.1 Requirements

Requirements of the digital camera can be understood through a requirement table given in Table 11.3.

***The detailed functions inside the camera are as follows:***

1. A set of controllers control shutter, flash, (for example, for peripherals, direct memory access, and buses), auto focus and eye-ball image control. GUI consists of the LCD display for graphics, and switches and buttons for inputs at camera. A touchscreen is another alternative for LCD and keypad. The user gives commands for switching on the camera, flash, shutter, adjust brightness, contrast, color, save and transfer. The user commands are in the form of interrupt signals. Each signal generates from a user input from an operated switch or button. When a button for opening the shutter is pressed, a flash lamp glows and a self-timer circuit switches off the lamp automatically.
2. The picture generates light, which falls on the CCD array, which through an ADC transmits the bits for each pixel in each row in the frame, and also for the dark area pixels in each row in a vertical strip for offset correction in CCD signaled light intensities for each row. The strip is in adjacent frame.
3. A picture consists of a number of pixels. The number of pixels used for a picture determines resolution. Each picture consists of a number of horizontal and vertical pixels. For 2592 × 1944 pixels, there are 2592 × 1944 = 5038848 sets of cells. Each set of pixel has three cells, for the red, green and blue components in a pixel. Each cell gets exposed to a picture when the shutter of camera opens on a user command. The camera records the pictures using a charge-coupled devices (CCD) array. The array consists of a large number of CCD cells, three at each pixel.

**Table 11.3** Requirements of a Digital Camera

| *Requirement* | *Description* |
|---|---|
| Purpose | 1. Digital recording and display of pictures.<br>2. Processing to get the pictures of required brightness, contrast and color.<br>3. Permanent saving of a picture in a file in a standard format at a flash-memory stick (or card).<br>4. Transfer files to a computer through a port. |
| Inputs | 1. Intensity and color values for each picture in horizontal and vertical rows of pixels in a picture frame.<br>2. Intensity and color values for unexposed (dark) areas in each horizontal row of pixels for offset correction in the row.<br>3. User control inputs. |
| Signals, events and notifications | User commands given as signals from switches/buttons. |
| Outputs | 1. Encoded file for a picture.<br>2. Permanent store of the picture at a file on a flash-memory stick.<br>3. Screen display of picture from the file after decoding.<br>4. File output to an interfaced computer. |
| Functions of the system | 1. A color LCD dot matrix displays the picture before shooting. This enables manual adjustment of the view of the picture.<br>2. For shooting, a shutter button is pressed. Then a charge-coupled device (CCD) array placed at the focus generates a byte stream in the output after operations by ADC on analog output of each CCD cell.<br>3. A file creates after encoding (compression) and pixel co-processing as follows: The byte stream is preprocessed and then encoded in a standard format using a CODEC.<br>4. The encoded picture file can be saved for permanent record. A memory stick saves the file.<br>5. The file is used for display of recorded picture using a display processor and can be copied or transferred to a memory stick and to a computer connected through a USB port.<br>6. The LCD displays a picture file after it is decoded (decompressed) using the CODEC. Texts such as picture-title, shooting date and time, and serial number are also displayed.<br>7. A USB port is used for transferring and storing pictures on a computer. Alternatively, Bluetooth or IR port can be used for interfacing the computer. |
| Design metrics | 1. *Power Dissipation*: Battery operation. Battery recharging after 400 pictures (assumed)<br>2. *Resolution*: High-resolution pictures with options of 2592 × 1944 pixels = 5038848 pixels, 2592 × 1728 = 3.2 M, 2048 × 1536 = 3 M and 1280 × 960 = 1M.<br>3. *Performance*: Shooting a 4M pixel still picture in 0.5s. 25 pictures per m (assumed)<br>4. *Process Deadlines*: Exposing camera process in a maximum of 0.1s. Flash synchronous with shutter opening and closing. Picture display latency, maximum of 0.5s.<br>5. *User Interfaces*: Graphic at LCD or touchscreen display on LCD and commands by the camera user through fingers on touchscreen, switches and buttons.<br>6. *Engineering Cost*: US$ 50000 (assumed).<br>7. *Manufacturing Cost*: US$ 50 (assumed). |
| Test and validation conditions | 1. All user commands must function correctly.<br>2. All graphic displays and menus should appear as per the program.<br>3. Each task should be tested with test inputs.<br>4. Tested for 30 pictures per m. |

4. The CCD cells charge up on exposure to light. The charging of each red or green or blue cell of a pixel is according to light intensity and color at that point in the picture. Three analog outputs to the system are generated for each pixel. Each analog output has to be first converted into bits for processing at the next stage [Section 1.3.7]. ADC operations thus take place for recording the picture that focuses on the cells. Each cell analog output is an input to the ADC which creates bytes for processing further. Each ADC byte corresponds to each value of the analog current which is as per the exposed intensity and color of a cell. This operation is called preprocessing.
5. ADC operations also take place for unexposed additional cells for the pixels. These pixels are adjacent to the focused frame in a vertical strip consisting of horizontal rows. These enable measuring the average dark (zero intensity) current value of output for each row of the picture frame. This average is subtracted as the offset dark current for each row of bytes obtained in Step 4. An average is taken after ADC conversions for each cell in the strip and averaging the values of bytes for the row. This subtraction operation for each row of bytes of the picture frame is also a part of preprocessing.
6. Bytes from preprocessing operations in Steps 3 and 4 for each cell for the picture pixels are stored in a file after compression. The processed signals are compressed using a JPEG CODEC and saved in one jpg file for each picture frame. [JPEG stands for Joint Photographic Experts Group]
7. The jpg file is created after JPEG encoding (compression) of bytes from the picture processor. The preprocessed bytestream is compressed using discrete cosine transformations (DCTs). Usually, integer arithmetic is used for saving the processing time at the expense of some inaccuracy in computations, with an advantage of a reduced number of VLSI gates in the preprocessor circuit of CCD bytestream, fast processing (about 6 times) and reduced battery energy (about 6 times). [Frank Vahid and Tony Givargis, *Embedded System —A unified Hardware / Software Introduction*, John Wiley and Sons, Inc. 2002] However, the inaccuracy is not perceptible to human eyes. After the DCTs, quantization is done and then Huffman coding does the compression in JPEG format. Quantization means, for example, division by 2 or 4 or 8 at the expense of reducing stored image resolution in order to create a smaller size file. The jpg file is stored from a memory address in a memory block. A block(s) is reassigned to each file.
8. Pictures and GUIs are shown on a colored LCD dot matrix screen. For display from the file obtained in Step 5 for the picture, the original bytes before encoding are retrieved back for decoding (decompression) operation using CODEC and display co-processor. The decompression operation on the file is by inverse DCTs.
9. The file is used for display through a display processor. A pixel display co-processor is used for displaying the pictures *directly* or after rotate right, rotate-left, move up, move-down, shift left, shift-right, zoom, stretch, change picture to next file and reprocess for display, and change to previous file and reprocess for display.
10. The JPEG file can be copied or transferred to a memory stick using a controller, and to a computer connected through USB port controller. Sony memory stick Micro (M2) has a size of 15 × 12.5 × 1.2 mm and has a flash memory of 2 GB, 160 Mbps data transfer rate [Section 1.3.5].
11. A USB port is used for interfacing a computer for transferring and storing pictures on a computer. A USB controller is used for transfer. Alternatively, a Bluetooth or IR port can be used for interfacing the computer.

## 11.2.2 Class Diagrams

Digital camera file creation, display and transferring to printer, memory stick and USB port can be modeled by the class diagrams of abstract class Picture_FileCreation, Picture_FileDisplay, and Picture_FileTransfer. Figure 11.8 shows three class diagrams of Picture_FileCreation, Picture_FileDisplay and Picture_FileTransfer for creating a JPEG file, displaying the picture and transferring the file to the memory stick, printer and USB port.

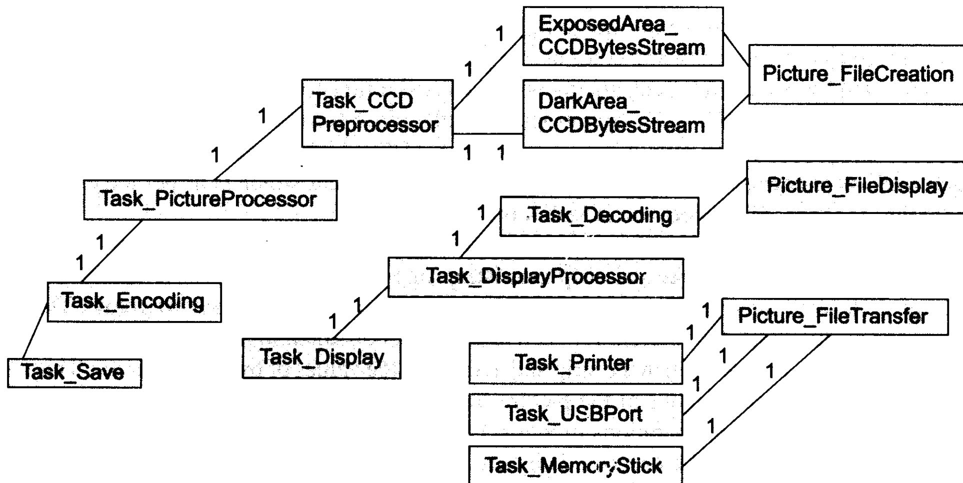



**Fig. 11.8** Three *class diagrams of* Picture_FileCreation, Picture_FileDisplay, Picture_FileTransfer

1. Picture_FileCreation is an abstract class from which an extended class(es) is derived to create a JPEG encoded picture. The task objects are instances of the classes ExposedArea_CCDBytesStream, DarkArea_CCDBytesStream, Task_CCD Preprocessor, Task_PictureProcessor and Task_Encoding.
2. ExposedArea_CCDBytesStream is to create a bytestream from the ADC outputs from the exposed cells in each row of the picture frame.
3. DarkArea_CCDBytesStream is to create a bytestream from the ADC outputs from the unexposed (dark area) cells in each row of the picture frame.
4. Task_CCD Preprocessor creates a stream after the subtraction of the average of bytes for each row of bytes in output from DarkArea_CCDBytesStream from the stream for rows in output of ExposedArea_CCDBytesStream.
5. Task_PictureProcessor creates a stream after processing the Task_CCD Preprocessor for picture brightness, contrast and color adjustment.
6. Task_Encoding creates Huffman encoding for JPEG encoding and creates a filestream for saving onto internal flash or memory stick.
7. Picture_FileDisplay is an abstract class which extends three classes Task_Decoding, Task_ DisplayProcessor, Task_Display.
8. Picture_FileTransfer is an abstract class which extends three classes, Task_Printer, Task_USBPort, Task_MemoryStick.
9. Controller_Tasks which extends to the following tasks: (i) Tasks_Initialization for initialization of tasks, (ii) Tasks_Shoot for shooting tasks, (iii) Initialize_Picture_FileCreation to initialize CCD processor (CCDP), (iv) Initialize_Picture_FileDisplay tasks, which initiates display processor (DisplP), (v) initiates processor (MemP), (vi) initiates processor PrintP, (vi) initiates USB port processor (USB_P), (vi) Task_LightLevel for control level control, (vi) Task_flash.

Drawing the class diagram for Controller_Tasks is an exercise for the reader.

## 11.2.3 Digital Camera Hardware Architecture

The digital camera hardware was described in Section 1.10.4. The camera embeds the following hardware units. It has keys, shutter, lens and charge-coupled device (CCD) array sensors, LCD display unit, a self-timer

lamp for flash, internal memory flash to store OS, embedded software and memory for limited number of picture files, flash memory stick of 2 GB or more for a large number of picture files, a Universal Serial Bus (USB) port (Section 3.10.3) for connecting it to computer and printer. Frank Vahid and Tony Givargis in *Embedded System—A unified Hardware/ Software Introduction*, John Wiley and Sons, Inc. 2002, described the VLSI and implementations of controller, CCD preprocessor with arithmetic implementation of floating point DCT, CCD preprocessor with fixed point DCT and CODEC. The book also described design metrics—performance, energy and cost considerations for each. Figure 11.9 shows a digital camera hardware architecture.

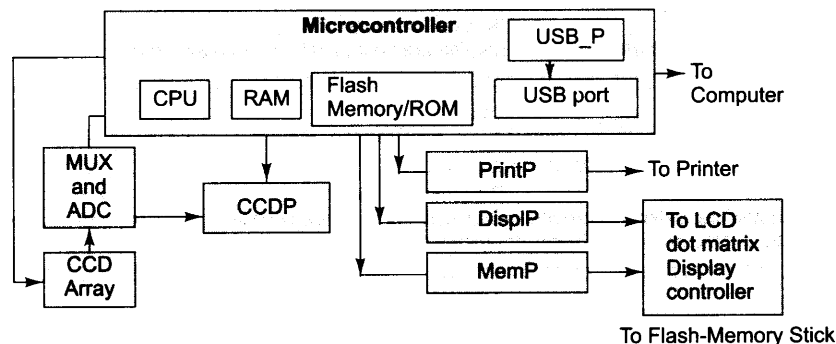



**Fig. 11.9** Digital camera hardware architecture

1. A microcontroller executes the Controller_Tasks. The controller tasks are the following: (i) Task_LightLevel control (ii) Task_flash (ii) initialization of tasks, (iii) shooting task, (iv) initiates Picture_FileCreation tasks, which execute on a single purpose CCD processor (CCDP) for the dark current corrections, DCT compression, Huffman encoding, DCT decompression, Huffman decoding and file save, (v) initiates Picture_FileDisplay tasks, which execute on a single purpose display processor (DisplP) for decoded and compressed file image display after the required file bytestream processing for shift or rotate or stretching or zooming or contrast or color and resolution, (v) initiates memory stick save on a notification from Picture_FileTransfer file system object using a single purpose transfer processor (MemP), (vi) initiates printing on a notification from Picture_FileTransfer using a single purpose print processor (PrintP), and (v) initiates USB port controller on a notification from Picture_FileTransfer using a single purpose USB process (USB_P).
2. Multiple processors (CCDP, DSP, Pixel Processor and others)—a DSP does compression using the discrete cosine transformations (DCTs) and decompression by inverse DCTs. After DCTs, it also does the Huffman coding for the JPEG compression. This operation is done using an ASIP (single purpose processor) and is called CODEC (Section 1.2.4).
3. RAM for storing temporary variables and stack
4. ROM/Flash for application codes and RTOS codes for scheduling the tasks
5. Flash memory for storing user preferences, contact data, user address, user date of birth, user identification code, ADC, and Interrupt controller
6. LCD dot matrix display
7. Memory stick
8. Battery

### 11.2.4 Digital Camera Software Architecture

The camera embeds the following software components. Figure 11.10 shows the software architecture.
There are the following layers:

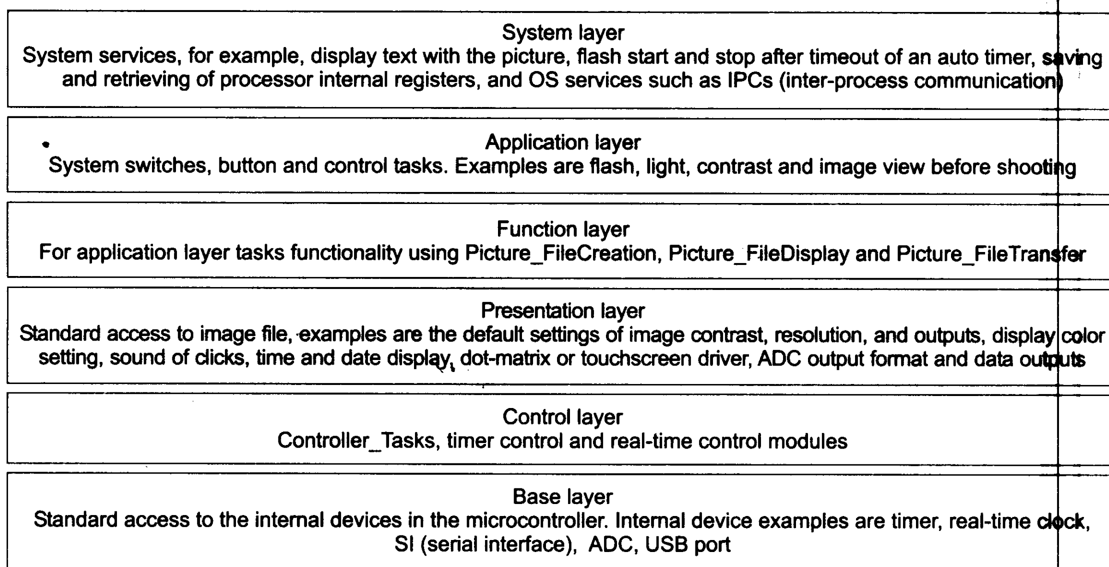



**Fig. 11.10** Software layers in software architecture of a camera system

***System layer*** System layer provides system services, for example, display text with the picture, flash start and stop after timeout of an auto timer, saving and retrieving of processor internal registers, and OS services such as IPCs (inter-process communication).

***Application layer*** Application layer is for system switches, button and control tasks. Examples are flash, light, contrast and image view before shooting.

***Function layer*** Function layer is for application layer tasks functionality using Picture_FileCreation, Picture_FileDisplay and Picture_FileTransfer.

***Presentation layer*** Presentation layer is for providing standard access to an image file. Examples are the default settings of image contrast, resolution, outputs, display color setting, sound of clicks, time and date display, dot-matrix or touchscreen driver, ADC output format and data outputs.

***Control layer*** Control layer is for Controller_Tasks, timer control and real time control modules.

***Base layer*** Base layer provides a standard access to the internal devices in the microcontroller. Internal device examples are timer, real-time clock, SI (serial interface), ADC, and USB port.

Figure 11.11 shows a synchronization model for camera tasks using bytestreams as message queues and semaphores.

Byte Stream
ISR_Shoot_ Picture
Picture_FileCreation Task_PictureProcessor
ExposedArea_ CCDBytesStream
DarkArea_ CCDBytesStream
Task_CCD Preprocessor
Task_Encoding
Task_Save
Byte Stream
Byte Stream
Picture_File Display
SemDecode
Task_Decoding
SemDisP
Task_DisplayProcessor
SemDispl
SemPrint
Task_Printer
Task_Display
Picture_FileTransfer
Task_USBPort
SemUSB
SemMem
Task_MemoryStick

Fig. 11.11 Synchronization model for camera tasks

## 11.3 CASE STUDY OF CODING FOR SENDING APPLICATION LAYER BYTE STREAMS ON A TCP/IP NETWORK USING RTOS VxWorks

Embedded systems find a large number of applications in data communication and networks. When a system embeds the TCP/IP protocol APIs, it becomes Internet enabled. Many systems are Internet enabled in order that the system communicates to other systems using the Internet.

A communication may be within the same system or between remote systems connected through the network drivers. A communication may be from peer-to-peer or between client and server. Figure 7.11 showed the functions of two sockets communicating through a stack. Socket *A* is usually a client socket and Socket *B* is a server socket on a network. Each network socket is identified by a distinct pair of parameters, IP address and port number. Inter-socket communication between two applications is exactly the same, whatever may be the application and OS used. Many RTOSes including VxWorks provides inbuilt APIs for TCP/IP stack and several other network protocols.

The coding given here in Example code 11.2 is already a part of VxWorks network APIs. VxWorks provides the APIs (Application Interfaces) for networking. It has a library, *sockLib* for socket programming. [Section 7.15]. [The reader may refer to VxWorks Network Programmer's Guide when using these APIs and *sockLib*. The guide's source is Wind River Systems (www.windriver.com)].

Since our objective is to learn the use of IPCs in multitasking RTOS through a case study, these APIs and *sockLib* are not used here for creating a TCP stack. The present case study is given in order to learn from the example of TCP and UDP as to how a VxWorks RTOS programming environment can be used to develop APIs for an embedded system for a new networking protocol.

Assume a case of an embedded system in which the RTOS tasks communicate the TCP stacks from an application. An example of the application is HTTP or FTP. The latter communicates to a web server or transfers a file, respectively. A TCP/IP stack is a stack containing the frames communicated on the network using a TCP/IP suite of protocols. A reader may refer to the book *Internet and Web Technologies* by this

author from Tata McGraw-Hill, 7th Reprint, 2007. It describes bit-wise formats of TCP segment, UDP datagram, IP packet, and SLIP and Ethernet frame. The application-layer bytestreams are formatted at the successive layers to obtain the final stack for the network. A stream format is as per protocol specifications for in-between layers. A TCP stack typically has many frames into which a network driver writes the bytes.

Example 11.2 shows the application of RTOS for writing into the stack. We use VxWorks functions described in Section 9.3 for the present case. At another node (end point of the network), the priorities of the tasks that receive the stack to retrieve the bytes put in by the application will obviously be in reverse order. Coding for that is left as an exercise for the reader.

The advantages of multitasking should become obvious by this example. An application may not output the strings continuously. The tasks at the other layers can therefore also process concurrently during the intermediate periods.

The semaphores provide an efficient way of synchronizing the tasks of various priorities. The semaphores are used as a mutex guard for the shared data problems when using the global variables for the I/O streams. The semaphores are also used as event flags. The use of VxWorks or any other tested RTOS simplifies the coding as scheduling and IPC functions are now readily available at the RTOS.

## 11.3.1 Requirements

Figure 11.12(a) shows the requirements of a sub-system for application, which is transmitting a TCP/IP stack. Figure 11.12(b) shows the scheduling sequences of the tasks during a TCP/IP stack transmission.

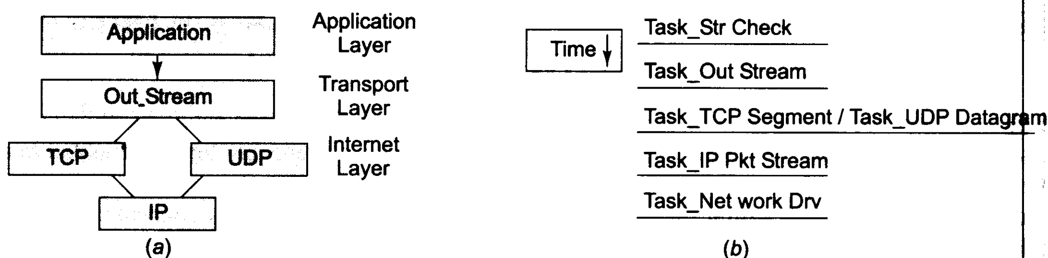



**Fig. 11.12** (*a*) Sub-system for transmitting a TCP/IP stack from an application (*b*) The tasks and their scheduling sequence during a TCP/IP stack transmission

Requirements for creating a TCP/IP stack can be understood through a requirement table given in Table 11.4.

## 11.3.2 Class Diagram, Classes and Objects

TCP stack transmission can be modeled by a class diagram of abstract classes TCP_Stack. Figure 11.13 shows class diagram of a TCP_Stack for sending a TCP or UDP packet to a socket. The diagram shows how the classes and objects of a class relate and also the hierarchical associations during the creation of TCP or UDP data stream and packets transmission. The task objects are the processes or threads that are scheduled by the RTOS.

1. Task_TCP is an abstract class from which extended class(es) is derived to create TCP or UDP packet to a socket. The task objects are instances of the classes (i) Task_StrCheck, (ii) Task_OutStream, (iii) Task_TCPSegment or Task_UDPDatagram, (iv) Task_IPPktStream (v) Task_NetworkDrv.
2. Task_StrCheck is to check and get the string.
3. Task_OutStream extends from the two classes Task_StrCheck and Task_TCP.

**Table 11.4 Requirements for creating a TCP/IP stack**

| *Requirement* | *Description* |
|---|---|
| Purpose | To generate a bytestream for sending on network using TCP or UDP protocol at transport layer and IP protocol at network layer |
| Inputs | 1. Bytes from application layer<br>2. Notification SemTCPFlag and SemUDPFlag for selecting UDP socket or TCP socket, respectively |
| Signals, events and notifications | After forming the packets at IP layer, SemPktFlag for network driver task |
| Outputs | 1. TCP or UDP bytestream to destination socket |
| Functions of the system | An HTTP application data is to be sent after encoding headers at transport and network layers. Tasks are scheduled in five sequences (i) Task_StrCheck, (ii)Task_OutStream, (iii) Task_TCPSegment or Task_UDPDatagram, (iv) Task_IPPktStream (v) Task_NetworkDrv |
| Test and validation conditions | 1. A loop back from the destination socket should enable retrieval of application data stream as originally sent<br>2. Buffer Memory overflow tests |

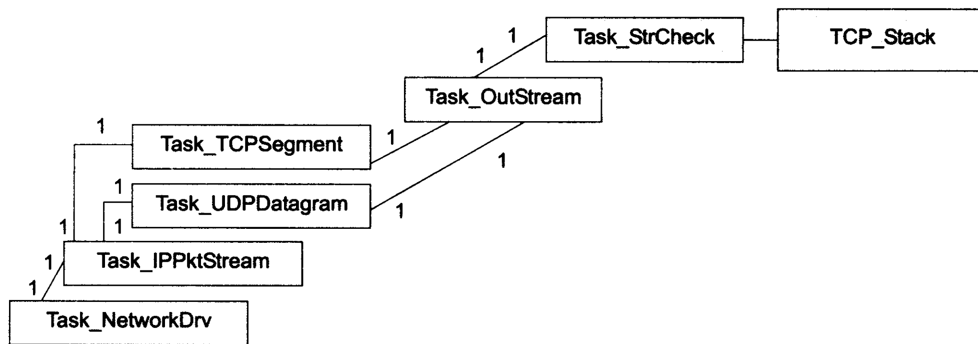



**Fig. 11.13** Class diagram of TCP_Stack for sending TCP or UDP packet to a socket

4. Task_TCPSegment creates a stream from TCP segment for IP layer. When datagram is to be sent then Task_UDPDatagram creates a stream using from Task_OutStream output bytes.
5. Task_IPPktStream creates a packet using the stream Task_TCPSegment or Task_UDPDatagram. It depends whether the application layer object posts SemTCPFlag or SemUDPFlag.
6. Task_NetworkDrv outputs the packets on the network.

***Class*** The left-hand side of Figure 11.14 shows an example of class Task_OutStream. It demonstrates how these two classes are shown using UML.

Task_OutStream has the following fields: (i) semFlag and SemMKey are two semaphores for IPC functions, (ii) three integers for the numBytes, (for number of bytes to be transmitted from application layer), task ID and task priority, respectively, and text lines 1, 2 and 3, (iii) OutStreamInputID and applStr static strings for the out stream ID and application data stream.

Task_OutStream has three methods (functions in C)—msgQSend ( ) posts message string into a message queue, TaskDelay ( ) delays the task and enables a low priority task to start, and TaskResume ( ) resumes the delayed task.

***Object*** The right-hand side of Figure 11.14 is an example of an object based on classes Task_OutStream of which that object is an instance and functional entity. An object is shown by a rectangular box with the object identity followed by semicolon and then the class identity. Object Task_OutStreamAppl is an instance of Task_OutStream.

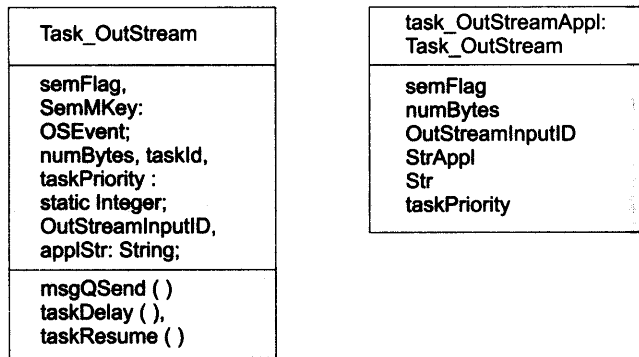



**Fig. 11.14** Class Task_OutStream and Object Task_OutStream

***State Diagram*** Figure 11.15 shows a state diagram for Task_Stack. It demonstrates how UML state diagram is shown. A state diagram shows a model of a structure for its start, end, in-between associations through the transitions and shows events labels (or condition) with associated transitions. The state transitions take place between the tasks, Task_GUI, Task_Display, TaskUser_KeypadInput, Task_Communication, Task_ReadPorts, Task_Refund, Task_ExcessRefund, Task_Collect, Task_Deliver and Task_Display. The steps which are used for drawing a state diagram are given in Section 11.3.3.

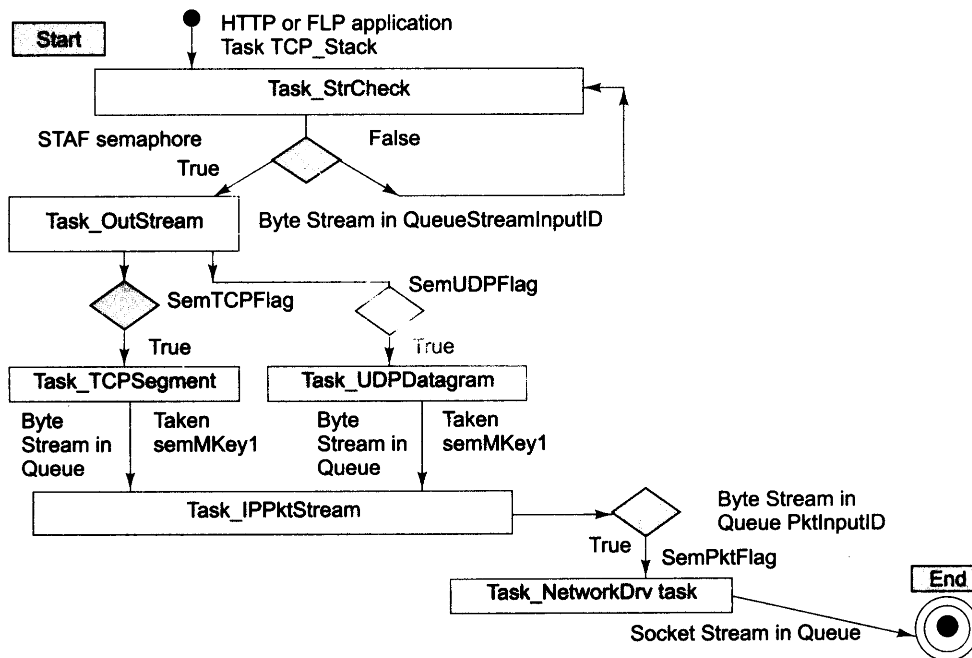



**Fig. 11.15** State diagram for TCP_Stack

## 11.3.3 TCP Stack Hardware and Software Architecture

A TCP stack will run on the same hardware as for the system which is networked with another system. The only additional hardware needed is memory for codes, data and queue for data streams, packets and sockets. A single TCP packet or UDP datagram is of maximum $2^{16}$ bytes. 2 MB ($512 \times 2^{16}$ bytes) RAM can be taken as additional memory requirement.

***Software Architecture*** The left-hand side of Figure 11.16 shows software architeure for a TCP_Stack. The right-hand side shows software model for a TCP/IP stack. Tasks of TCP_Stack and their priorities are defined as follows:

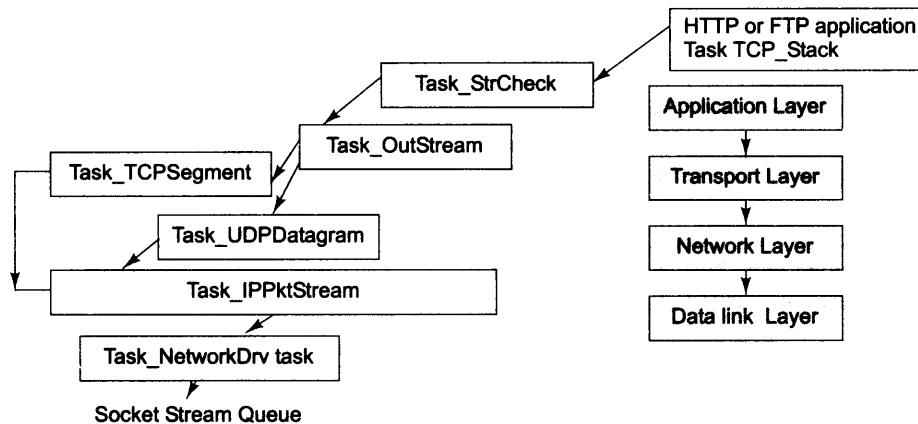



Fig. 11.16 TCP_Stack software architecture and TCP/IP model

**Creating a List of Tasks, Functions and IPCs**

1. VxWorks user-task priorities are defined above 100 and in layer-wise sequence.
2. Let us set an IPC naming convention. An IPC starting with initial characters as 'Sem' means binary semaphore and 'SemM' means mutex. An IPC for flagging an event will have four characters, 'Flag'. An IPC for resource locking has three characters, 'Key'. A message queue identifier begins with the character 'Q'. A pipe identifier begins with four characters 'pipe'.
3. Either *SemTCPFlag* or *SemUDPFlag* is given by the application task. It is as per the protocol to be employed at the transmission control layer. The IPC *SemFinishFlag* is to be taken by the application or any other task. It flags the intimation that the task of the message strings put into the pipe by the network driver is finished. It is an acknowledgement. Another IPC SemFlag2 is to be taken by OutStream as an acknowledgement that the stream sent to the driver has been successfully put into the pipe and the new stream can be sent in the output.
4. TCP header differs from UDP header. The former facilitates the connection establishment, network flow control and management, and connection termination by proper exchanges of parameters through the network. TCP is thus called a connection-oriented protocol. The latter is a simple datagram message to the receiver. UDP is thus a connection-less protocol. The task priorities in both cases are assigned equal. Any of the flags, which specifies protocol, can be given (posted) by the application.
5. A datagram is an independent (unconnected to the previous or next) message stream of maximum size $2^{16}$ bytes. UDP conveys only the source and destination port numbers, stream length and checksum to

the 'internet' layer. The socket stream (output of this layer) has the source and destination port numbers as well as IP addresses. A packet from a socket is a message stream with a of maximum size of $2^{16}$ bytes. Each packet has an inserted IP header. The socket is identified by its IP address and port number.

6. The network driver forms the frame as per network-driver configuration.

Table 11.5 gives a list of tasks.

***Synchronization Model for Multiple Tasks and Their Functions*** Figure 11.17 gives a synchronization model to show how the tasks can be synchronized. The top layer is an *application layer* in a TCP/IP network. Steps from the top layer are as follows:

*Step A:* Let a task, **Task_StrCheck,** *check* for availability of a string at output from an application. This is the highest priority task during sending to the net. If *check* shows string availability through a status flag semaphore, *STAF,* the task gives (posts) a semaphore to a waiting task, **Task_OutStream**. Let the maximum size of the stream be *maxSizeOutStream.*

*Step B:* The waiting task unblocks on taking the semaphore. It then reads a string and sends a byte stream into a message queue. Let the SemFlag2 be the semaphore taken before sending the string from application into the queue for the buffer, *OutStream.*

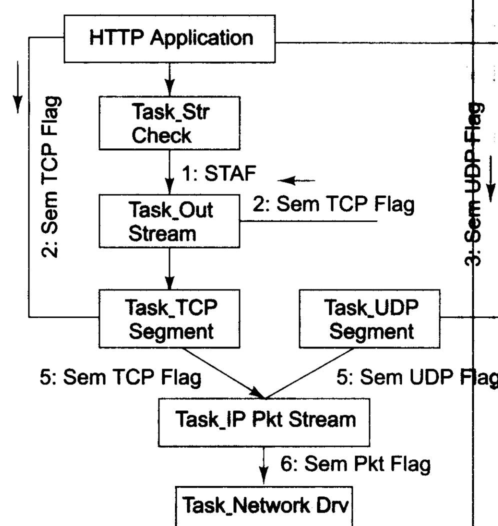



**Fig. 11.17** Model for how the tasks are being synchronized

Let semaphore given by Task_StrCheck and taken by Task_OutStream be *SemFlag1*. Let the second task use the message queue, *OutStream,* for sending the strings from the application to the next waiting task. Let the outstream give bytes to the message queue identified by *QStreamInputID.*

*Step C:* Next to the application layer, there is a transmission control layer in the network. It is the equivalent of *transport layer* in the OSI model. The task is next in priority. The application task posts two other semaphore flags, besides *STAF*. One flag is to unblock a waiting task, **Task_TCPSegment**. It takes the semaphore and unblocks if this layer protocol is TCP. The other semaphore flag is to unblock another waiting task, **Task_UDPDatagram**. It unblocks when this layer protocol is UDP.

*Step D:* Appropriate headers must be inserted at the front of the stream at the *transport layer*, and the stream formats into either a TCP segment or UDP datagram, depending on which task unblocks (undergoes to the running state), Task_TCPSegment or Task_UDPDatagram. The task puts the *blocks,* each of 256 bytes, *blkSize* = 256 bytes, into a queue. This is necessary because if bytes are put into the queue, a time called context-switching time for inter-task through RTOS will be added to the overheads. The overall execution time will increase. Thus interrupt latencies also increase. [Refer to Section 4.6.]

*Step E:* A TCP segment or UDP datagram is too long compared to a *block*. The block size has to be optimised later, during a simulation run of the codes. A bigger block size lets another task wait till a block is ready for sending. A smaller block size, on the other hand, increases the task-switching overheads and the interrupt latencies. This task has a critical section. Header bytes are inserted into this section at the queue front and no bytes should be inserted by any other task at this stage. A mutex, *SemMKey1,* protects the section by resource

locking. Let semaphores taken by TCP task and taken by UDP task be *SemTCPFlag* and *SemUDPFlag*, respectively. Let the task give the blocks to message queue, identified by *QStreamInputID*.

**Table 11.5** List of Tasks, Functions and IPCs used in Example 11.2

| *Task Function* | *Priority* | *Layer in TCP/IP*[@] | *Action* | *Taken IPC* | *Posted IPC* | *Output Byte Stream* |
|---|---|---|---|---|---|---|
| Task_Str-Check | 120 | Application | Get a string from the application | – | SemFlag1 | None |
| Task_Out-Stream | 121 | Application | Read string and put bytes into an output stream | SemFlag1, SemFlag2 | *QStream-InputID* | Out-Stream |
| Task_TCP-Segment | 122 | Transmission Control (Transport) | Insert TCP header to the stream | *SemTCPFlag, SemMKey1, QStreamInputID* | *QStream InputID and SemMKey1,* | Out-Stream |
| Task_UDP Datagram | 122 | Transmission control (Transport) | Insert UDP header to the stream sent as a datagram | *SemUDPFlag, SemMKey1 QStreamInputID* | *QStream InputID, SemMKey1* | Out-Stream |
| Task_IPPkt Stream | 123 | internet (Network) | Form the packets of $2^{16}$ bytes | *SemMKey1, QStreamInputID* | *SemMKey1, SemPktFlag, QPktInputID* | *socket-Stream* |
| Task_Net-workDrv | 124 | Network Interface (Data-link) | Send the packets as the frames | *SemPktFlag, QPktInputID, SemMKey1* | SemFinish-Flag, SemFlag2 | *pipeNet Stream* |

[@] Corresponding layer name in OSI model is given in the bracket.

*Step F:* Next to the transport layer is the 'internet' layer. [It is a small 'i', not a capital, in internet. We are referring to inter-networking, not the Internet.] It is the equivalent of the network layer in the OSI model. A task, **Task_IPPktStream**, waits and unblocks on availability of a block from Task_TCPSegment or Task_UDPDatagram. Each packet has a maximum size of $2^{16}$ bytes. This task is next in priority. **Task_IPPktStream** is for forming an IP packet for transmission on the network through the network driver. This task inserts (writes) an IP header into a socket stream, *SocketStream,* to a network socket. The task inserts the header into the stream after the blocks received from the upper layer stack together in the packet. This task also sends a semaphore flag when the packet is ready for delivery to the network driver. Let the semaphore given by the task and taken later by Task_NetworkDrv be *SemPktFlag*. Let the task send the packets to a message queue, identified by *QPktInputID.*

*Step G:* Next to the internet layer is the 'network interface' layer. It is the equivalent of the data-link layer in the OSI model. Network driver task, **Task_NetworkDrv**, waits for the packet ready flag, *SemPktFlag*. When the task is unblocking, it reads *SocketStream* and writes the frame, *frame* into a pipe, *pipeNetStream.* The pipe stores and transmits the bytes at the frame header, *frameHeader*, at the *SocketStream* data or its fragment and at the trailing bytes, *trailBytes*. The latter are usually for error control functions or frame terminal (end) functions. Let the task give the *frame,* one by one, to a pipe identified by *pipeNetStream*. Let the semaphore given by the task and taken later by the application task be *SemFinishFlag,* when no more bytes are available

for transmission. A semaphore *SemFlag2* is given by the task when a block of byte is ready. This lets the next layer task unblock outStream and initiate the formation of packets whenever the RTOS schedules it next.

SLIP, PPP, Ethernet, and Token ring are examples of interface layer protocols. The *frameHeader* and *trailBytes* are as per the protocol used by the driver. Certain protocols do not write any *frameHeader*, for example SLIP. Certain protocols do not write *trailBytes*.

Task_NetworkDrv opens a configuration file. The configuration file is a virtual file device in the embedded system. [Virtual file device means a file not on the disk but in the memory and using similar open, read, write and close functions as for the files on the disk]. The file points to the configuration information. The function creat ( )
[*Note:* missing 'e' in creat function] and remove ( ) are used to create and remove a file device in VxWorks. VxWorks open, read and write functions are used as shown earlier in Example 9.26. The use of the select ( ) and close ( ) functions for a file is analogous to that for a pipe [Section 9.3.4.12].

Exemplary configuration information in the file can be as follows: We can define, in the case of a serial link, the following parameters:

(i) Protocol for the link (SLIP or PPP)
(ii) Host
(iii) Port
(iv) Baud rate
(vi) Number of bits per character, for Example, 8
(vii) Number of stop bits, for Example, 1

A file can describe configuration as follows, in the case of an Ethernet card used as a network driver. The first line format may begin with *net*. This specifies that the line be for specifying the network card addresses. The next three words are then '*Ethernet* 3COM 0xXYZ'. This specifies that the network is Ethernet. The card is made by 3COM. The card address at the system is hexadecimal XYZ. The next line format may begin with IP. This specifies that the IP address should be defined at the line as the next word. The address is of the node that connects to the network. If another connection exists with the same network driver, there may be another line to assign another IP address. [An IP address is of 4 bytes. It is 0xFFFFFFFF for a broadcasting connection to all nodes. It can also be conventionally written as 255.255.255.255].

A programmer designing the codes first prepares a list of tasks, the task priorities, the layers at which the tasks function, actions by the tasks, IPCs for which each task or section waits (takes) before unblocking, IPC which it gives (posts) to let another waiting section or task unblock and the output stream, which the task sends as output. Table 11.5 lists these. Then the coding is done. The following points are to be noted from the table.

### 11.3.4 Exemplary Coding Steps

The task objects in Figure 11.16 can be implemented as C functions in a VxWorks environment with C as the programming language. VxWorks IPC functions can implement the synchronization model shown in Figure 11.15. Following are the VxWorks coding steps for the task objects listed in Table 11.5.

#### Example 11.2

```
1. # include "vxWorks.h" /* Include VxWorks functions. */
# include "semLib.h" /* Include semaphore functions library. */
# include "taskLib.h" /* Include multitasking functions library. */
# include "msgQLib.h" /* Include message queue functions library. */
```

```
# include "fioLib.h" /* Include file-device input-output functions library. */
# include "sysLib.c" /* Include system library for system functions. */
pipeDrv ( ); /* Install a pipe driver. */
# include "netDrvConfig.txt" /* Include network driver configuration file for frame formatting protocol (SLIP, PPP, Ethernet) description, card description/make, address at the system, IP addresss of the nodes that drive the card for transmitting or receiving from the network. */
# include "prctlHandlers.c" /* Include file for the codes for handling and actions as per the protocols used for driving streams to the network. */
2. sysClkRateSet (1000); /* Set system clock rate 1000 ticks per second. */
3. /* Initialise the socket parameters and other network parameters initial values*/
/* SourcePort means source port number of the application used. DestnPort means destination port number. Define a variable string, Str. */
unsigned short SourcePort; unsigned short DestnPort; unsigned char [ ] Str;
unsigned short SourcePort = ;
unsigned short DestnPort = ;
unsigned short SourceIPAddr = ;
unsigned short DestnIPAddr = ;
4. /* Declare data types of Output Byte Streams for arguments in the tasks. */
unsigned char [ ] applStr, OutStream, socketStream, pipeNetStream;
5. /* Declare data types of the maximum sizes of streams from and to the tasks. Declare data type of block size, blkSize. It is the number of bytes that must be available first before sending an IPC to a buffering stream. It avoids repeated switching from one task to the next after each byte. */
unsigned int blkSize, strSize, strSize, maxSizeOutStream, maxSizeSocketStream, maxSizepipeNetStream;
6. /* Allocate Default Values to various sizes*/
maxSizeOutStream = 1024 * 1024; /* Let application put 1 MB on the net. */
unsigned int strSize = 1; /* Let default string size from an application be 1 byte. */
blkSize = 256; /* Let default block be 256 bytes. */

maxSizeSocketStream = 64 * 1024; /* Let Socket put 64 kB packet on the net. */
maxSizepipeNetStream = 16 * maxSizeSocketStream; /* Let network driver put 16 packets = 1 MB maximum number of bytes. */
7. /* Declare all Table 11.5 Task function prototypes. */
void Task_StrCheck (SemID SemFlag1); /*Task check for the string Availability. */

void Task_OutStream (SEM_ID SemFlag1, SEM_ID SemFlag2, MSG_Q_ID QStreamInputID);
void Task_TCPSegment (SEM_ID SemTCPFlag, SEM_ID SemMKey1, MSG_Q_ID QStreamInputID);
void Task_UDPDatagram (SEM_ID SemUDPFlag, SEM_ID SemMKey1, MSG_Q_ID QStreamInputID, OutStream, SourcePort, DestnPort, maxSizeOutStream, blkSize);
void Task_IPPktStream (SEM_ID SemMKey1, SEM_ID SemPktFlag, MSG_Q_ID QPktInputID);
void Task_NetworkDrv (SEM_ID SemMKey1, SEM_ID SemPktFlag, SEM_ID SemFinishFlag, SEM_ID SemFlag2, MSG_Q_ID QPktInputID socketStream, pipeNetStream, blkSize, maxSizeSocketStream, maxSizepipeNetStream, MSG_FRAME aFrame);
8. /* Declare all Table 11.5 Task IDs, Priorities, Options and Stacksize. Let initial ID till spawned be none. No options and stacksize = 4096 for each of six tasks. */
int Task_StrCheckID = ERROR; int Task_StrCheckPriority = 120; int Task_StrCheckOptions = 0; int
```

Task_StrCheckStackSize = 4096;
int Task_OutStreamID = ERROR; int Task_OutStreamPriority = 121; int Task_OutStreamOptions = 0;
int Task_OutStreamStackSize = 4096;
int Task_TCPSegmentID = ERROR; int Task_TCPSegmentPriority = 122;
int Task_TCPSegmentOptions = 0; int Task_TCPSegmentStackSize = 4096;
int Task_UDPDatagramID = ERROR; int Task_UDPDatagramPriority = 122;
int Task_UDPDatagramOptions = 0; int Task_UDPDatagramStackSize = 4096;
int Task_IPPktStreamID = ERROR; int Task_IPPktStreamPriority = 123;
int Task_IPPktStreamOptions = 0; int Task_IPPktStreamStackSize = 4096;
int Task_NetworkDrvID = ERROR; int Task_NetworkDrvPriority = 124;
int Task_NetworkDrvOptions = 0; int Task_NetworkDrvStackSize = 4096;
9. /* ***Create and Initiate (Spawn) all the six tasks*** of Table 11.5. */
*Task_StrCheckID* = taskSpawn (" tTask_StrCheck", Task_StrCheckPriority,
Task_StrCheckOptions,
Task_StrCheckStackSize, void (*Task_StrCheck) (SEM_ID STAF, SEM_ID SemFlag1), 0, 0, 0, 0, 0, 0,
0, 0, 0, 0);
*Task_OutStreamID* = taskSpawn (" tTask_OutStream", Task_OutStreamPriority, Task_OutStreamOptions,
Task_OutStreamStackSize, void (*Task_OutStream) (SEM_ID *SemFlag1*, MSG_Q_ID QStreamInputID,
applStr, OutStream, maxSizeOutStream, blkSize), 0, 0, 0, 0, 0, 0, 0, 0, 0, 0);
*Task_TCPSegmentID* = taskSpawn (" tTask_TCPSegment", Task_TCPSegmentPriority,
Task_TCPSegmentOptions, Task_TCPSegmentStackSize, void (*Task_TCPSegment) (SEM_ID
SemTCPFlag, SEM_ID SemMKey1, MSG_Q_ID QStreamInputID, unsigned char [ ] OutStream, int
maxSizeOutStream, int blkSize, unsigned char [16] *txtcpState*, unsigned char [16] *txtcpOpPdFormat*)
unsigned short *SourcePort*, unsigned *DestnPort*, unsigned int *SequenNum*, unsigned int *AckNum*, unsigned
char * *TCPHdrLen*, *unsigned *TCPHdrFlags*, unsigned short **TCPChecksum16*, unsigned short *window*,
unsigned short **UrgPtr*, unsigned char [optPdLen] *extras*)
*Task_UDPDatagramID* = taskSpawn (" tTask_UDPDatagram", Task_UDPDatagramPriority,
Task_UDPDatagramOptions, Task_UDPDatagramStackSize, void (*Task_UDPDatagram) (SEM_ID
SemUDPFlag, SEM_ID SemMKey1, MSG_Q_ID QStreamInputID, unsigned char [ ] OutStream, int
maxSizeOutStream, int blkSize), SourcePort, DestnPort, 0, 0, 0, 0, 0, 0, 0, 0);
*Task_IPPktStreamID* = taskSpawn ("tTask_IPPktStream", Task_IPPktStreamPriority,
Task_IPPktStreamOptions, Task_IPPktStreamStackSize, void (*Task_IPPktStream) (SEM_ID SemPktFlag,
MSG_Q_ID QPktInputID, unsigned char [ ] OutStream, int maxSizeOutStream, blkSize, unsigned char
[ ] SocketStream, maxSizeSocketStream), 0, 0, 0, 0, 0, 0, 0, 0, 0, 0);
***Task_NetworkDrvID*** = taskSpawn ("tTask_NetworkDrv", Task_NetworkDrvPriority, Task_
NetworkDrvOptions, Task_NetworkDrvStackSize, void (* Task_NetworkDrv) (SEM_ID SemMKey1,
SEM_ID SemPktFlag, SEM_ID SemFinishFlag, SEM_ID SemFlag2, MSG_Q_ID QPktInputID socketStream,
unsigned char [ ] pipeNetStream, int blkSize, int maxSizeSocketStream, int maxSizepipeNetStream, unsigned
char [ ] frameHeader, unsigned char [ ] trailBytes), 0, 0, 0, 0, 0, 0, 0, 0, 0, 0);
10. /* Declare IDs and create the binary semaphore flags, keys and message queues. */
SEM_ID SemTCPFlag, SemUDPFlag; /* Declared at the application */
*SemTCPFlag* = **semBCreate** (SEM_Q_FIFO, SEM_EMPTY); /* Declared at the application */
*SemUDPFlag* = **semBCreate** (SEM_Q_FIFO, SEM_EMPTY); /* Declared at the application */
/* Note: The application posts wither SemUDPFlag or SemTCPFlag to let one of the two tasks run.

```
[Task_TCPSegment or Task_UDPDatagram. */
SEM_ID SemFlag1, SemFlag2, SemPktFlag, SemFinishFlag, SemMKey1; /* Declared for the six tasks listed in Table 11.5. */
11. /* Create the binary semaphores, message queue for a stream of bytes from an application and pass the options selected to it. */
SemFlag1 = semBCreate (SEM_Q_PRIORITY, SEM_EMPTY); / *Task higher in priority takes it first. */
SemFlag2 = semBCreate (SEM_Q_PRIORITY, SEM_EMPTY); / *Task higher in priority takes it first. */
SemPktFlag = semBCreate (SEM_Q_PRIORITY, SEM_EMPTY); /*Task higher in priority takes it first.*/
SemMKey1 = semBCreate (SEM_Q_FIFO, SEM_EMPTY); / *Taken in FIFO */
SemFinishFlag = semBCreate (SEM_Q_FIFO, SEM_EMPTY); / *Taken in FIFO */
MSG_Q_ID QStreamInputID;
void char [ ] msgStream; /* Pointer for the message buffer */
12. /* Create the message queue identity and pass the parameters and chosen options to it. Let maximum number of messages be 256 kB and the message be of 1 byte each. An IP packet has a maximum 2^16 bytes. Assume that a TCP segment stream for transmitting has maximum of 256 kB. Let 64 bytes be additionally assigned for the headers. Header bytes add at the lower layers (refer column 3 in Table 11.5). */
QStreamInputID = msgQCreate (maxSizeOutStream, strSize, MSG_Q_FIFO | MSG_PRI_NORMAL);
QPktInputID = msgQCreate (maxSizeSocketStream, strSize, MSG_Q_FIFO | MSG_PRI_NORMAL);
13. /* Steps 1 to 3 as per Example 7.21 for creating a pipe. */
.
.
14. /* Create pipe named as pipeNetStream for including overheads and frame messages, each if 1 byte. Overheads mean header bytes as well as trailing bytes. */
STATUS pipestatus;
pipestatus = pipeDevCreate ("/pipe/pipeNetStream", maxSizepipeNetStream, 1);
mode = 0 = 0x0;
15. /* Other declarations that are needed. */
.
.
16. /* Declare common functions needed in the networking tasks. */
/* Declare the functions to get 32-bit, 16-bit and 8-bit lengths from a stream or string. */
unsigned int getLength32 (unsigned char [ ] Str) {
/* Codes for finding as an unsigned integer the length of a string or stream up to 2^32 bytes * /
.
.
};
unsigned short getLength16 (unsigned char [ ] Str) {
/* Codes for finding, as an unsigned short, the length of a string or stream up to 2^16 bytes. * /
.
.
};
unsigned byte getLength8 (unsigned char [ ] Str) {
/* Codes for finding, as an unsigned byte, the length of a string up to 256 bytes. * /
.
.
```

```
};
17. /* Declare Codes for adding extra padding in shorter size message or string or stream to fill
0s so that string size now equals block size, blkSize. */
unsigned char [ ] StrAddPadding (unsigned char [ ] Str, unsigned int blkSize) {
.
.
};
18. /* Exemplary codes for finding a function to get 16-bit checksum from a byte
stream or string. */
unsigned short checksum16 (unsigned char [ ] Str) {
/* Codes for finding, as an unsigned integer, the length of a byte stream or string* /
/* StrPtr is pointer to the string or stream. For the while loop, 'i' is a 16-bit integer. Number of carries
generated = numCarry. */
static unsigned short *strPtr, i, numCarry;
static unsigned int sum = 0;
unsigned short num = (unsigned short) getLength32 (unsigned char [ ] Str); /* num is 16-bit number for
total number of bytes in the string. We are typecasting the length data type to 16-bit */
/* A 16-bit Checksum method is as follows: Sum the 16-bit words and first count the carries, which
generate on successive additions. Then add these carries to get the 16-bit checksum. However, we are
having the byte stream. So method is as follows: */
strPtr = (unsigned short) Str; / Type cast address to 16-bit value pointer. */
i = num/2; /* Let us split the calculation into two parts because we are adding the bytes instead of 16-bit
words to get the checksum*/
while (i --) {
sum += *strPtr ++; /* add the byte into the sum and then pointer to next byte. */
if (i & 1){sum += *(unsigned char *) strPtr;}; /* Sum the odd byte from next address. */
while ((numCarry = (unsigned short) (sum >>16)) ! = 0) {sum = (sum & 0xFFFF) + numCarry;}
return ((unsigned short) sum); / Type cast sum to 16-bit. */
};
/* Reader to write 32-bit checksum codes for function checksum32 by himself (or herself). */
.
.
/* Declare a Function for Cycle Redundancy Check */
unsigned int CRC32 (unsigned int crc, unsigned char aByte) {/* Codes for finding 32-bit cycle redundancy
check bits as an unsigned integer for the given message frame. * /
.
.
};
19. /* On a network, while we are sending the bytes a protocol uses the different data types, such as
unsigned int, unsigned short, unsigned char, etc. Hence, the definitions of the following six functions to get
the lower byte and higher byte from a 16-bit short, data16 and to get four bytes byte0, byte1, byte2 and
byte3 from a 32-bit int, data32 is essential. */
unsigned char LByte (unsigned short data16) { ... }; unsigned char HByte (unsigned short data16)
{ ... };
```

```
unsigned char byte0 (unsigned int data32) { ... }; unsigned char byte1 (unsigned int data32)
{ ...};
unsigned char byte2 (unsigned int data32) { ... }; unsigned char byte3 (unsigned int data32)
{ ...};
/*------------------------------------------------------------------------------------------------*/
20. to 27./* Code for other declaration steps specific to the various networks */
.
.

/* End of the codes for creation of the tasks, semaphores, message queue, pipe tasks, and variables and all
needed function declarations */
/****************************************************************************************/
28. /* Start of Codes for Task_StrCheck*/
void Task_StrCheck (SEM_ID STAF, SEM_ID SemFlag1) {
.
.
29.
while (1) { /* Start of while loop. */
/* When character output is generated by the application, the semaphore is given. */
SemTake (STAF, WAIT_FOREVER); /* Wait for Status flag from Application. */
30. /* Codes for the task. *
.
.
semGive (SemFlag1);
semGive (SemMKey1); /* Release the mutex if any taken before. */
taskDelay (10); /* Delay 10 ms to let lower priority task run. */
}; /* End of while loop. */

} /* End of Task_StrCheck. */
31. /* Start of Codes for Task_OutStream. */
void Task_OutStream (SEM_ID SemFlag1, MSG_Q_ID QStreamInputID, applStr, OutStream,
maxSizeOutStream, blkSize) {
32. /* Initial assignments of the variables and pre-infinite loop statements that execute once only*/
int numBytes;
.
.
33. while (1) { /* Start an infinite while-loop. We can also use FOREVER in place of while (1). */
34. /*Wait for SemFlag1 state change to SEM_FULL by semGive function for string availability check
task */
semTake (SemFlag1, WAIT_FOREVER);
35. /* Take the key to not let any application other than the present task put the message into the
queue port decipher task that needs SemMKeyID run */
semTake (SemMKey1, WAIT_FOREVER); /* SemMKeyID is now not available and the critical
region starts */
```

```
36. /* Codes for Encrypting applStr if any or illegal character or message check, if any */
.
.
37. /* At the end, send the byte input at OutStream as message to queue, QStreamInputID. Refer to
Section 9.3.4. for understanding the msgQSend function. NO_WAIT if string is more than the block size
256 bytes. Otherwise string is too short to deserve sending on the stream without padding it with extra 0s.
Call a function to add padding bytes. */
numBytes = getLength32 (applStr);
if (numBytes < blkSize) {StrAddPadding (applStr, blkSize); numBytes = blkSize};
/* Now wait if any of the previous bytes of applStr have not been put into the socket streams.
Task_IPPktStream does that. It posts SemFlag2 on successfully sending the applStr into the stream with
two transport and internet layer headers. */
semTake (SemFlag2, WAIT_FOREVER);
msgQSend (QStreamInputID, applStr, numBytes, NO_WAIT, MSG_PRI_NORMAL);};
semGive (SemMKey1); /* Critical Region ends here. */}
38. /* Resume the delayed task, Task_StrCheck as message has been put into the message queue. */
taskDelay (20)
taskResume (Task_StrCheckID);
. /* Other remaining codes for the task. */
.
.
};
39. } /* End of the codes for Task_OutStream. */
40. /* Start of Codes for Task_TCPSegment. */
void Task_TCPSegment (SEM_ID SemTCPFlag, SEM_ID SemMKey1, SEM_ID SemFlag2, MSG_Q_ID
QStreamInputID, OutStream, maxSizeOutStream, blkSize, unsigned char [16] txtcpState, unsigned char
[16] txtcpOpPdFormat) {
41. /* Initial assignments of the variables and pre-infinite loop statements that execute once only*/
static int numBytes; /* Number of bytes successfully read. Equals error on timeout or message queue ID
not identified. */
static int timeout = 20; /* Let after 20 system clock-ticks 20 ms*/
unsigned char [16] txtcpState;
unsigned char [16] txtcpOpPdFormat;
unsigned char [ ] OptionAndPds (unsigned char [16] txtcpState, unsigned char [16] txtcpOpPdFormat);
unsigned char [optPdLen] extras;
unsigned char [ ] Str, unsigned int SequenNum, unsigned int AckNum, unsigned char * TCPHdrLen, unsigned
*TCPHdrFlags, unsigned short window, unsigned short *TCPChecksum16, unsigned short *UrgPtr,
unsigned char [optPdLen] extras);
static unsigned char [ ] header;
unsigned char [ ] TCPHeader (unsigned short SourcePort, unsigned short DestnPort, unsigned char
[16] txtcpState, unsigned char [16] txtcpOpPdFormat, unsigned char [ ] Str, unsigned int
SequenNum, unsigned int AckNum, unsigned char * TCPHdrLen, unsigned *TCPHdrFlags,
unsigned short window, unsigned short *TCPChecksum16, unsigned short *UrgPtr, unsigned
char [optPdLen] extras;
.
.
```

```
42. while (1) { /* Start task infinite loop. */
/* Take mutex so that till header is inserted into the stream, it is not released to
Task_OutStream. */
semTake (SemTCPFlag, WAIT_FOREVER);
semTake (SemMKey1, WAIT_FOREVER); /* SemMKeyID is now not available.
Wait for entering critical region*/
43. /* Receive the message sent by Task_OutStream */
numBytes = msgQReceive (QStreamInputID, OutStream, maxSizeOutStream, 20);
if (numBytes != ERROR) {
44. /* Code for defining the txtcpState as per the connection status. */
.
.
/* Code for defining the SequenNum as per txtcpState */
.
.
/* Code for defining the AckNum as per txtcpState */
.
.
/* Code for defining the window as per txtcpState */
.
.
/* Code for defining the txtcpOpPdFormat as per txtcpState */
.
.
45. /* Codes for finding the additional integers as options and padding to be put as the header. */
extras = OptionAndPds (txtcpState, txtcpOpPdFormat);
46. /* Codes for retrieving the TCP header bytes */
header = TCPHeader (SourcePort, DestnPort, txtcpState, txtcpOpPdFormat, OutStream, SequenNum,
AckNum, * TCPHdrLen, *TCPHdrFlags, window, *TCPChecksum16, *UrgPtr, extras);
47. /* Send header into the front of the queue */
msgQSend (QStreamInputID, header, getLength32 (header), NO_WAIT, MSG_PRI_URGENT);
48. /* Send data to the back of the queue */
msgQSend (QStreamInputID, applStr, numBytes, NO_WAIT, MSG_PRI_NORMAL);
}; /* End of the message handling codes. */
semGive (SemMKey1); /* Critical region ends here. */}
semGive (SemTCPFlag); /* SemTCPFlag. */
taskDelay (20);
taskResume (Task_OutStream);
}; /* End of while loop. */
49. } /* End of codes for Task_TCPSegment */
50. /* Start of codes for Task_UDPDatagram */
void Task_UDPDatagram (SEM_ID SemUDPFlag, SEM_ID SemMKey1, SEM_ID SemFlag2,
MSG_Q_ID QStreamInputID, OutStream, maxSizeOutStream, blkSize) {
51. /* Initial assignments of the variables and pre-infinite loop statements that execute once
only*/
```

```
static int numBytes; /* Number of bytes successfully read. Equals error on timeout or if
message queue ID not identified. */
static int timeout = 20; /* Let after 20 system clock-ticks 20 ms*/
static unsigned char [8] header;
unsigned char [8] UDPHeader (unsigned short SourcePort, unsigned short
DestnPort, unsigned char [ ] OutStream);
.
.
52. while (1) { /* Start task infinite loop. */
/* Take mutex so that till header is inserted into the stream; it is not released to Task_OutStream. */
semTake (SemUDPFlag, WAIT_FOREVER);
semTake (SemMKey1, WAIT_FOREVER); /* SemMKeyID is now not available. Wait for entering the
critical region*/
53. /* Receive the message sent by Task_OutStream */
numBytes = msgQReceive (QStreamInputID, OutStream, maxSizeOutStream, 20);
if (numBytes != ERROR) {
54. /* Codes for retrieving the UDP header bytes */
header = UDPHeader (SourcePort, DestnPort, OutStream); 55. /* Send Header to the front of the queue */
msgQSend (QStreamInputID, header, 8, NO_WAIT, MSG_PRI_URGENT);
56. /* Send Data to the back of the queue */
msgQSend (QStreamInputID, applStr, numBytes, NO_WAIT, MSG_PRI_NORMAL);
}; /* End of the message handling codes. */
/* Critical Region ends here. */
57. semGive (SemMKey1);
semGive (SemUDPFlag); /* SemUDPFlag release for use in new application string, applStr. */
58. /* Let lower priority task, Task_IPPktStream start Higher priority one resume. */
taskDelay (20);
taskResume (Task_OutStream);
}; / * End of while loop. */
59. } /* End of codes for Task_UDPDatagram */
60. /* Start of Codes for Task_IPPktStream */
void Task_IPPktStream (SEM_ID SemFlag2, SEM_ID SemPktFlag, MSG_Q_ID QPktInputID, int
maxSizeOutStream, int blkSize, unsigned short SourceAddr, unsigned short DestnAddr, unsigned short
IPverHdrPrioSer, unsigned char [16] txipState, unsigned char [16] txipOpPdFormat, unsigned char
*timeToLive, unsigned char *PrctlField) {
61. /* Initial assignments of the variables and pre-infinite loop statements that execute once only*/
static int numBytes; /* Number of bytes successfully read. Equals error on timeout or if message queue
ID not identified. */
static int timeout = 20; /* Let after 20 system clock-ticks 20 ms*/
static unsigned char [8] header; unsigned char [ ] Str;
unsigned char [ ] IPHeaderSelPkt (unsigned short SourceAddr, unsigned short DestnAddr,
unsigned short IPverHdrPrioSer, unsigned short *IPPktLen, char * IPVerHdrLen, unsigned
char * IPHdrLen, unsigned char *IPHdrFlags, unsigned short *IPHdrFrag, unsigned short
*IPChecksum16, unsigned short UniqueID, unsigned char *timeToLive, unsigned char
```

```
*PrctlField, unsigned char [ ] Str, unsigned char [ ] OutStream, char [optPdLen] IPextras,
unsigned char [16] txipState, unsigned char [16] txipOpPdFormat);
62. while (1) { /* Start task infinite loop. */
/* Take mutex so that till header is inserted into the stream; it is not released to
Task_OutStream. */
semTake (SemMKey1, WAIT_FOREVER); /* SemMKeyID is now not available. Wait for entering critical
region */
63. /* Receive the message sent by Task_OutStream */
numBytes = msgQReceive (QStreamInputID, OutStream, maxSizeOutStream, 20);
if (numBytes != ERROR) {
64. /* Codes for retrieving the IP Packet header bytes */
header = IPHeaderSelPkt (SourceAddr, DestnAddr, IPverHdrPrioSer, *IPPktLen, * IPVerHdrLen, *
IPHdrLen, *IPHdrFlags, *IPHdrFrag, *IPChecksum16, UniqueID, *timeToLive, *PrctlField, Str,
OutStream, IPextras, txipState, txipOpPdFormat);
65. /* Send Header into the front of the queue */
msgQSend (QPktInputID, header, * IPHdrLen, NO_WAIT, MSG_PRI_URGENT);};
66. /* Send data to the back of the queue */
numBytes = *IPPktLen - * IPHdrLen;
msgQSend (QPktInputID, Str, numBytes, NO_WAIT, MSG_PRI_NORMAL);};
/* Further send header and Str bytes into the queue if more IP packets are to be sent */
.
.
}; /* End of the message handling codes. */
/* Critical section ends here. */}
67. semGive (SemMKey1);
semGive (SemPktFlag); /* SemPktFlag release for use in Network Driver Task. */
68. /* Let lower priority task, Task_IPPktStream start Higher priority one resume. */
taskDelay (20);
taskResume (Task_UDPDatagramID);
taskResume (Task_TCPSegmentID);
}; / * End of while loop. */
69. } /* End of codes for Task_IPPktStream. */
70. /* Start of Codes for Task_NetworkDrv. */
void Task_NetworkDrv (SEM_ID SemMKey1, SEM_ID SemPktFlag, SEM_ID SemFinishFlag, SEM_ID
SemFlag2, MSG_Q_ID QPktInputID socketStream, unsigned char [ ] pipeNetStream, int blkSize, int
maxSizeSocketStream, int maxSizepipeNetStream, unsigned char [ ] frameHeader, unsigned char [ ]
trailBytes) {
71. /* Declare data type for specifying the headers. Let header length of frame be length; type of
network driver be netDrvType; and fragment offset at the stream be frameOffset, which specifies the
index in the byte array from which a frame starts in case the stream is sent in fragments. */
unsigned char [ ] frame, frameHeader, trailBytes, fragment, trailBytes;
unsigned short fragOffset;
unsigned short *FRlength, * HdrLen, * endLen, *FrameHdrFlags, *FrameHdrFragOffset,
*PrctlField;
```

```
unsigned int *FrameCRC32;
unsigned char [ ] Str, unsigned char [ ] SocketStream, unsigned char [ ] FRextras,
unsigned char [16] txFRState, unsigned char [16] txFROpPdFormat, unsigned char
[12] netDrvType
72. /* Declare Network Driver Function. */
void NetHdrFrTr (unsigned char [ ] frame, unsigned char [ ] frameHeader, unsigned char [ ] trailBytes,
unsigned short fragOffset, unsigned char [ ] fragment, unsigned char [ ] trailBytes, unsigned short *FRlength,
unsigned short * HdrLen, unsigned short * endLen, unsigned short *FrameHdrFlags, unsigned short
*FrameHdrFragOffset unsigned short *FrameCRC32, unsigned short *PrctlField, unsigned char [ ] Str,
unsigned char [ ] SocketStream, unsigned char [ ] FRextras, unsigned char [16] txFRState, unsigned char
[16 ] txFROpPdFormat, unsigned char [12] netDrvType);
73. /* Integers for number of bytes successfully read, message size and file device, respectively. */
int numBytes, lBytes, fd;
74. /* Let a pointer netDrvConfig define a pointer to a configuration file for a network driver. */
int fd; /* Define an integer for a file device. */
fd = open (netDrvConfig.txt, O_RDWR, 0); /* Refer Step 5 Example 8.26. */
75. /* Code for reading netDrvType, protocol for the link (SLIP or PPP), data link layer format (say
Ethernet), host IP, port, baudrate, card specifications, etc. Use lBytes and function, read as follows. */
lBytes =12; /* Let the first message bytes read = 12*/
numBytes = read (fd, Str, lBytes);
.
.
close (fd);
75. /* Open the pipe for the Network Stream. Refer Step 5 Example 8.26. */
fd = open ("/pipe/pipeNetStream", O_RDWR, 0);
76. while (1) {;
/* Task reads SocketStream on unblocking and writes the frame, frame into a pipe, pipeNetStream. The
pipe stores and transmits the bytes at the frame header, frameHeader, and frame fragment from the
SocketStream in case frame is of smaller size than the SocketStream and at the end trailing bytes, trailBytes.
The latter are usually for error control functions or frame terminal (end) functions. Let the task give the
frame one after another to a pipe identified by pipeNetStream. This lets the next layer task unblock outStream
and initiate the formation of packets whenever the RTOS schedules it. */
semTake (SemPktFlag, WAIT_FOREVER);
semTake (SemMKey1, WAIT_FOREVER);
77. /* Receive the message sent by Task_IPPktStream */
numBytes = msgQReceive (QStreamInputID, SocketStream, maxSizeOutStream, 20);
if (numBytes != ERROR) {
78. /* Codes for retrieving the frame bytes */
NetHdrFrTr (frame, frameHeader, trailBytes, fragOffset, fragment, trailBytes, *FRlength, * HdrLen,
*endLen, *FrameHdrFlags, *FrameHdrFragOffset, *FrameCRC32, *PrctlField, Str, SocketStream,
FRextras, txFRState, txFROpPdFormat, netDrvType);
/* Write a message, info of lBytes. */
unsigned char [ ] info; /* Let the message be a string of characters. */
int lBytes;
lBytes =12;
```

```
write (fd, frame, *FRlength);
79. /* Further write into the pipe if more frames are to be sent */
.
.
; };
semGive (SemMKey1);
semGive (SemFinishFlag); /* Post the semaphore for next waiting task at the application layer. */
semGive (SemFlag2); /* Post the semaphore for waiting task for sending an Out Stream. */
taskResume (Task_IPPktStreamID); /* Resume the previously delayed higher priority task. */
}; /* End of While-loop */
80. } /* End of codes for Task_NetworkDrv */
/* Start of codes for Task_NetworkDrv. */
/* *********************************************************************/
```

A file *prcltHandlers.c* has the codes for UDP, TCP, IP and network protocols for including and handling the headers functions, selecting the fragments bytes when forming IP packets and frame bytes when forming the frames. The codes are designed as follows:

```
1. /* Declare a Function for returning a UDP header string. DatagramLen means length of UDP datagram, which transmits to the next layer. Str means the stream or string from the application layer to be sent with UDP protocol. */
unsigned char [8] UDPHeader (unsigned short SourcePort, unsigned short DestnPort, unsigned char [ ] Str) {
2. /* Codes for returning UDP Header String. Remember that UDP protocol transmits the integers with big endian. Refer to paragraph (1) in the instructions for a hardware designer in Section 2.1. Big endian means the most significant bytes transmit first. * /
unsigned char [8] UDPHStr;
unsigned short DatagramLen = getLength16 (Str) + 8; /* Add 8 byte header length also. */
unsigned short UDPChecksum16 = checksum16 (Str);
UDPHStr [0] = HByte (SourcePort); UDPHStr [1] = LByte (SourcePort);
UDPHStr [2] = HByte (DestnPort); UDPHStr [3] = LByte (DestnPort);
UDPHStr [4] = HByte (DatagramLen); UDPHStr [5] =LByte (DatagramLen);
UDPHStr [6] = HByte (UDPChecksum16); UDPHStr [7] = LByte (UDPChecksum16);
return (UDPHStr);
};
/* ————————————————————————————————————————— */
3. /* Declarations and codes for the function, TCPHeader for returning a TCP header string. Str means the stream from the application layer at a node, node1 end to be sent with TCP protocol to other end, node2. */
/* Define the byte position (index) of the present OutStream bytes from the application. Initial value = 0*/
unsigned int SequenNum = 0;
/* Define the total number of bytes already received at an input stream from node2 to this node1. Input stream is the one that was sent as TCP stacks and received at the node1 from the receiver node2. This is to let an outstream simultaneously convey to the other end an acknowledgement along with a new sequence of bytes. OutStream and input stream synchronize and there is controlled flow of bytes between the two ends, node1 and node2. Initial value = 0. */
unsigned int AckNum =0;
/* Declare 4-bit and 4-bit unused, reserved for future expansion. TCP headers vary between 5 to 15 unsigned integers of 32-bit each. */
```

unsigned char *TCPHdrLen;
/* Let 16-character string txtcpState specify the state of transmission of the current TCP segment and its action required for the controlled transmission. Action is to be as desired. It may be to establish a connection, for termination or management or flow control. Refer to TCP protocol in any standard text for definitions.
*/
unsigned char [16] txtcpState;
/* Declare TCPHdrFlags for 6 bits for flags and 2 bits unused and reserved for future modifications in protocol. Bit 0 is FIN, bit 1 is SYN, bit 2 is RST, bit 3 is PUSH, bit 4 is ACK and bit 5 is URG. */
unsigned char *TCPHdrFlags;
/* Let another 16-character string txtcpOpPdFormat specify format, which gives the number and meaning of optionally added integers and padded integers. For example, an optional integer may be to specify an alternative window of 32 bits in place of 16 given in the 16-bit window field at fourth integer in the TCP header. Another option integer may specify the TCP maxSizeOutStream. */
unsigned char [16] txtcpOpPdFormat;
4. /* Start of the codes for returning a *short*. It specifies the 4-bit TCP Header length field as well as six-flag fields. These are as per transmission state and transmitting TCP options and padding format, *txtcpState* and *txtcpOpPdFormat*, respectively */
unsigned short **TCPHdlenFlagBits** (unsigned char [16] *txtcpState*, unsigned char [16] *txtcpOpPdFormat*, &TCPHdrFlags, &TCPHdrLen) {
Boolean bit 15, bit 14, bit 13, bit 12, bit 11, bit 10, bit 9, bit 8;
5. /* Codes to have four bits, bit 15, bit 14, bit 13, bit 12 in the returned integer as per the TCP header size, which specifies the total number of unsigned integers in the TCP header. The total is 5 plus the unsigned integers for options and padding. These are as per txtcpState and txtcpOpPdFormat used by TCP segment that is transmitting. The bits, bit 11, bit 10, bit 9, bit 8 are reserved. */
/* Bits 0 to 5 are as per six flags. Bits 6 to 11 are reserved. Bits 0 to 7 are at unsigned character pointer TCPHdrFlags. */
*TCPHdrFlags = *TCPFlags (txtcpState); /* Using the function find bits 0 to 7, */
.
.
6. /* Code to find TCPHdrLen from bit 15, bit 14, bit 13, bit 12. */
.
.
}; /* End of the codes for returning an integer that specifies TCP Header length and flag fields. */
7. /* Declare a function TCPFlags to return TCPHdrFlags as per *txtcpState*. Start of the codes for finding the byte to represent the TCP flag fields. */
unsigned char ***TCPFlags** (unsigned char txtcpState) {
Boolean FIN, SYN, RST, PUSH, ACK, URG, bit 6, bit 7;
/* Codes to create as per txtcpState in which TCP segment is transmitting. */
.
.
}; /* End of the codes for returning the pointer for a byte for the TCP flags field. */
8. /* Declare other TCP Header fields. */
unsigned short *window*; /* node1 specifies by *window* as an advertisement to node2 how many more bytes at *node1* can be buffered in its buffer beyond the ones already buffered and acknowledged by *node1* to

```
node2. Buffering is an intermediate state in which bytes are still to be sent to an upper application layer by
a TCP stack receiving entity. Node 2 if finds window = 0 or too small a number, then it should not send any
thing to this node1. This 16-bit field then is a request to the other end node not to flood data bytes on the
network unnecessarily. It is then indirectly a request to wait. */
unsigned short *TCPChecksum16; /* Define 16-bit Checksum. */
/* Define a 16-bit Urgent pointer. */
unsigned short Urgent;
/ *A 16-bit value that will indicate an urgency to node2 whenever the URG flag is set. It indicates the first
byte of the start of the urgent data from node1. It is a request to the node2 that it should consider the pointed
bytes first in place of attending to the bytes in the buffer and the bytes after this segment header. The
buffered data and beginning data may be ignored and bytes beginning from UrgPtr are requested to be
given urgency. Given or not depends on node2 program task at TCP segment that sends the bytes to its
application layer. */
unsigned short *urgPtr = Urgent;
unsigned short *TCPURGENT (unsigned char txtcpState, &TCPHdrFlags, *urgPtr) {
Boolean URG;
URG = getFlag (&TCPHdrFlags) {
/ * Codes to extract the URG bit. It is bit 5 of byte at address of TCPHdrFlags.
.
   }
/* If URG is true, then set the urgent field and return a 16-bit short. Codes to return a 16-bit short as urgent
pointer as per txtcpState in which TCP segment is transmitting. */
if (URG) {
.
. };
 };
unsigned char [ ] extras; unsigned char OptPdLen;
extras = OptionAndPds ( txtcpState, txtcpOpPdFormat);
optPdLen = getLength8 (extras);
*TCPHdrLen = (optPdLen + 20) /4 ;
unsigned char [ ] OptionAndPds (unsigned char [16] txtcpState, unsigned char [16]
txtcpOpPdFormat) {
9. /* Codes for returning an array of bytes for the 32-bit integers for the options and padding.
These are as per the State and thus TCP header length parameters. */
.
.
   };
10. /* Codes for returning TCP Header String. Remember that TCP protocol transmits
the integers with big endian. It means that the most significant byte should transmit first. * /
/* Note: Instead of using many arguments, a typedef could have been used for the
header data structure. However, this will consume more memory. */
unsigned char [ ] TCPHeader (unsigned short SourcePort, unsigned short DestnPort, unsigned char [16]
txtcpState, unsigned char [16] txtcpOpPdFormat, unsigned char [ ] Str, unsigned int SequenNum, unsigned
int AckNum, unsigned char * TCPHdrLen, unsigned *TCPHdrFlags, unsigned short window, unsigned
short *TCPChecksum16, unsigned short *UrgPtr, unsigned char [optPdLen] extras) {
```

```
int i; unsigned char [ ] TCPHStr; unsigned short lenflag; unsigned char optPdLen;
unsigned short TCPChecksum16 = checksum16 (Str); unsigned short *urg; unsigned char [ ] extras;
*urg = *TCPURGENT (unsigned char txtcpState, &TCPHdrFlags, *urgPtr)
TCPHStr [0] = HByte (SourcePort); TCPHStr [1] = LByte (SourcePort); TCPHStr [2] = HByte (DestnPort);
TCPHStr [3] = LByte (DestnPort); TCPHStr [4] = Byte0 (SequenNum); TCPHStr [5] = Byte1 (SequenNum);
TCPHStr [6] = Byte2 (SequenNum); TCPHStr [7] = Byte3 (SequenNum);
TCPHStr [8] = Byte0 (AckNum); TCPHStr [9] = Byte1 (AckNum); TCPHStr [10] = Byte2 (AckNum);
TCPHStr [11] = Byte3 (AckNum);
*lenflag = * TCPHdlenFlagBits (unsigned char [16] txtcpState, unsigned char [16] txtcpOpPdFormat
&TCPHdrFlags, &TCPHdrLen);
TCPHStr [12] = HByte (lenflag); TCPHStr [13] = LByte (lenflag);
TCPHStr [14] = HByte (window); TCPHStr [15] = Byte3 (window);
TCPHStr [16] = HByte (TCPChecksum16); TCPHStr [17] = LByte (TCPChecksum16);
TCPHStr [18] = *Urg++); TCPHStr [19] = *(unsigned char *) urg);
extras = OptionAndPds (unsigned char [16] txtcpState, unsigned char [16] txtcpOpPdFormat);
optPdLen = getLength8 ( extras);
*TCPHdrLen = (optPdLen + 20) /4;
while (optPdLen > = ++i) {TCPHStr [i] = extras [i – 20];}; /* Fill the options and padding bytes. */
unsigned short TCPSegLen = getLength16 (Str) + (optPdLen + 20); /* Add byte of header length also. */
return (TCPHStr);
};
/* ------------------------------------------------------------------------------------------------ */
```

/* Declarations and codes for the function, ***IPHeaderSelPkt*** for returning an IP header string and the packet data in socketStream from OutStream. *Str* means the stream from transmission control layer at a node, *node1*, end to be sent with header as per IP protocol to the other end, *node2*, through the network driver, switches, bridges and routers. */

11. /* First let us declare IPVerHdrLen for 4 bits (bit 7, bit 6, bit 5 and bit 4) for the version number. Presently, IP version 4 is mostly used. So these bits are 0100. Another four bits (bit 3, bit 2, bit 1 and bit 0) are the numbers of unsigned integers in the header. Header length includes 5 unsigned standard integers and 0 to 10 integers for options and padding. The latter integers are used for controlling the path and flow through the routers. Some of these integers may also be used for adding network security. Options may be recording of the route of packet, time stamp on the packet, source router IP address to be used before routing through a common source, any flexible source option, security option, etc. Details can be found in the classic work of *D. E. Comer and D. Stevens, Internetworking with TCP/IP Vol.1, Principles, Protocols and Architecture from Prentice Hall, NJ, 1995.* Usually, the options and padding are not present. Then the four lower bits are 0101. */

```
unsigned char * IPVerHdrLen;
/* Header Length as per four upper bits in IPVerHdrLen. */
unsigned char *IPHdrLen;
```

12. /* Let us assume that the options and padding are as per the format specified by a 16-characters string, txipOpPdFormat. It represents the number and meanings of optionally added integers and padded integers. */

```
unsigned char [16] txipOpPdFormat;
```

13. /* Let a 16-character string txipState specify the protocol of the current IP segment and its action

required for the controlled transmission. Protocol can be ICMP, IGMP, OSPF, EGP and BGP. Refer to this author's book *Internet and Web Technologies from Tata McGraw-Hill* for definitions and any standard text for details of these protocols. */
unsigned char [16] ***txipState***;
14. /* Start of the codes for returning a *short*. It specifies 4-bit version field plus 4-bit IP Header length field in * IPVerHdrLen followed by 3-bit specifying precedence of the IP packet on the network plus 5 bits for the type of service. Let the latter 8 bits be at the address pointed by IPPrioServ. Precedence bits, '000' means usual precedence on the Internet. '111' means highest precedence, the one needed for streaming the audio and video on the net service bits for quality of service to be provided in terms of security, speed, delayed or cost to be charged from the sender. The 16-bit integer returned by function, IPVerHdrPrioServBits is thus as per version and IP packet transmission state and transmitting IP options and padding format, *txipState* and *txipOpPdFormat*, respectively. */
unsigned char * IPPrioServ;
unsigned short **IPVerHdrPrioServBits** (unsigned char [16] *txipState*, unsigned char [16] *txipOpPdFormat*, &IPPrioServ, &IPHdrLen) {
15. /* Returned integer bit 15, bit 14, bit 13, bit 12, bit 11, bit 10, bit 9 and bit 8 are as per 8 bits at IPHdrLen. Three precedence bits, bit 7, bit 6 and bit 5 in the returned integer are as per the IP precedence and service bits bit 4, bit 3, bit 2, bit 1 and bit 0 are as per QOS (quality of service specified in *txipState*. */
Boolean bit 15, bit 14, bit 13, bit 12, bit 11, bit 10, bit 9, bit 8, bit 7, bit 6, bit 5, bit 4, bit 3, bit 2, bit 1, bit 0;
/* The codes to find character at IPPrioServ from *txipState*. */
.
.
16. /* Code to find IPHdrLen from *txipOpPdFormat* */
.
.
}; /* End of the codes for returning the integer that specifies IP version, Header length, precedence and service. */
17. /* Two 16-bit short integers for packet length and packet identification by receiver. */
unsigned short IPPktLen; /* Specify the length of IP Packet */
unsigned short UniqueID; /* Specify the UniqueID of the present IP Packet This is put by the router or the transmitter, node1. It uniquely identification for the packet routed. This will help the receiver to reassemble the fragments of this IP Packet at node2 for upward transmission to transport and application layer there. *Note*: A fragment may be lost on the net in between routers due to some error popping in. This helps in recovering the lost fragment. */
18/* A stream from OutStream may be bigger than 6536 bytes minus the header bytes in the IP packet. Therefore, it is to be fragmented. Fragmentation is also necessitated when the network driver and other units in-between do not permit even an IP packet of 65536 bytes and need shorter frames. Function, IPFlagFragBits Codes is for returning 3 flag bits and 13 fragment-offset bits. The offset specifies fragment number. Besides precedence and QOS, *txipState* specifies what 3 flag bits and 13 fragment-field bits should be defined by this function. */
unsigned char *IPHdrFlags; unsigned short *IPHdrFrag;
/* Function for finding a stream of bytes actually been put with the packet. */
void **selectPktData** (unsigned char [ ] Str, unsigned char OutStream, unsigned char [16] *txipState*, unsigned char [16] *txipOpPdFormat*, unsigned char [ ] Str) {

.

.

};

unsigned short **IPFlagFragBits** (unsigned char [16] *txipState*, unsigned char [16] *txipOpPdFormat*, &IPHdrFlags, &IPHdrFrag), char [ ] OutStream) {

Boolean bit 15, bit 14, bit 13, bit 12, bit 11, bit 10, bit 9, bit 8, bit 7, bit 6, bit 5, bit 4, bit 3, bit 2, bit 1, bit 0;

19. /* Codes to have three bits, bit 15, bit 14 and bit 13 in the returned integer as per the flags. A flag is *mfb*. mfb = 0 means that the receiver should wait as there are more fragments to follow this fragment. mfb = 0 for last fragment. It refers to the IP header size, which specifies the total number of unsigned integers in the IP header. The total is 5 plus the unsigned integers for options and padding. These are as per txipState and txipOpPdFormat used by IP segment that is transmitting. The bits, bit 11, bit 10, bit 9, bit 8 are reserved. */

/* Bits, bit 12 down to bit 0 are as fragment offset bits and as per already transmitted bytes from network driver task at node1. */

/* Code to find character IPHdrFlags from bit 15, bit 14, and bit 13. The bits are as per txipState.

.

.

20. /* Code to find 16-bit short integer IPHdrFrag from bit 13 down to bit 0. These bits are as per txipState specification, OutStreamSize and the bits already transmitted by the network driver. IPHdrFrag = *i* means OutStream byte numbering $(8)^i$ is being sent with this packet. */

.

.

}; /* End of the codes for returning an integer that specifies IP Header flags and fragment-offset bits. */

21. /* Declare other header field, 16-bit Checksum, 8-bit time to live and 8-bit, protocol filed, source node1 IP address and destination node 2 IP address. */

unsigned short **IPChecksum16*;

unsigned char **timeToLive*; /* *Note*: It decrements at each router on the way to node2. */

unsigned char **PrctlField*; / *Note*: *PrctlField* = 17 for UDP, = 6 for TCP, = 1 for ICMP.

unsigned short SourceIPAddr;

unsigned short DestnIPAddr;

unsigned char [ ] *extras*; unsigned char OptPdLen;

*IPextras* = OptionAndPds (*txipState*, *txipOpPdFormat*);

optPdLen = getLength8 (*IPextras*);

*IPHdrLen = (optPdLen + 20) /4;

unsigned char [ ] **IPOptionAndPds** (unsigned char [16] *txipState*, unsigned char [16] *txipOpPdFormat*) {

22. /* Codes for returning an array of bytes for the 32-bit integers for the options and padding. These are as per the State and thus IP header length parameters.

.

.

};

23. /* Codes for returning IP Header String. Remember that IP protocol transmits the integers with big endian. * /

/* Note: Instead of using many arguments, a typedef could have been used for the header data structure. However, this will consume more memory. */

```
unsigned char [ ] IPHeaderSelPkt (unsigned short SourceAddr, unsigned short DestnAddr, unsigned
short IPverHdrPrioSer, unsigned short *IPPktLen, char * IPVerHdrLen, unsigned char * IPHdrLen, unsigned
char *IPHdrFlags, unsigned short *IPHdrFrag, unsigned short *IPChecksum16, unsigned short UniqueID,
unsigned char *timeToLive, unsigned char *PrctlField, unsigned char [ ] Str, unsigned char OutStream,
char [optPdLen] IPextras, unsigned char [16] txipState, unsigned char [16] txipOpPdFormat) {
int i; unsigned char [ ] IPHStr; unsigned short lenflag; unsigned char optPdLen; unsigned short new;
24. /* Code for finding a stream of bytes actually been put with the packet. */
new = IPVerHdrPrioServBits (txipState, txipOpPdFormat, &IPPrioServ, &IPHdrLen);
IPHStr [0] = HByte (new);
IPHStr [1] = LByte (new);
25. /* Codes for estimating assigning UniqueID to the packet. * /
.
.
IPHStr [4] = HByte (UniqueID);
IPHStr [5] = LByte (UniqueID);
26. /* Codes for assigning Time to live and Protocol field from txipState. */
.
.
IPHStr [8] = *timeToLive; IPHStr [9] = *PrctlField;
IPHStr [12] = byte0 (SourceAddr); IPHStr [13] = byte1 (SourceAddr);
IPHStr [14] = byte2 (SourceAddr); IPHStr [15] = byte3 (SourceAddr);
IPHStr [16] = byte0 (DestnAddr); IPHStr [17] = = byte1 (DestnAddr);
IPHStr [18] = = byte0 (DestnAddr); IPHStr [19] = = byte0 (DestnAddr);
IPextras = IPOptionAndPds (txipState, txipOpPdFormat); /* Find Options and Padding. */
optPdLen = getLength8 (IPextras);
*IPHdrLen = (optPdLen + 20) /4;
while (optPdLen > = ++i) {IPHStr [i] = IPextras [i - 20];}; /* Fill the options and padding bytes.
*/
27. /* Codes for selecting the socket stream data to be sent and then estimating IPPktLen /
selectPktData (Str, OutStream, txipState, txipOpPdFormat, IPextras);
unsigned short *IPPktLen = getLength16 (Str) + (optPdLen + 20); /* Add byte of header
length also. */
IPHStr [2] = HByte (IPPktLen); IPHStr [3] = LByte (IPPktLen);
new = IPFlagFragBits (txipState, txipOpPdFormat, &IPHdrFlags, &IPHdrFrag, Str)
IPHStr [6] = HByte (new); IPHStr [7] = LByte (new);
unsigned short IPChecksum16 = checksum16 (IPHStr); /* Find checksum of IP header part only. */
IPHStr [10] = HByte (IPChecksum16); IPHStr [11] = LByte (IPChecksum16);
return (IPHStr);
};
/*---------------------------------------------------------------------------*/
/* Declarations and codes for the function, NetHdrFrTr for returning a Frame header string, frameHeader
and the fragment data in frame [ ] array from SocketStream. Str means the stream from internet layer at a
node, node1, end to be sent with header as per data link protocol and card protocol to other end, node2,
through the pipe having byte streams from the network driver. */
```

```
void NetHdrFrTr (unsigned char [ ] frame, unsigned char [ ] frameHeader, unsigned char [ ] trailBytes,
unsigned short fragOffset, unsigned char [ ] fragment, unsigned char [ ] trailBytes, unsigned short *FRlength,
unsigned short * HdrLen, unsigned short * endLen, unsigned short *FrameHdrFlags, unsigned short
*FrameHdrFragOffset unsigned short *FrameCRC32, unsigned short *PrctlField, unsigned char [ ] Str,
unsigned char [ ] SocketStream, unsigned char [ ] FRextras, unsigned char [16] txFRState, unsigned char
[16 ] txFROpPdFormat, unsigned char [12] netDrvType) {
unsigned char [ ] frame, frameHeader, trailBytes, fragment, trailBytes; unsigned short fragOffset;
unsigned short *FRlength, * HdrLen, * endLen, *FrameHdrFlags, *FrameHdrFragOffset
*FrameCRC32;
unsigned int CRC32 (unsigned int crc, unsigned char aByte);
/* Codes as per netDrvType, transmitting frame state txFRState, transmitting frame option
and padding format, txFROpPdFormat create an array of characters for the frameHeader,
for the fragment and for trailBytes. Each is of length, HdrLen, fraglen, endLen, FRlength.
Other fields calculated are short integer for fragOffset and unsigned integer for frame
CRC bits, FRCRC32, etc. */
.
.
.
}
/*****************************                :*******************************/
```

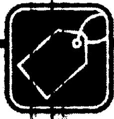
## Summary

The following is a summary of what we learnt in this chapter.

- Three case studies were explained. These are for automatic chocolate-vending machine, digital camera, and TCP/IP stack systems.
- Software engineering approach is first studying the requirements, then specifications, and finally the UML modeling.
- Drawing hardware architecture and software architecture simplifies design implementation.
- Use of MUCOS and VxWorks RTOSes and use of the IPCs for task synchronization and concurrent processing.
- ACVM system is modeled by three class diagrams of abstract classes ACVM_Devices, ACVM_Output_Ports and ACVM_Tasks.
- MUCOS RTOS functions have been used in an embedded system for an *automatic chocolate vending machine*. After initialization by the first task, seven tasks have been shown to control the various machine functions: (a) System user coins are of three denominations, Rs 1, 2 and 5 (b) Collection of coins at a chest (c) Refund in case of short change (d) Excess refund from a channel in case excess amount inserted into the machine (e) Delivering the chocolate through a port (f) Displaying messages as per the machine status (g) Displaying time and date. It shows how the semaphore as the MUCOS can be used as event flag. It also shows how to use the MUCOS mailboxes and the system RTC.
- Camera tasks are modeled by four class diagrams are divided Picture_FileCreation, Picture_FileDisplay, Picture_FileTransfer and Controller_tasks. We learnt that besides microcontroller and the several ASIPs are required for performance. A microcontroller executes the Controller_Tasks. The controller tasks are the following: (i) Task_LightLevel control (ii) Task_flash (ii) initialization of tasks, (iii) shooting task, (iv) initiates Picture_FileCreation tasks, which execute on a single purpose CCD processor (CCDP) for the dark current corrections, DCT compression, Huffman encoding, DCT decompression, Huffman decoding and file save,

(iv) initiates Picture_FileDisplay tasks, which execute on a single purpose display processor (DisplP) for decoded and compressed file image display after the required file bytestream processing for shift or rotate or stretching or zooming or contrast or color and resolution, (v) initiates memory stick save on notification from Picture_FileTransfer file system object using a single purpose transfer processor (MemP), (vi) initiates printing on a notification from Picture_FileTransfer using a single purpose print processor (PrintP), and (v) ) initiates USB port controller on notification from Picture_FileTransfer using a single purpose USB processor (USB_P).

- TCP stack generation is modeled by class Task_TCP, which is an abstract class from which the extended class (es) is derived to create TCP or UDP packet to a socket. The task objects are instances of the classes (i) Task_StrCheck, (ii)Task_OutStream, (iii) Task_TCPSegment or Task_UDPDatagram, (iv) Task_IPPktStream (v) Task_NetworkDrv.
- VxWorks RTOS has been used for embedded system codes for driving a network card after generating the TCP/IP stack. The exemplary codes show a method of code designing for sending the bytestreams on a *network*. A bytestream first passes through the multiple layers with TCP/IP protocols. A network driver finally sends the stream on a TCP/IP network. How VxWorks schedules the six tasks is shown. (a) A task is for checking the insertion of application strings by the application. (b) A task for creating the application stream for the transmission control layer. (c) Two task codes are given for inserting the header fields using either of two protocols, TCP or UDP, at the transport layer. (d) A task creates IP packets for the network by fragmenting the TCP segment stream. (e) A task drives the network driver and places the bytestream into a pipe (a virtual device). How the tasks handle the multiple data types at the header fields was also shown. This case study gives the reader a thorough understanding of the application and the various functions of the system RTC in VxWorks RTOS. The reader must be able to develop solutions for an embedded networking system using the VxWorks.



## Keywords and their Definitions

***Application Layer*** : A layer consisting of fields attached before placing the message on the network so that the application at the other end understands the request and service.

***Checksum*** : A sum representing the number of carries generated by adding the 8-bit numbers or 16-bit numbers or 32-bit numbers.

***CCD*** : Charge couple device consisting of a large number of horizontal and vertical cells. Red, green and blue (RGB) cells generate three outputs for a pixel.

***CODEC*** : A processing unit for encoding and decoding the bytes or bytestreams, for example, JPEG CODEC for encoding and decoding the bytes or bytestreams from a CCD preprocessor.

***Compression*** : To convert the bytes in a file by suitable encoding into a reduced number so that they can be retrieved back by a reverse process of suitable encoding for decompression.

***CRC (Cyclic Redundancy Check)*** : A 32-bit or 16-bit integer calculated so that if there is an error in the transmission, then it can be detected by comparing the CRC of the received message. It takes a longer time to calculate but it is better than the checksum.

***Connection-Oriented protocol*** : A protocol in which inter-network communication first takes place for connection establishment, then the message flows under a flow control mechanism and then the connection termination occurs after suitable inter-network communication. Transmission Control Protocol, TCP, is an example.

***Connection-less protocol*** : A protocol in which inter-network communication takes place without first connection establishment and without a flow control mechanism and without the connection termination by inter-network communication. Usually, in broadcast mode, this protocol is observed. User Datagram Protocol, UDP, is its example.

***Cryptographic Software*** : Software for encrypting and deciphering a message or a set of bytestreams. It uses an algorithm for encrypting and another algorithm for decrypting.

***Datagram*** : A stream of bytes that is independent of the previous stream. The UDP datagram has a maximum size of $2^{16}$ bytes.

***DES*** : Data Encryption Standard.

***DES3*** : A version of DES.

***Electromechanical Device*** : A device to operate a mechanical system using electronic signals.

***Fabrication Key*** : A key embeded in ROM at the time of card fabrication so that the card gets a unique identity.

***internet Layer*** : A layer for inter-networking by TCP/IP protocol. The layer consists of fields attached before placing the message on the network through routers so that the routers send the message as per the source and destination IP address fields in the layer. The fields at this layer are for network-flow controlling mechanism. For example, IP header fields in TCP/IP.

***IP header*** : Bytes as per IP protocol placed over the bytes of a packet of TCP segment stream.

***IP packet*** : An IP protocol based packet of size upto $2^{16}$ bytes that transmits after packetization of TCP segment stream-data and after placing IP header bytes.

***LCD dot Matrix*** : A set of Liquid Crystal Display consisting of rows and columns, like a matrix. A suitable controller shows the characters or pictures at the matrix.

***Network Driver*** : A card that sends and receives the messages physically from the network and facilitates the physical connection for the stream of bytes between the different layers. A driver may use SLIP or PPP protocol for driving on the net after placing the header and trailing bytes as per Ethernet of another protocol. Each driver has a unique?

***Packet*** : A set of bytes which includes application, transport and internet layer headers that routes through a set of routers along a path presently available towards a destination.

***Pixel*** : Smallest unit of an image, the area of which depends inversely on image resolution. An image consists of horizontal and vertical pixels. At each pixel, in coloured image there are three cells (RGB).

***RGB*** : red, green and blue colors. Any colour in the visible range is a combination of these three colors. There can be 256 or $2^{16}$ colours. It depends on the colour resolution.

***SLIP*** : A Serial Line Interface Protocol.

***TCP Header*** : A header placed at the transmission control layer as per TCP protocol.

***TCP/IP stack*** : A stack of the bytes that transmit from network drive after placing appropriate TCP/TP suite protocol headers and option bytes over that.

***TCP Segment*** : A segment of the bytes along with the headers of TCP and application layer protocols that transmits in a single instance through lower layers.

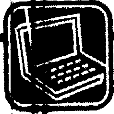
***Transmission Control Layer*** : A layer for placing TCP header fields or UDP header fields on a TCP/IP network.

***UDP*** : A protocol at the transmission control layer for sending a TCP/IP network datagram.

***UDP Header*** : A header consisting of four fields, source and destination port addresses and for length and checksum.

## Review Questions

1. Why must a designer first understand the system requirements before designing the codes? Why are the list of tasks and synchronization models required before using RTOS functions.
2. The while loop of the first task contains only task suspend functions. Explain why it should be so?
3. Explain how the task for reading ports synchronizes with the port device driver.
4. How do you use MboxAmount in Example 11.1.
5. What is the role of Task_Display? How do you code for the multiple line messages at an LCD matrix? (MUCOS)
6. Explain use of semFlags in the ACVM tasks in Example 11.1. (MUCOS)
7. We can use the number of mailbox IPC messages from a task. Explain how this has been effectively used. Why is a message queue not used in place of multiple mailboxes? (MUCOS)
8. Explain the role of each semaphore used. How will the semaphore be used and consequently the codes in Example 11.1 be modified in the following modification to the system? ACVM using a random number generator C function delivers one chocolate free out of an average of eight coin insertions and refunds all the coins to the lucky ones. (MUCOS)
9. Why do we need multiple single purpose processors along with a microcontroller in a digital camera?
10. Why are the DCTs and inverse DCTs done in a digital camera? Why is fixed point arithmetic used instead of floating point arithmetic in digital camera?
11. Describe classes, the objects of which are used in writing codes for saving an image file in a digital camera.
12. How does the range of priority assignments differ from MUCOS assignments in VxWorks?
13. Explain how the semaphore is used in Example 11.2 to direct the use of UDP in place of TCP at the transport layer. (VxWorks)
14. How are size streams bigger than permitted in a packet, and permitted by MTUs (Maximum Transferable Units) at a time handled?



## Practice Exercises

15. Explain how the task for reading ports synchronizes with the port device driver.
16. Draw the class diagram of Controller_Tasks for a digital camera.
17. Explain the use of the statement: 'fd = open (*netDrvConfig.txt*, O_RDWR, 0);'. (VxWorks)
18. Draw the state diagram of ACVM functions.
19. Draw the state diagram of digital camera functions.
20. Draw the state diagram of TCP stack generation.

# Design Examples and Case Studies of Program Modeling and Programming with RTOS-2

12

Recall

In the previous chapter, ACVM was an interesting example designed to teach embedded system design concepts and RTOS applications. We have seen three case studies in that chapter. These were for ACVM, digital camera and TCP stack systems. We learnt that first we study the requirements, get clarity of the required purpose, inputs, signals, events, notifications, outputs, functions of the system, design metrics, and test and validation conditions, specifications, in terms of users, objects, sequences, activity, state and class diagrams for abstracting the component, behaviour and events and create a description in terms of signals, states and state machine transitions for each task that help us greatly in defining the system architecture for hardware and software. The coding for implementation is done as per the architecture. The system is then finally tested.

LEARNING OBJECTIVES

*Like in the previous chapter, we will first start with an interesting example of robot orchestra. We will learn aspects of embedded system design from the four new case studies described in this chapter.*

1. *Inter-robot communication in a robot* ***orchestra****.*
2. *Embedded systems required in an automobile. Understand design of control system and code implementation using the example of an* ***automotive cruise control*** *(ACC) system in a car. Also learn how the code design can be done using VxWorks after emulating OSEK (Section 10.2) features.*
3. *Learn card-host communication tasks and* ***smart card*** *OS features. Also learn system design and code implementation for communication of smart card with the host machine (for example, bank ATM machine or payment machine when using a credit card at a shopping mall).*
4. *Learn interrupts, tasks, and state machine concepts by example of SMS create application in a* ***mobile phone****.*

## 12.1 CASE STUDY OF COMMUNICATION BETWEEN ORCHESTRA ROBOTS

Qrio was an invention of Sony. *Times* magazine described it in 2003. Qrio means **Q**uest for cu**RIO**sity. Qrio was a smoother and faster humanoid robot than ever designed before. Qrio weighed 7.3 kg and was 58 cm, and had a one-hour battery. Four Qrios performed a complicated dance routine in 2003.

Qrio was a bipedal robot which could wave hello and recognize voice, its two CCD cameras in the eyes recognized up to 10 different faces and objects, and determined the location of objects in view. Three CCD cameras were used in all, one in each eye and one at the centre. Qrio could converse, sing, walk uphill, dance, kick and play using its fingers. Qrio had seven microphones. Qrios could sing in unison and interact with humans with movements and speech with more than 1000 words. It could learn new words also. Emotions were shown by flashing lights.

In 2006, ten Qrios of the fourth generation performed a dance number. They also conducted an entire orchestra. They gave a unique rendition of Beethoven's 5$^{th}$ symphony (1808), one of the most well known and popular compositions of western classical music, which is often played in the orchestra using several musical instruments and conducted by a conductor. [http://en.wikipedia.org/wiki/Symphony_No._5_(Beethoven)]. The Qrio robot orchestra had novel instruments played by robotic actuators.

Each Qrio had 38 fluid motion flexible joints controlled by separate motors for each with an ASIP in each. Three central system microcontrollers controlled motion, recognized speech and visual images, respectively. Memory was 64 MB with each microcontroller.

Qrio embedded complex algorithms for complex mechanical manoeuvres for balancing during dancing or in the orchestra. Qrios embedded complex orchestral and/or choreographic programs.

Figure 12.1 shows a robot orchestra.

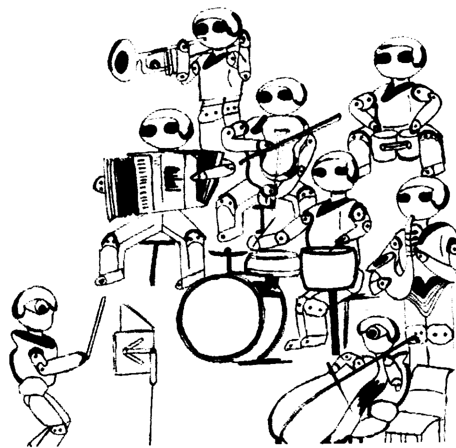


**Fig. 12.1** Robot-orchestra

Commands and messages communicated in each Qrio between the microcontrollers, sensors and actuators. A robot was programmed using an *Orchestrator.*

Assume that there are $k$ sensor inputs to the module and $q$ outputs generate to actuators and $p$ outputs to *message boxes* (also called mailboxes in certain OSes or notifications in certain OSes) in a sequence. *Orchestrator* is software which sequences, synchronizes the inputs from the $1^{st}$ to the $k^{th}$ sensor and generates messages and outputs for the actuators, display and message boxes at the specified instances and time intervals. Message boxes store the notifications, which initiate the tasks as per the notifications.

Figure 12.2(a) shows embedded software module Orchestrator-1 at a microcontroller 1. Figure 12.2(b) shows commands and messages communication between Orchestrator-$x$, Orchestrator-$y$ and Orchestrator-$z$ software modules at same or different microcontrollers.

A musical device communicates data to another using a protocol called MIDI (Musical Instrument Digital Interface). Reader is advised to study a tutorial on MIDI. Popular websites are http://www.borg.com/~jglatt/tutr/miditutr.htm and http://www.xtec.es/rtee/eng/tutorial/midi.htm for this purpose.

Most musical instruments are MIDI compatible and have MIDI IN and MIDI OUT connections, which are optically isolated with the musical-instrument hardware. Three MIDI specifications define (i) what a physical connector is, (ii) what message format is used by connecting devices, controlling them in *real time* and standard for MIDI (musical instruments digital interface) files. Each message consists of a command and corresponding data for that command. Data are sent in byte formats and are always between 0 and 127 and corresponding command bytes in a channel message are from 128 to 255.

A channel (command) message 128–255 (between 0x80 to 0xEF) in MIDI file has musical *note*, *pitch-bend*, *control change*, *program change* and *after-touch* (poly-pressure) messages. There are a maximum of 16 channels in a system. MIDI specifies system messages, manufacturer's system exclusive messages and real time system exclusive messages (between 0xF0 to 0xFF). A real time system message example is as follows: A MIDI Start from conductor is 0xFA command and always begins playback at the very beginning of the song (called MIDI Beat 0)]. So when a slave player receives a MIDI Start (0xFA), it automatically resets its song position = 0.

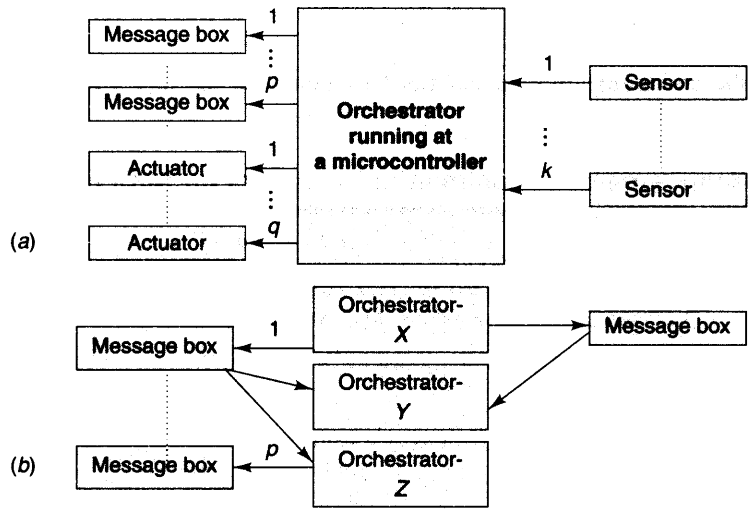



Fig. 12.2 (*a*) Embedded software module of Orchestrator-1 at a microcontroller 1 with *k* sensor inputs, *q* outputs to actuators and *p* outputs to message boxes (*b*) Commands and messages communication between Orchestrator-*x*, Orchestrator-*y* and Orchestrator-*z* software modules at same or different microcontrollers

MIDI messages define an event as what musical note is pressed and with what speed it was pressed. This event is input to another MIDI receiver. MIDI receiver plays that note back with the specified speed. Therefore, an entire ensemble of a robot-orchestra can be controlled using MIDI protocol. Also the actuators are synchronized as per the notes and speeds.

Transport of a MIDI file can be over Bluetooth, high Speed Serial, USB or FireWire. For the robot orchestra, communication protocol for MIDI files can be Bluetooth or WLAN 802.11.

Figure 12.3 shows communication between the robot conductor *C* and four slave-robots (players *P1* to *P4*) in the robot-orchestra. Each slave robot plays distinct musical instruments and thus plays the notes from distinct track (channel) of MIDI file from the master (conductor).

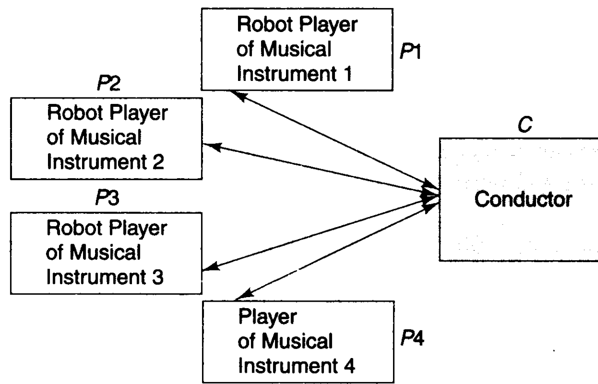



Fig. 12.3 Communication model of a robot- orchestra

### 12.1.1 Requirements

Requirements of the communicating robots can be understood through a requirement table given in Table 12.1.

**Table 12.1** Requirements of Communication system for MIDI files in Robot orchestra

| *Requirement* | *Description* |
|---|---|
| Purpose | To communicate selected MIDI file's data over Bluetooth personal-area-network from Robot Conductor communication task to music playing robots |
| Inputs | 1. Orchestra_Choice for selected MIDI file<br>2. MIDI File |
| Signals, Events and Notifications | 1. Commands (channel-messages) in the file's data |
| Outputs | 1 MIDI File over the communication network<br>2. Messages to actuators for movements as per specific track messages and data |
| Functions of the system | A user signals an Orchestra_Choice, say Beethoven's 5th symphony to start. The conductor C (Figure 12.3) task_MIDI selects the chosen MIDI file and posts the bytes from the files in message queue for task_Piconet_Master. Piconet is a network of Bluetooth devices. Bluetooth devices can form a network known as *piconet* with the devices within a distance of about 10 m. Bluetooth piconet consists of network of *C*, *P*1, *P*2, *P*3 and *P*4. task_PICONET_SlaveP1, task_PICONET_SlaveP2, task_PICONET_SlaveP3 and task_PICONET_SlaveP4 accept the messages from task_Piconet_Master. A task_Piconet_Slave posts the MIDI messages to a task_ MIDI_Slave. task_ MIDI_Slave posts the messages to a task_Orchestrator, which controls the motor movements of the slave robot motor actuators and musical-note actuators. |
| Design metrics | 1. *Power Dissipation*: As required by mechanical units, music units and actuators<br>2. *Performance*: Near human equivalent of Orchestra play<br>3. *Engineering Cost*: US$ 150000 (assumed) including mechanical actuators<br>4. *Manufacturing Cost*: US$ 50000 (assumed) |
| Test and validation conditions | 1. All MIDI commands must enable all orchestral functions correctly |

### 12.1.2 Class Diagram and Classes

A class diagram [Table 6.3] can be modeled by a class diagram of Task_Conductor. Figure 12.4 shows class diagrams of Task_Conductor and other classes, which are used for posting a MIDI file messages over the piconet to the slaves.

1. Task_Conductor interfaces ISR_ Orchestra_Choice and extends to Task_MIDI. Task_ Conductor extends to Task_Piconet_Master.
2. Task_SlaveP1, Task_SlaveP2, Task_SlaveP3, and Task_SlaveP4 are abstract classes and extend to classes Task_Piconet_SlaveP1, Task_ Piconet _SlaveP2, Task_ Piconet _SlaveP3 and Task_ Piconet _SlaveP4, respectively.
3. Task_MIDI_SlaveP1, Task_MIDI_SlaveP2, Task_MIDI_SlaveP3 and Task_MIDI_SlaveP4 are classes in slaves P1, P2, P3 and P4, respectively.

4. Task_Orchestrator_SlaveP1, Task_ Orchestrator _SlaveP2, Task_ Orchestrator_SlaveP3 and Task_ Orchestrator _SlaveP4 are classes in slaves *P*1, *P*2, *P*3 and *P*4, respectively.

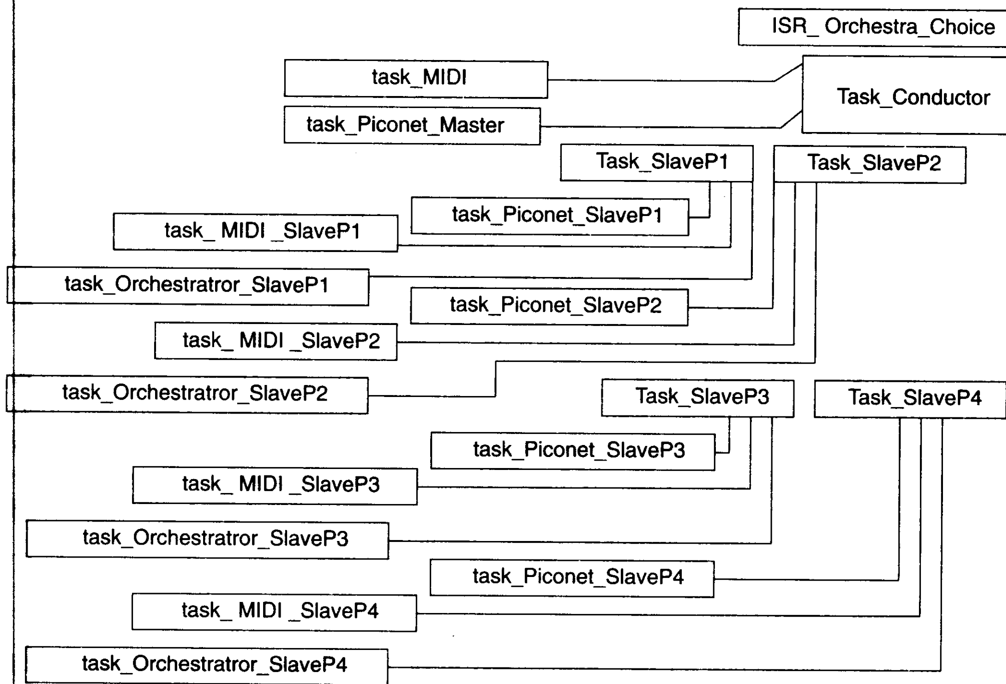



**Fig. 12.4** Task_Conductor class diagram of for posting a MIDI file data over the Bluetooth piconet to the slaves

***Class*** A class is a logical group and has fields (attributes) and methods (operations). Figure 12.5 shows an example of fields in class Task_MIDI. The class consists of *fileType* field. It specifies the type of file. It defines Type 0 file if the file contains only one track, which has all of the MIDI messages, which means the entire orchestra performance for that one track. A track can have many musical parts in different MIDI channels, a channel for each slave robot. Type 1 file specifies each musical part in a separate track for each slave robot Both Type 0 and 1 store one orchestral performance, for example, Beethoven's 5th symphony. The class other fields are MsgMusicalNote, MsgPitchBend, MsgControlChange, MsgProgramChange, MasgAfterTouch, MsgSystem, MsgManufExclMsg and MsgRealTime. These are string objects for musical note, pitch-bend, control change, program change, after-touch, system messages,

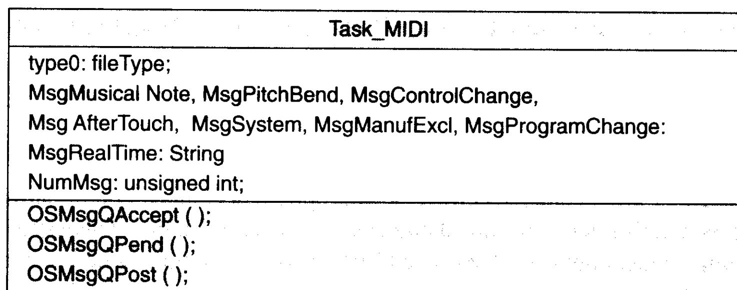



**Fig. 12.5** Class MIDI

manufacturer's system exclusive messages and real-time system exclusive messages. NumMsg is the number of messages which transfer in one cycle from masters to slave. OSMsgQAccept ( ), OSMsgQPend ( ) and OSMsgQPost ( ) are the OS methods to accept, wait and post the messages of orchestra choice and MIDI messages.

***Objects*** An object is an instance of a class. Assume a notation representation in which we represent an object ID for process or task by characters starting with lower case and for the corresponding class by characters starting with first character in capital case. For example, the task_MIDI, task_Piconet_Master, task_Piconet_SlaveP1, task_ Piconet _SlaveP2, task_ Piconet _SlaveP3 , task_Piconet_SlaveP4, task_MIDI_SlaveP1, task_ MIDI_SlaveP2, task_MIDI_SlaveP3 and task_MIDI_SlaveP4, task_Orchestrator_SlaveP1, task_Orchestrator_SlaveP2, task_ Orchestrator_SlaveP3 and task_ Orchestrator _SlaveP4 are the objects (processes) of the classes Task_MIDI, Task_Piconet_Master, Task_Piconet_SlaveP1, Task_ Piconet _SlaveP2, Task_ Piconet _SlaveP3 , Task_ Piconet _SlaveP4, Task_MIDI_SlaveP1, Task_MIDI_SlaveP2, Task_MIDI_SlaveP3 and Task_MIDI_SlaveP4, Task_Orchestrator_SlaveP1, Task_Orchestrator_SlaveP2, Task_Orchestrator_SlaveP3 and Task_ Orchestrator_SlaveP4.

Message queue objects MsgQMidi, MsgQBluetooth, MsgQMidiP1, MsgQBluetoothP2, MsgQMidiP2, MsgQBluetoothP3, MsgQMidiP3, MsgQBluetoothP4, MsgQMidiP4, SigPort1, SigPort2, SigPort3 and SigPort4 are used for posting and accepting messages. MsgQMidi posts NumMsg messages from MIDI file in one cycle to the object task_Piconet_Master. MsgQBluetooth posts Bluetooth stack data in one cycle to Port_Bluetooth. Signal objects SigPorts to ports initiates transfer of Bluetooth stack. Signal object SigPortP1, SigPortP2, SigPortP3 and SigPortP4 initiates reception of Bluetooth stack.

### 12.1.3 State Diagram

Figure 12.6 shows a state diagram for objects from classes of Task_Conductor. A state diagram shows a model of a structure for its start, end, in-between associations through the transitions and shows events-labels (or condition) with associated transitions. A dark rectangular mark within a circle shows the end. [Table 6.3] The state transitions takes place between the tasks, task_MIDI, task_Piconet_Master, task_Piconet_SlaveP1, task_ Piconet _SlaveP2, task_ Piconet _SlaveP3 , task_ Piconet _SlaveP4, task_MIDI_SlaveP1, task_ MIDI_SlaveP2, task_ MIDI_SlaveP3 and task_ MIDI_SlaveP4, task_Orchestrator_SlaveP1, task_Orchestrator_SlaveP2, task_ Orchestrator_SlaveP3 and task_ Orchestrator _SlaveP4.

### 12.1.4 Robot Orchestra MIDI Communication Hardware and Software Architecture

Hardware architecture specifies the appropriate decomposition of hardware into processors, ASIPs, memory, ports, devices and mechanical and electromechanical units. It also specifies interfacing and mapping of these components. Figure 12.7 shows a block diagram of communication hardware architecture. Following are the specifications:

1. A microcontroller at master and each slave to control Orchestrator for movements. An ASIP for each motor movement.
2. An ASIP at master and each slave for Bluetooth piconet communication between master and slaves.

***Software Architecture*** Software architecture specifies the appropriate decomposition of software into modules, components, appropriate protection strategies, and mapping of software. Software consists of the following at master and each slave.

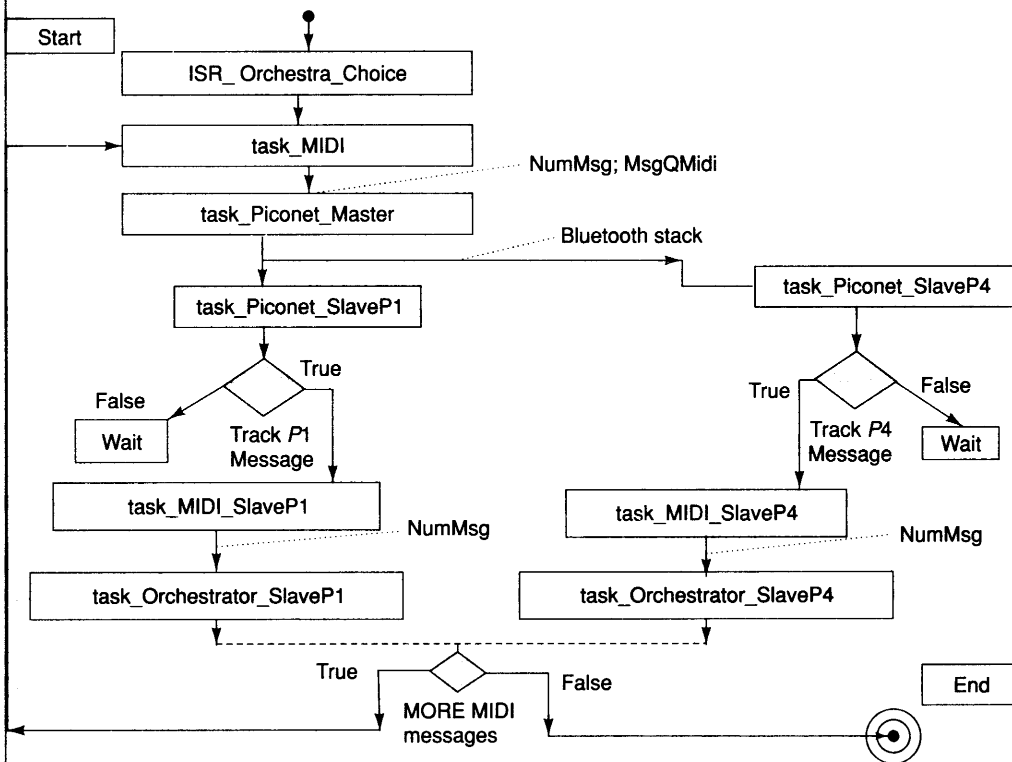



**Fig. 12.6** State diagram for task_MIDI

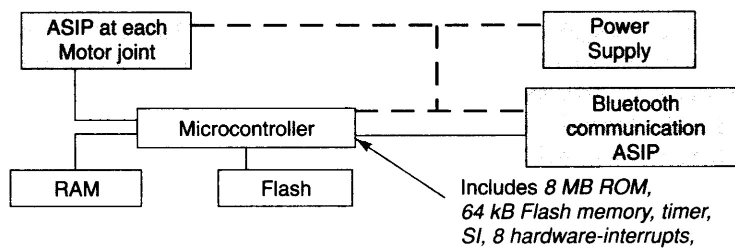



**Fig. 12.7** Block diagram of communication-system Microcontroller and ASIPs at master and each slave

1. OS
2. ISRs for initiating action on user inputs and GUI notifications, for example for Orchestra_Choice.
3. The tasks are the objects of the classes shown in Figure 12.4.
4. Orchestrator tasks for master and slaves.
5. OSMsgQAccept, OSMsgQPost and OSMsgPend are the message queue methods (functions) for system call to the OS. Three methods synchronize the tasks and their concurrent processing such that first

task_MIDI waits for message for Orchestra_Choice from the ISR_Orchestra_Choice, then the task_MIDI posts the bytes in message queue for task_Piconet_Master.

6. task_MIDI waits for NumMsg MIDI messages from MIDI file for chosen orchestra and posts the NumMsg MsgQMidi messages to task_Piconet_Master.
7. task_Piconet_Master discovers the slaves and sets up piconet Bluetooth device network on first initiation. On accepting MsgQMidi messages, it sends protocol stack outputs through the communication port.
8. A task_Piconet_Slave receives Bluetooth stack for the NumMsg messages of MIDI file and sets up network with master as server. The MIDI messages are sorted for the messages to be directed to track *P*1 or *P*2 or *P*3 or *P*4. For example, track*P*1 messages post to task_MIDI_SlaveP1 from task_Piconet_SlaveP1. Then task_MIDI_SlaveP1 posts the messages to actuators and Orchestrator task_OrchetratorP1.
9. Steps similar to Step 8 above repeat for all slaves tracks in robot-orchestra.

### 12.1.5 Communication Tasks Synchronization Model

Figure 12.6 showed state diagram of synchronizing cycle of different tasks. The communication system has a cycle of actions in a task synchronizing model.

1. A cycle starts from task_MIDI, which receives the *events* (Orchestra_Choice). It posts MIDI messages into message queue MsgQMidi. The MIDI messages are in the MIDI file of Orchestra_Choice stored at master.
2. A task task_Piconet_Master is for discovering the piconet slaves, establishing Bluetooth communication with the slaves *P*1, *P*2, *P*3 and *P*4 and then accepting MIDI messages from the MsgQMidi. task_Piconet_Master posts Bluetooth stack into message queue MsgQBluetooth and then signals sigPort to Port_Bluetooth.
3. task_Piconet_SlaveP1 executes on a signal SigPortP1 from Bluetooth port Port_BluetoothP1 at the slave. Similarly, actions are at Port_BlueoothP2, Port_BluetoothP3 and PortBluetoothP4, which signal three signals SigPortP2, SIgPortP3 and SigPortP4 to the task_Piconet_SlaveP2, task_Piconet_SlaveP3 and task_Piconet_SlaveP4 respectively.
4. task_Piconet_SlaveP1, task_Piconet_SlaveP2, task_Piconet_SlaveP3 and task_Piconet_SlaveP4 (i) accept Bluetooth stack of the master, (ii) select the track messages for the four tracks of four slaves, and post into four message queues the four messages in MIDI format task_MIDI _SlaveP1, task_ MIDI _SlaveP2, task_ MIDI _SlaveP3 and task_ MIDI _SlaveP4 for track *P*1, *P*2, *P*3 and *P*4, respectively.
5. task_MIDI _SlaveP1, task_ MIDI _SlaveP2, task_ MIDI _SlaveP3 and task_ MIDI _SlaveP4 post to track *P*1, *P*2, *P*3 and *P*4 Orchestrators the MIDI messages. Orchestrators direct the messages to the actuators of the music track and robot movements.

Figure 12.8 shows a synchronization model for master–slave robots communication tasks using message queues and signals. (Tasks waiting for IPCs 6b, 6c, 6d, 5b, 5c and 5d are not shown in the figure.)

## 12.2 EMBEDDED SYSTEMS IN AUTOMOBILE

Present-day automobiles have many embedded systems. Figure 12.9 shows the type of systems in a car. Each system has at least one microprocessor or microcontroller and software. A car may contain the following nine types of systems embedded in it.

1. **Engine control**: Engine control system is by automatic control of fuel injection.
2. **Speed control and brake**: Speed Control and brake systems are automatic cruise control (*ACC*), antilock braking, automatic braking and regenerative braking.

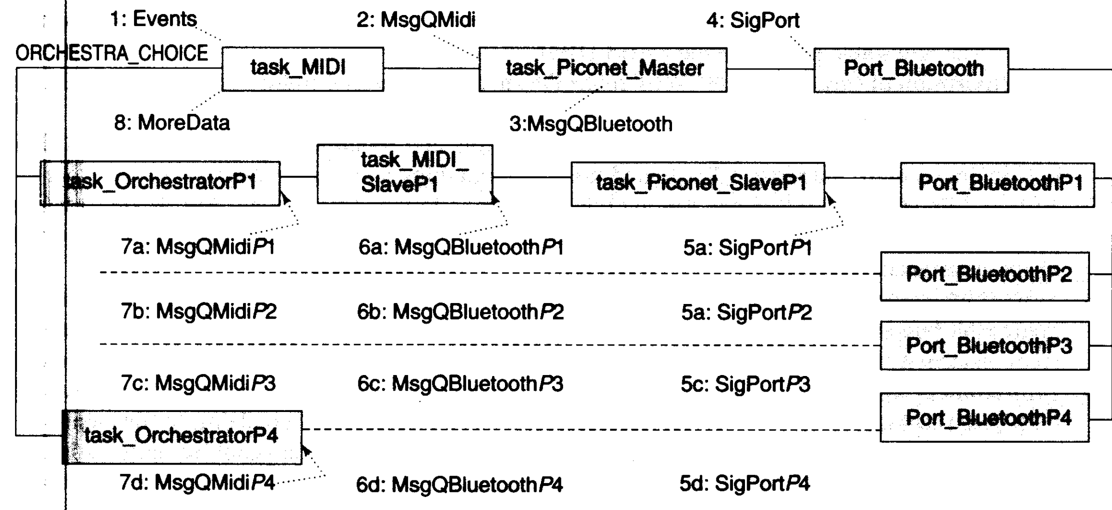



**Fig. 12.8** Synchronization model for master-slave robots communication tasks

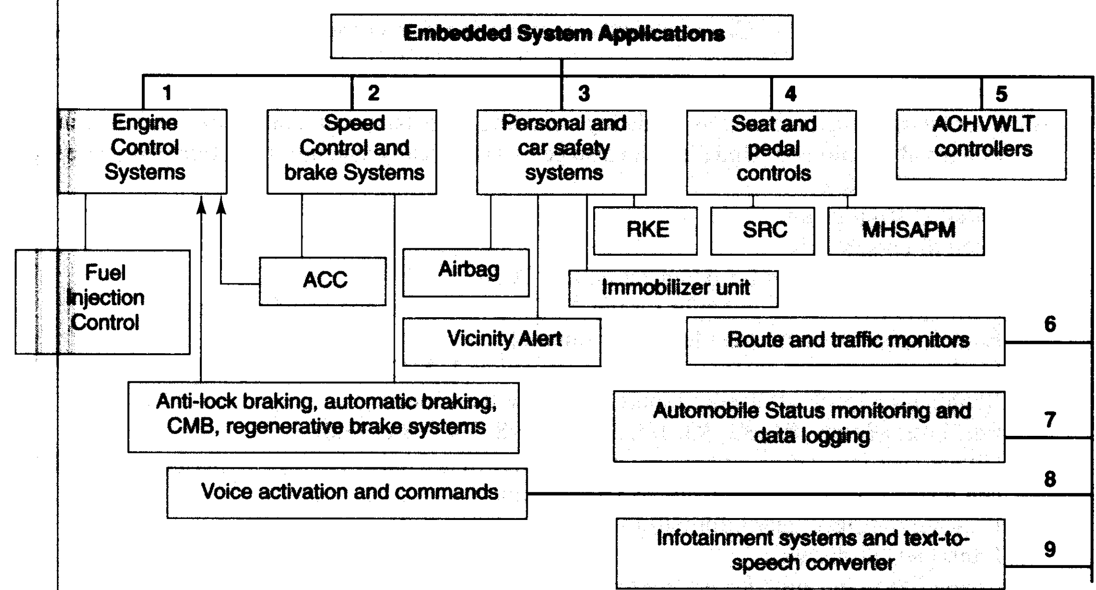



**Fig. 12.9** Type of applications of the embedded systems in the car

3. **Safety systems**: Safety systems are for the personal safety and car safety. Personal safety is provided by multiple and variable speed airbags. Airbags are used for cushioning, in particular for automatic rapid inflation of the bag in the case of a collision followed by deflation. Car safety is provided by RKE (remote key entry) and *immobilizer unit*. RKE system has embedded CPU and software to control door locks. An immobilizer unit immobilizes the car if an unauthorized user attempts to drive the car away. Security systems include alarms, vicinity alert and lane departure alert. A CMB (collission mitigation brake) system is also deployed for safety.

4. **Seat and pedal controls**: Seat and pedal controllers are the seat restraint controller (SRC) and memory heated seat adjustable pedal module (MHSAPM).
5. **Car environment controls**: ACHVWLT controllers to control the air-conditioning, heater, ventilation, windows, light and temperature.
6. **Route and traffic monitors**: Route and traffic controls and monitoring are done by GPS (global positioning system) mapping, GPS guided route programming and route planners.
7. **Automobile status monitoring**: Automobile status monitoring, data loggers and driving assisting device management, steerable head lights, windshield IR camera, windscreen heads up display, night vision assistance.
8. **System interfaces for commands, voice activation, and interfacing**: System interfaces are soft programmable buttons, voice commands, Bluetooth interfacing for handsfree telephony and wireless personal area connectivity in the automobile, WiFi Internet connectivity, and GSM/CDMA for mobile connectivity.
9. **Infotainment systems**: Infotainment systems are as following: displayed text-to-speech converters, GPS based car location and surrounding area maps, cached traffic reports for real time traffic monitoring, cassette player, car phones, audio CD player, LCD screen, touch screen panel, satellite or Internet based Radio, VCD/DVD players and hands free telephony using Bluetooth and GSM or CDMA.

The following functions are the discrete components which are later integrated to a central serving system.

1. ***RTC*** and ***watchdog timers*** for the tasks. [Refer to Sections 1.3.4 and Section 3.7]
2. ***Real-time control*** for all electronic, electromechanical and mechanical systems.
3. ***Adaptive Cruise Control*** (ACC) to maintain constant speed in cruise mode and to decelerate when a vehicle comes in front at a distance less than safe and to accelerate again to cruise mode in adaptive mode.
4. ***Data Acquisition System*** (DAS) for the following:
   (a) Current time and date display, updating, recoding of instances of malfunction and time intervals of normal functioning. Updating can also be by periodically synchronizing time with time signals from GPS system
   (b) External temperature
   (c) Internal temperature
   (d) Total distance covered for odometer and for ACC
   (e) Road speed of vehicle in km/hr for the ACC, speedometer and speed warning
   (f) Engine speed in r.p.m. [revolution per minute]
   (g) Coolant temperature, instances of its excess above 115° C and constancy within 80 to 90° C
   (h) Illumination levels at display panels and inside the vehicle
   (i) Fuel level (Empty, *R*1, *R*2, *R*3, 1/4, 3/8, 1/2, 5/8, 3/4, 7/8 or Full)
   (j) Oil pressure for starting oil pressure warning system activation for engine speed above 5000 r.p.m. and alarm system activation above 15000 r.p.m.
   (k) Presently engaged gear information
   (l) Front-end car distance
5. ***Front panel Switches and display controls***
6. Port for ***alarm*** signals. These are sent to display panel, beep and buzzer-sound systems. A warning is issued by displaying pictograms and raising sound-alarms and their appropriate records is *saved* for diagnostic analysis. [A pictogram is a pre-recorded picture in an image data file. It displays on an LCD matrix. Pictogram displayed is as per the required message]
7. ***Diagnostics computations*** for driving time malfunctions statistics and analysis results for fast fault diagnosis by mechanic
8. ***Multi Media Interfaces***
9. ***Control Area Network*** Interfaces
10. ***Serial Communication Interface*** (SCI), transmitter and receiver.

Hardware of embedded systems of a car can be designed using ASICs and microcontrollers and DSPs, for example, 80x51, 68HC11/12, PIC, C167, ADSP2106x, 68HC0x, MCORE, Star12, TMS470, Hitachi H8S2xxx series, and ARM 9 based ST9 series. Section 12.3 describes a case study of software implementation aspects of an ACC system in a car.

## 12.3 CASE STUDY OF AN EMBEDDED SYSTEM FOR AN ADAPTIVE CRUISE CONTROL (ACC) SYSTEM IN A CAR

The choice of case study of an ACC is taken up to understand a control system design and also to understand use of RTOS for code implementation. The system has a number of ports for data input and output. The system uses a control algorithm. Sections 12.3.1 and 12.3.2 give the design steps, for requirements and class diagram of ACC system tasks. Sections 12.3.3 and 12.3.4 describe hardware and software architecture. Sections 12.3.5 and 12.3.6 give the ACC tasks synchronization model and software implementation, respectively.

### 12.3.1 Requirements

Requirements of the ACC system can be understood through a requirement table given in Table 12.2.

**Table 12.2** Requirements of an Adaptive Cruise Control

| *Requirement* | *Description* |
|---|---|
| Purpose | 1. Controlled cruising of car using adaptive control for the car speed and inter-car distance. |
| Inputs | 1. Present alignment of radar (or laser) beam emitter.<br>2. Delay interval in reflected pulse with respect to transmitted pulse from emitter.<br>3. Throttle position from a stepper motor position sensor.<br>4. Speed from a speedometer.<br>5. Brake status for brake activities from brake switch and pedal. |
| Signals, Events and Notifications | 1. User commands given as signals from switches/buttons. User control inputs for ACC ON, OFF, Coast, resume, set/accelerate buttons.<br>2. Brake event. (Brake taping to disable the ACC system, as alternative to "cancel" button at front panel).<br>3. Safe/Unsafe distance notification. |
| Outputs | 1. Transmitted pulses at regular intervals.<br>2. Alarms<br>3. Flashed Messages<br>4. Range and speed messages for other cars (in case of string stability mode).<br>5. Brake control<br>6. Output to pedal system for applying emergency brakes and driver non-intervention for taking charge of cruising from the ACC system. |
| Control Panel | Control front-end panel has the following: (a) Switch cum display for 'ON', for 'OFF', 'COAST', 'RESUME', and SET/ACCELERATE. The driver activates or deactivates, the ACC system by pressing ON or OFF, respectively. S/he hands over or resumes the ACC system |

(*Contd*)

| Requirement | Description |
|---|---|
| | charge by pressing COAST or RESUME, respectively. S/he sets the cruise speed by SET/ ACCELERATE switch. A switch glows either green or red as per the status when the ACC activates. (b) Alarms and message flashing unit issues appropriate alarms and message flashing pictograms. |
| Functions of the system | 1. Cruise control system takes *charge* of controlling the throttle position from the driver and enables the cruising of the vehicle at the preset constant speed. A radar system maintains inter-car distance and warns of emergency situations.<br>2. An alignment circuit aligns the radar emitter. When driving in a hilly area, the emitter alignment is a must. A stepper motor aligns the attachment so that transmitter beam of radar emits with the required beam alignment for the given driving lane and divergence to maintain the in-lane line of sight of the front-end car. task_Align does this function.<br>3. Transmit pulses emit at periodic intervals and the delay period in receiving its reflection from front-end vehicle is computed. Each pulse can be suitably modulated so that there is noise immunity in the system and beams of multiple sources don't interfere. task_Signal does this function.<br>4. The measured delay at periodic intervals is multiplied by $1.5 \times 10^8$ m/s (half of light velocity) gives the computed distance d (= RangeNow) of front end car at that instance. task_ReadRange does this function.<br>5. The differences of d with respect to safe $d_{safe}$ and preset distances $d_{set}$ (in case of maintaining string stability) are cyclically estimated. task_RangeRate does this function.<br>6. The speedometer measures the speed and error in preset speed and measured speed is also periodically estimated. task_Speed does this function.<br>7. All estimated differences are cyclically sent as input to an adaptive algorithm, which adapts the control parameters and sends the computed output to vacuum actuator of throttle valve. task_Algorithm does computations and task_Throttle initiates the control output functions for this action. Interrupt service routine ISR_ThrottleControl does the critical functions of throttle control. The car decelerates and accelerates as per setting of throttle valve orifice.<br>8. The brake is controlled when the safe distance is not maintained and warning message is flashed on the screen. task_Brake initiates the critical functions of brake control. Interrupt service routine ISR_BrakeControl performs these functions.<br>9. When battery power becomes low, the ACC system deactivates after issuing alarm and flashing messages (notifications). |
| Design metrics | 1. *Power Source and Dissipation*: Car Battery operation.<br>2. *Resolution*: 2 m inter-car distance.<br>3. *Performance*: Safe distance setting 75 to 200 m. No overshooting of controlled output for the throttle from adaptive algorithm.<br>4. *Process Deadlines*: Less than 1 s response on observation of unsafe distance of front-end car.<br>5. *User Interfaces*: Graphic at LCD or touch screen display on LCD and commands for ACC cruise mode ON or OFF, resumption of driver control from ACC, setting the preset cruising speed, alarms of different tones for ACC messages to driver.<br>6. *Extendibility*: The system is extendable to maintain string stability of multiple cars in a row.<br>7. *Engineering Cost*: US$ 50000 (assumed).<br>8. *Manufacturing Cost*: US$ 600 (assumed). |
| Test and validation conditions | 1. Tested in dense as well light traffic conditions.<br>2. Tested on plains, hills and valley roads.<br>3. All user commands must function correctly. |

Following are the detailed functions of an ACC system. Cruise control (CC) is also called auto-cruise control or speed control. It automatically controls the car-speed. The driver presets a speed and the system will take over the control of the throttle of the car. In systems a current control in a solenoid controls the throttle position. A stepper motor with worn drive attachment can be used. An adaptive algorithm calculates and sends the control signals to the stepper motor at the vacuum valve actuator. The orifice opening of the vacuum valve controls the throttle valve. The valve is electro-pneumatic. Vacuum creation provides the force via bellows. [A d.c. or stepper motor with a worn drive attachment to the throttle can also be directly used instead of vacuum actuator and bellows].

Generally, the driver holds the vehicle steady during the drive using the accelerator pedal. Cruise control relieves the driver from that duty and the driver hands over the charge to the CC when the road conditions are suitable (not wet or icy, or there are no strong winds or fog) and, if the car is cruising at high speed, when there is no heavy traffic. The driver resumes the charge in these conditions.

*Adaptive cruise control* (ACC) or *autonomous cruise control* (ACC) or *active cruise control* (ACC) or *intelligent cruise control* (ICC) system is commonly used in aviation electronics and defense aircrafts for cruising since long. Use in the automotives is of recent origin. [Refer to *http://www.ee.surrey.ac.uk/Personal/R.Young/java/html/cruise.html*]. An ACC-system car moves in cruise mode at a preset speed. A radar or laser or UV (ultraviolet) emits signals at regular intervals. These signals are reflected from vehicle in the front. When the reflected signal is received earlier than expected from a minimum safe distance, it notifies the presence of another vehicle to the system. The ACC system then decelerates and the car slows down. It accelerates again to the preset speed when conditions permit. A common method for ACC is only to control speed through throttle position downshifting. New systems also apply the brakes and deploy a CMB system. CMB alerts the driver if front object is less than 100 m distance. For a close object, CMB applies brakes softly and alerts by tugs at the seatbelt. If driver still doesn't react, the system retracts, locks the seatbelt and brakes hard.

A sophisticated ACC system can also maintain *string stability* to control the multiple cars streaming through highways and in case of VIP convoys. [Refer to a paper Chi-Ying Liang and Huei Peng, *"Optimal Adaptive Cruise Control with Guaranteed String Stability"* Journal of Vehicle System Dynamics, *31, pp. 313–330, 1999*].

An adaptive control refers to an algorithm parameters in which *adapt* to the present status of the control inputs in place of a constant set of mathematical parameters in the algorithmic equations. Parameters adapt dynamically. Exemplary parameters, which are adapted continuously, are *proportionality*, *integration* and *differentiation constants*.

Figure 12.10 shows *how an adaptive control algorithm can adapt and function*. It calculates the output values for the control signals. For an ACC system, an adjustable-system subunit generates output control signal for throttle valve. The desired preset cruise velocity $v_t$, desired preset distance $d_{set}$ and safe preset distance $d_{safe}$ are the inputs to index-of-performance measurement-subunit. The measured cruise-velocity v and distance d are also the inputs to index-of-performance computing-subunit. The comparison and decision subunit has inputs of set performance parameters and observed performance parameters. It sends outputs which are inputs to adaptive mechanism subunit. The adaptive mechanism subunit sends outputs, which are inputs to adjustable system. [For details of the control system algorithms, the reader may refer to the standard texts in Control Engineering, namely *Continuous and Discrete Control Systems by John F. Dorsey, McGraw-Hill International Edition, 2002. Digital Control and State Variable Method by Madan Gopal, Tata McGraw-Hill, New Delhi, 1997 and Modern Control Systems – Analysis and Design by Walter J. Grantham and Thomas L. Vincent, John Wiley & Sons, New York, 1993].*

The port devices and their functions are as follows:

1. Port_Align: It is a stepper motor port. Motor steps up clockwise or anticlockwise on a signal from task_Align. The motor aligns the radar or UVHF transmitting device in the lane of front-end car.

2. Port_ReadRange: It is a front-end car range-measuring port. Time difference deltaT is read on a signal from task_Signal to port device. The port radar emits the signals through the antenna and sensor antenna receives the reflected signal from the front-end car. task_ReadRange reads the Port device circuit for the computations of delay period between the transmission and reception instances. Delay period in s multiplied by $1.5 \times 10^5$ m measures the range *rangeNow* (= distance d) in km of the front-end car. task_ReadRange sends message for d to task_RangeRate and all other streaming cars.

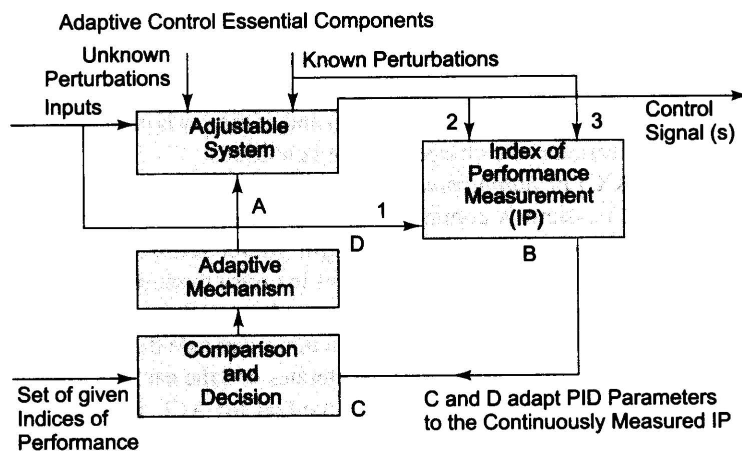



**Fig. 12.10** Model for an adaptive control algorithm adaptation and functions

3. Port_Speed: The port control function routine enables free running counter overflow interrupt. On receiving a signal from task_Speed, the port device reads a free running counter *count0* on first one-bit input from the wheel and also sets a parameter N_rotation = 0. The counts are saved in a memory buffer allotted for the port data. Each successive wheel rotation causes port to note the *countN* at this instance, where countN is the count on $N^{th}$ rotation. Also N_rotation increments by 1 each time *countN* saves. The port control function routine disables when free running counter overflows. After the counter overflows, it finds the difference of the last value of countN in the buffer and *count0*. Current speed v = *speedNow* = (N_rotation × wheel circumference in m)/ [$T_{clock}$ × (*countN* - *count0*) × 1000] km/s. task_Speed sends message to task_RangeRate and the for display controller at the speedometer, for *speedNow*. task_RangeRate then sends messages for the (*rangeNow* – $d_{set}$) and (*speedNow* – $v_{set}$) to task_Algorithm.
4. Port_Brake: Port device applies the brakes or emergency brakes on an interrupt signal. The service routine ISR_BrakeControl disables the interrupts at the beginning and enables on exiting the critical section. It applies the brakes and signals this information to all the other streaming cars also.

Figure 12.11 shows block diagram of units of ACC system. Figure 12.12 shows ACC system cycle of actions and synchronizing cycle of different units.

## 12.3.2 Class Diagrams

Table 12.2 lists the required functions and different tasks. A cycle of actions and synchronization of units showed in Figure 12.12 leads us to a model for the system tasks. ACC system measurements of front end car range, distance and error estimations and adaptive control can be modeled by two class diagrams of abstract classes, Task_ACC and Task_Control. Figure 12.13 shows two class diagrams of Task_ACC and Task_Control and also other extended classes from these classes.

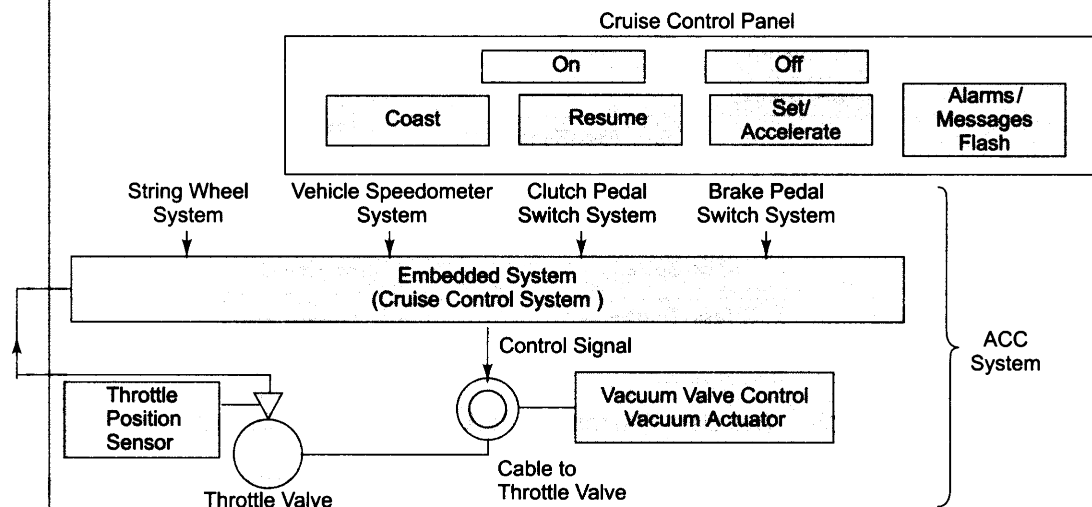



**Fig. 12.11** Block diagram of units of ACC system

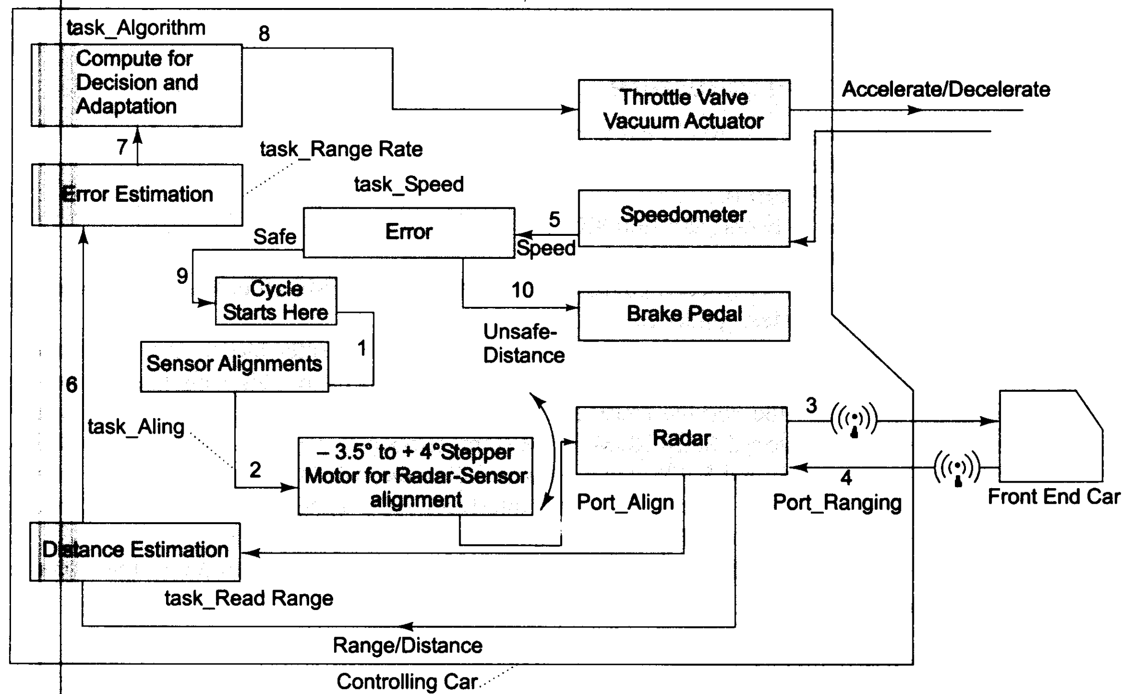



**Fig. 12.12** ACC system cycle of actions and synchronizing cycle of different units

1. Task_ACC is an abstract class from which extended class(es) is derived to measure the range and errors. A task objects is instances of the extended class, which Task_ACC extends. Task_ACC extends to Task_Align, Task_Signal, Task_ReadRange and Task_Algorithm.

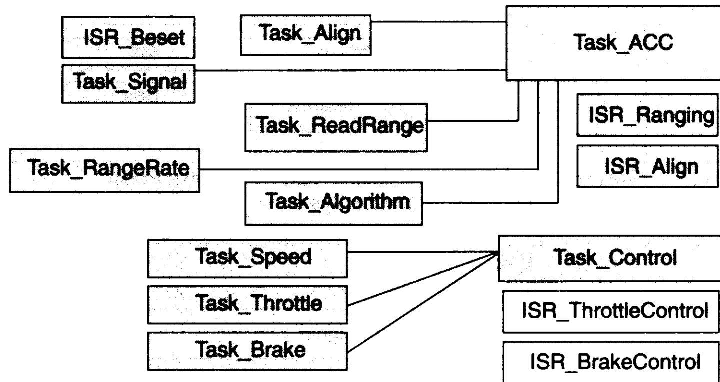



**Fig. 12.13** Two class diagrams of Task_ACC and Task_Control

2. Task_Control is an abstract class from which an extended class is derived to measure the range and errors. The task objects are instances of the classes (i) Task_Brake, (ii)Task_Throttle and (iii) Task_ Speed, which extends from task_Control. Task_Algorithm interfaces Task_Brake, Task_Throttle, Task_Speed and Task_ ReadRange.
3. There are two ISR objects, ISR_ThrottleControl and ISR_BrakeControl.

### 12.3.3 ACC Hardware Architecture

A hardware system in automotive electronics has to provide functional safety. Important hardware standards and guidance at present are the following:

(a) TTP (Time Triggered Protocol)
(b) CAN (Controller Area Network) [Section 3.10.2]
(c) MOST (Media Oriented System Transport)
(d) IEE (Institute of Electrical Engineers) guidance standard exists for EMC (Electromagnetic Magnetic Control) and functional safety guidance.

Figure 12.14 shows hardware subunits in an ACC system. An automotive embedded system-based control unit uses microcontroller and separate microprocessor or DSP. ACC embeds the following hardware units:

1. A microcontroller runs the service routines and tasks (Figure 12.12) except task_Algorithm. Microcontroller has the internal RAM/ROM. RAM stores temporary variables and stack. ROM/Flash saves the application codes and RTOS codes for scheduling the tasks. CAN port interfaces with the CAN bus (Section 3.10.2) at the car. The CAN interfaces ACC system with the other embedded systems of the car. Interrupt controller at microcontroller control the interrupts.
2. A separate processor with RAM and ROM for the task_Algorithm executes the adaptive control algorithm (Figure 12.10).
3. Speedometer
4. Stepper motor-based alignment unit.
5. Stepper motor-based throttle control unit.
6. Transceiver for transmitting pulses through an antenna hidden under the plastic plates.
7. LCD dot matrix display controller, display panel with buttons.
8. Port devices are Port_Align, Port_Speed, Port_ReadRange, Port_Throttle and Port_Brake. These five port devices are used for five actions as follows: aligning transmitted beam toward the lane, measuring speed *v*, range *d* at the ACC, throttle positioning (as per the control messages fro task_Algorithm) and braking action (as per messages fro ISR_BrakeControl).

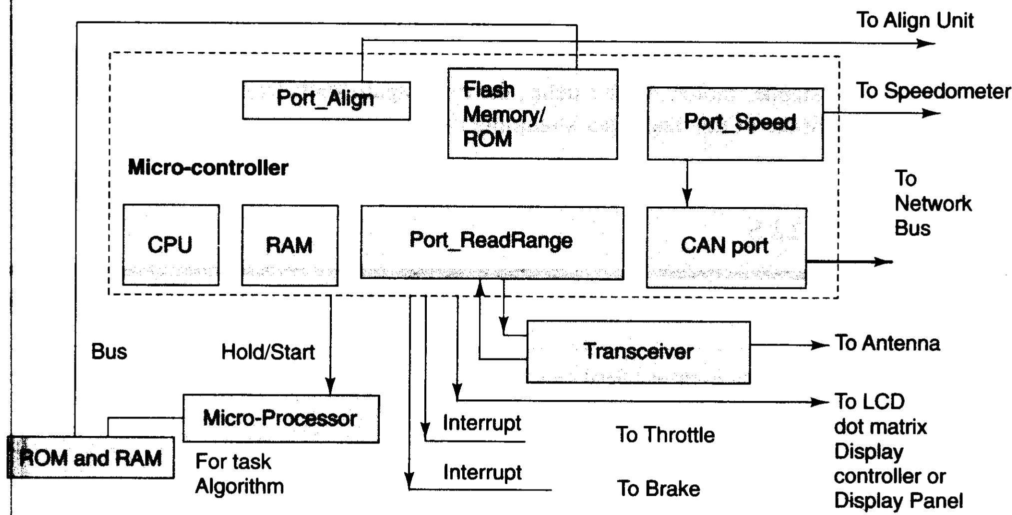



Fig. 12.14 ACC hardware

## 12.3.4 ACC Software Architecture

OSEK (Section 10.2) mentioned that tasks can be classified into four types and a programmer must have a clear-cut distinction: which class to use for what modules in the system. For the ACC system, Basic Conformance Class 1 (BCC1) is used. Table 12.3 lists the BCC1 tasks, functions and IPCs, which are needed for the ACC with string stability algorithm. OSEK tasks can consist of three types of objects, *event* (semaphore), *resource* (statements and functions) and devices including port devices. The columns 5 and 4 give of signal and semaphore IPCs posted and taken. The semaphores are used for defining the sequence of running of tasks and cyclically running the ACC tasks and executing the string stability controller algorithm at task_Algorithm.

## 12.3.5 ACC Software tasks Synchronization Model

Figure 12.12 showed the units, tasks of ACC system, cycle of actions and synchronizing cycle of different units. ACC system cycle of actions and task scheduling model. Note the marks 1 to 10 of cycle starting and finishing in the figure.

1. Cycle starts from a task, task_Align on an event (ISR call). It sends the signal to a stepper motor port Port_Align and Port_Ranging. The stepper motor moves by one step clockwise or anticlockwise as directed.
2. A task task_ReadRange is for measuring front-end car range. The task disables all interrupts as it is entering a critical section. We need real-time measurement. Port_ReadRange finds d.
3. task_Speed gets the port reading at a port Port_Speed. Task sends v, using the countN and count0 interval between the initial and $N^{th}$ rotation.
4. task_RangeRate sends the *rangeNow*. It estimates the final error in maintaining the string stability from task_ReadRange output. It estimates of the error for maintaining the car speed from the task_Speed output. It outputs both error values for use by the control system adaptive algorithm. Port_Speed connects to the speedometer system of the DAS, which displays speedNow after appropriate filtering function. Port_RangeRate transmits the *speedNow* also to other streaming cars also. Now Task calculates range and rate errors, and transmits both *rangeNow* and *speedNow*.

5. task_Algorithm runs the main adaptive algorithm. It inputs are as follows. It gets inputs from task_RangeRate. The task outputs are events to Port_Throttle and brake. Port_Throttle attaches to the vacuum actuator stepper motor. After a delay, the cycle again starts from the task_Align. It reads the statuses of Port_Brake of this and other streaming cars.

**Table 12.3** List of BCC 1 task Functions and IPCs objects of the Classes shown in Figure 12.5

| *BCC 1 task Function* | *BCC 1 Priority* | *Action* | *IPCs pending* | *IPCs posted* | *Input* | *Output* |
|---|---|---|---|---|---|---|
| task_Align | 101 | Starts on Event and Send signal to Port_Align and Port_Ranging | Reset | Align | deltaStep, step | step |
| task_Read-Range | 103 | Disable interrupts, gets signal from Port. Port activates a radar flashing, records activation time, gets time of sensing the reflected radar signal and finds time difference, deltaT. Enable interrupts. | Align | Range | – | deltaT |
| task_Speed | 105 | Event Port_Speed to start a timer, counter start message and wait for the 10 counts for the number of wheel rotations. Event outputs deltaT from port. | – | Speed | – | speedNow |
| task_Range-Rate | 107 | It calculates *rangeNow.* Get preset front car *range* and stringRange from memory and compare. Get preset cruising speed, $v_{set}$; compare it with current speed *speedNow.* | Speed | ACC | avgTire Circum, time-Diff, deltaT, string Range, Speed, Cruise N_rotation | range-Error, speed-Error, range-Now, speed-Now |
| task_Algorithm | 109 | (i) Get errors of speed and range and execute adaptive control algorithm. (ii) Get errors of other vehicles through Port_RangeRate. (iii) Get other vehicles Port_Brake status. (iv) Get present throttle position. (v) Send output, throttleAdjsut to Port_Throttle. (vi) Send signal to Port_Brake in case of emergency braking action needed. (vii) Port_Brake transmits the action needed to other vehicles also. | ACC | – | range-Error, speed-Error, All Port_RangeRate values and Port_Brake statuses, VehicleID, | throttle-adjust, emergency |

[1]Basic task with one task of each priority and single activation. It is called BCC 1 (Basic Conformance Class 1)

task_Algorithm generates output for ACC action. Output port for control signals generates event for a port with a throttle valve. The signals at this port, Port_Throttle, are calculated as follows: An adjustable algorithm, *A*, gets inputs of the speed '*speed*' and front-object range '*range*' as well as the unknown and unknown perturbations, P_Unknown and P_Known. It adjusts the output signal to Port_Throttle. An algorithm, *B*, estimates the index of performance *IP*. An algorithm, *C*, compares *IP with a set of given IP* values. Algorithm *D* is as per the adaptive control that adapts the output of *C*. *C* sends the new parameters that are to be adapted by *A*.

Figure 12.15 shows a synchronization model for ACC tasks and semaphores.

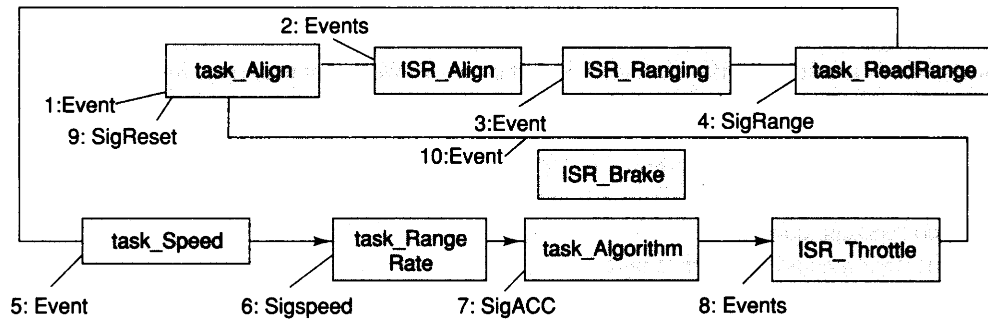



**Fig. 12.15** Synchronization Model for ACC tasks

## 12.3.6 ACC Software Implementation

ACC software for use in automobiles must first be certified by an organization authorized to issue that certification. We have seen that OSEK OS standard is required [Section 10.2]. Only those VxWorks or MUCOS functions that adhere to OSEK must be used. Software coding IEC 61508 part 3 and MISRA C version 2 (2004) specifications of safety standards and coding language *must be used*. [MISRA stands for Motor Industry Reliability Association.] MISRA C specifies a collection of rules to be used while coding in C.

MISRA-C is a standard for C language software and defines the guidelines for automotive systems. MISRA-C version 2 (2004) specified 141 rules for coding and gave a new structure for C. Details can be found at http://www.misra.org.uk. Figure 12.16 shows important rules and coding standards in MISRA-C.

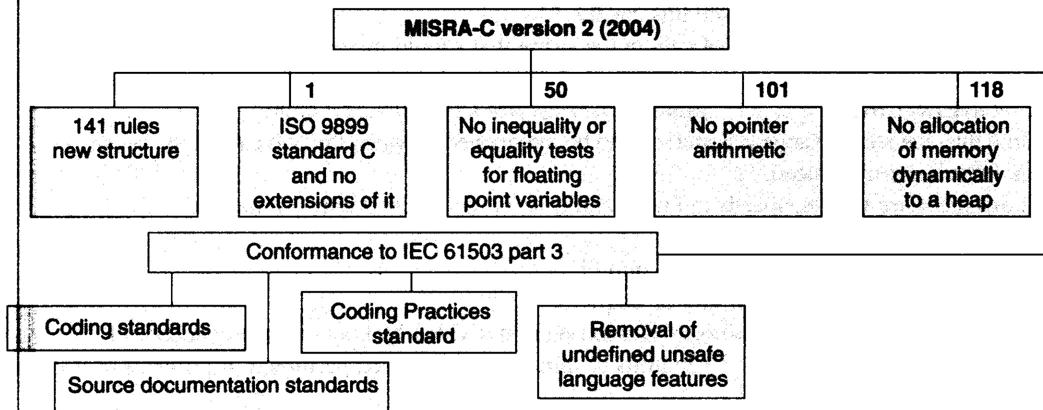



**Fig. 12.16** Important rules and coding standards in MISRA-C

A few rules are discussed next. Its *first rule* is that all C codes used in automobiles must conform to ISO 9899 standard C and no extensions of it should be permitted. Rule 43 does not permit use of implicit cast that may result in a loss of information. Rule number 50 does not permit inequality or equality tests for floating point variables. Floating-point calculations undergo rounding off errors. The logic for introducing this rule is as follows: Consider 'If ((1/3) * 3 = 1) then …". 1/3 = 0.33333 with 3 or 4 in the last digit and the result is always uncertain. Rule number 65 does not permit use of floating-point numbers as a loop counter. Rule 101 does not permit use of pointer arithmetic. It is similar to the rule in Java. Rule 118 does not permit allocation of memory dynamically to a heap. Dynamic allocation has risk of using additional memory allocation than available in the system, which may cause memory leaks.

We have already learnt VxWorks (Section 9.3). Let us use VxWorks for understanding code implementation for the ACC system. To demonstrate ACC application, let us see the application of VxWorks and how we adapt OSEK in the following exemplary codes (Example 12.2).

1. Use BCC 1 type of tasks. as shown in VxWorks code application in Example 12.2. We define each task of different priority and activate it only once in the codes.
2. Use no message queues.
3. Use no creation and deletion of task.
4. Use semaphores as event flag only with no task having run-time deletion or creation of these.
5. Use MISRA C rules in coding.
6. Use disable interrupts when a task or function enters critical section and enable interrupts when leaving critical section.

## Example 12.1

```
1. # include "vxWorks.h" /* Include VxWorks functions. */
# include "semLib.h" /* Include semaphore functions library. */
# include "taskLib.h" /* Include multitasking functions library. */
# include "sysLib.c" /* Include system library for system functions. */
#include "sigLib.h" / * Initialize kernel component for signal functions. */
2. /* Set system clock rate 10000 ticks per second. Every 100 µs per tick*/
sysClkRateSet (10000);
3. /* Declare and Initialize Global parameters. */
unsigned byte VehicleID; /* Declare this car ID. */
static numCars = ...; /* Numbers of cars in the string that should move at cruise speed. */
static unsigned byte N_rotation = ....; /* Initialize number of rotations needed when finding the speed. */
static int avgTireCircum = ... ; /* Initialize average tire Circumference in mm. */
4. /* Initialize the string Range = Separation to be maintained between the two cars in the string
in mm. Initialize cruise speed. */
/* All distances are in mm, speeds in km/hr and time in nanosecond unless otherwise
specified. */
static int stringRange = ...; /* In unit of mm */
static int CruiseSpeed = ...; /* In unit of km/hr */
static float unitChange = 3600000.0; /* Km in one mm divided by hours in one nanosecond. */
float permittedSpeedError = ... ; / * In units of mm/ns error in speed permitted. It prevents oscillations in
the control system. */
static boolean alignment = false; /* Declare alignment = false to initiate radar transmitter alignment. */
```

5. /* Other Variables. */
static byte *step; /* Stepper motor step angle in degrees. */
static byte * deltaStep = 0; /* Stepper motor step angle change in degrees. */
/* Time difference between emitted signal and reflected signal from front-end car. rangeNow is present range in mm. It is timeDiff multiplied by speedNow after a filter function application. */
static unsigned long *timeDiff; static int *rangeNow;
static unsigned long *deltaT; /* *Time interval for N rotation in ns.* */
6. /* Declare pointers for variables range error, range now, speed error, speed now. */
int rangeError, speedError, *rangeNow* = 0, *speedNow* =0; /* Speeds are in km/hr and range in mm. */
7. /* Declare arrays of size number of cars, numCars. These many cars are running as a string. Declare brakeStatus, RangeErrors, SpeedErrors, Ranges, Speeds for all the cars. */
boolean brakeStatus [numCars];
int RangeErrors [numCars], Ranges [numCars], SpeedErrors [numCars], Speeds [numCars];
boolean emergency [numCars]; /* Declare variable for emergency message sent to Port_Brake of $N^{th}$ car. */
int *throttleAdjsut; /* Declare variable for throttle adjusting parameter */
8. /* Other Variables. */
.
.
9. /* Declare all Table 12.3 ***task function prototypes***. */
void task_Alignment (SemID *SigReset*, SemID *SigAlign*, byte *step, byte *deltaStep); /*task for aligning stepper motor in front-end car view. */
void task_ReadRange (SIGID *SigAlign*, SIGID *SigRange*, unsigned long **timeDiff*); /*task for receiving the timeDiff using the radar for calculating *rangeNow*. */
void task_Speed (SIGID *SigRange*, SIGID *SigSpeed*, unsigned long *deltaT); /*task for receiving the deltaT using the wheel counter and timer for calculating *speedNow*. */
void task_Range_Rate (SIGID *SigSpeed*, SIGID *SigReset*, SIGID SigACC, int *avgTireCircum*, unsigned byte N_rotation, int *CruiseSpeed*, int *stringRange*, unsigned long * *time-Diff*, unsigned long **deltaT*, int * range-Error, int *speedError, int **range-Now*, int **speed-Now*); /*task for calculating rangeNow, speedNow, rangeError, speedError. */
void task_Algorithm (SIGID *SigACC*, SIGID *SigReset*, boolean brakeStatus [numCars],
int RangeErrors [numCars], int Ranges [numCars], int SpeedErrors [numCars], int Speeds [numCars], boolean emergency [numCars], unsigned byte VehicleID); /* Declare array for emergency message sent to Port_Brake of $N^{th}$ car. */
int *throttleAdjsut; /* Declare variable for throttle adjusting parameter */
10. /* Declare all Table 12.3 ***task IDs, Priorities, Options and Stacksize***. Let initial ID, till spawned (initiated) be none. No options and Stacksize = 4096 for each of six tasks. */
int task_AlignID = ERROR; int task_AlignPriority = 101; int task_AlignOptions = 0; int task_AlignStackSize = 4096;
int task_ReadRangeID = ERROR; int task_ReadRangePriority = 103; int task_ReadRangeOptions = 0; int task_ReadRangeStackSize = 4096;
int task_SpeedID = ERROR; int task_SpeedPriority = 105; int task_SpeedOptions = 0; int task_SpeedStackSize = 4096;
int task_RangeRateID = ERROR; int task_RangeRatePriority = 107; int task_RangeRateOptions = 0; int

task_RangeRateStackSize = 4096;
int task_AlgorithmID = ERROR; int task_AlgorithmPriority = 109; int task_AlgorithmOptions = 0; int task_AlgorithmStackSize = 4096;
11. /* ***Create and Initiate (Spawn) all the six tasks*** of Table 12.3. */
*task_AlignID* = taskSpawn (" ttask_Align", task_AlignPriority, task_AlignOptions, task_AlignStackSize, void (*task_Alignment) (SemID *SigReset*, SemID *SigAlign*, byte *step, byte *deltaStep), 0, 0, 0, 0, 0, 0, 0, 0, 0, 0);
*task_ReadRangeID* = taskSpawn (" ttask_ReadRange", task_ReadRangePriority, task_ReadRangeOptions, task_ReadRangeStackSize, void (*task_ReadRange) (SIGID *SigAlign*, SIGID *SigRange*, unsigned long **timeDiff*), 0, 0, 0, 0, 0, 0, 0, 0, 0, 0);
*task_SpeedID* = taskSpawn (" ttask_Speed", task_SpeedPriority, task_SpeedOptions, task_SpeedStackSize, void (*task_Speed) (SIGID *SigRange*, SIGID *SigSpeed*, unsigned long *deltaT), 0, 0, 0, 0, 0, 0, 0, 0, 0, 0);
*task_RangeRateID* = taskSpawn (" ttask_Range_Rate", task_RangeRatePriority, task_RangeRateOptions, task_RangeRateStackSize, void (*task_Range_Rate) (SIGID *SigSpeed*, SIGID *SigReset*, SIGID *SigACC*, int *avgTireCircum*, unsigned byte N_rotation, int *CruiseSpeed*, int *stringRange*, unsigned long * *time-Diff*, unsigned long **deltaT*, int * range-Error, int *speedError, int **rangeNow*, int **speedNow*), 0, 0, 0, 0, 0, 0, 0, 0, 0, 0);
*task_AlgorithmID* = taskSpawn ("ttask_Algorithm", task_AlgorithmPriority, task_AlgorithmOptions, task_AlgorithmStackSize, void (*task_Algorithm) (SIGID *SigACC*, Sig_ID *SigReset*, boolean brakeStatus [numCars], int RangeErrors [numCars], int Ranges [numCars], int SpeedErrors [numCars], int Speeds [numCars], boolean emergency [numCars], unsigned byte VehicleID), 0, 0, 0, 0, 0, 0, 0, 0, 0, 0);
12. /* Declare IDs and create binary semaphore event flags. */
SIGID *SigAlign*, *SigRange*, *SigSpeed*, *SigACC*, *SigReset*; /* Declared for Table 12.3 five tasks. */
13. /* Create the binary semaphores taken in FIFO as empty to start with. */
*SigAlign* = semBCreate (SEM_Q_FIFO, SEM_EMPTY);
*SigRange* = semBCreate (SEM_Q_FIFO, SEM_EMPTY);
*SigSpeed* = semBCreate (SEM_Q_FIFO, SEM_EMPTY);
*SigACC* = semBCreate (SEM_Q_FIFO, SEM_EMPTY);
*SigReset* = semBCreate (SEM_Q_FIFO, SEM_EMPTY);
14. /* Declare function for starting an RTC timer at the Port_Ranging. */
void RTCtimer_Port_Ranging_Start ( ) {
.
.
};
15. /* Declare a function for starting an RTC timer at the Port_Speed. */
void RTCtimer_Port_Speed_Start ( ) {
.
.
};
16. /* Declare a function to read 64-bit time from an RTC. */
unsigned long timer_gettime (&RTC) {
.
.
};

```
17. /* Define a macro for calculating time between instance when control bit = true and Status flag = true.
*/
boolean * CB; /* Control bit */
boolean * SF; / * Status flag. */
*CB =0; /* Control bit = 0; */
*SF = 0; /* Status flag = 0; */
unsigned short * RTC; /* Pointer to a real-time clock. */
#define unsigned long calculate_TimeInterval (unsigned short * RTC, boolean * CB, boolean *SF) (
unsigned long timeInstance0, timerInstance1;
while (*CB != 1 && *SF = = 0) { }; /* Wait for read instruction to timer. */
timeInstance0 = timer_gettime (&RTC); /* Find initial time at start in the timer */
while (*SF ! = 1) { }; /* Wait for sensor status flag to be true */
timeInstance1 = timer_gettime (&RTC); /* Find initial time at start in the timer */
*SF =0; *CB =0;
return (timeInstance1 – timrInstance0);
) /* End of Macro to calculating time interval between two instances specified by CB and SF becoming
true. */
18. /* Define a macro for calculating time interval deltaT between instance when control run bit = true and
instance of Nth count input. */
boolean CR; /* Control Run bit. */
boolean countInput; / *Status flag. */
*CR =0; /* Control run bit = 0; */
*countInput = 0; /* countInput initial value. */
#define unsigned long DelT (unsigned short &RTC, boolean *CR, N_rotation, boolean * countInput)(
unsigned long timeInstance = 0; byte N = 0;
for (byte N =0; N < N_rotation; N++) {
while (*CR != 1 && *countInput != 0) { }; /* Wait for count input true. */
timeInstance += timer_gettime (&RTC); /* Find initial time at start in the timer */
*countInput =0; /*Reset countInput. It will set on start of next rotation. */
}; *countInput =0;
return (timeInstance);
) /* End of Macro to calculate time interval for N count inputs. */
19. /* Declare Macro for sending a byte for step angle setting to Port_Align. */
unsigned short * Port_Align; /* Declare a pointer for Port_Align. */
# define Out_Alignment (&Port_Align, Step) / * Codes in Assembly Language for stepper motor routine.
* /
.
.
); /* End of Macro to sending byte for step angle to Port_Align. */
20. /* Declare Macro to find timeDiff from Port_Ranging. */
unsigned short RTC_Port_Ranging =...; /* Declare Address of RTC at Port_Ranging. * /
RTCtimer_Port_Ranging_Start ( );
#define RANGE (unsigned short * Port_Ranging, unsigned long * timeDiff) (
unsigned short * RTC_Port_Ranging = .....; /* Declare address of RTC of Port_Ranging. */
```

```
intLock ( ); /*disable interrupts. */
/ * Codes in Assembly Language for Ranging routine for Start Radar transmission by making control bit
CB = 1* /
*CB = 1;
*timeDiff = calculate_TimeInterval (&RTC_Port_Ranging, &CB, &SF);
intUnlock ( ); /* Enable Interrupts. */
) /* End of Macro to find time interval for reflected radar signals. */
21. /* Declare Macro to find deltaT from Port_Speed. */
unsigned short RTC_Port_Speed =...; /* Declare Address of RTC at Port_Speed. * /
RTCtimer_Port_Speed_Start ( );
#define SPEED (unsigned short * Port_Speed, unsigned long * deltaT) (
unsigned short * RTC_Port_Ranging = .....; /* Declare address of RTC of Port_Ranging.*/
intLock ( ); /*disable interrupts. */,
/ * Codes in Assembly Language for Speed routine for Start counting the tire rotation count inputs by
making control bit CR = 1* /
*CR = 1;
*deltaT = DelT (&RTC_Port_Speed, &CR, N_rotation, &countInput);
intUnlock ( ); /* Enable Interrupts. */
) /* End of Macro to find time interval for N count-inputs. */
22.
# define float filter_speed (float calculatedSpeed, int *speedNow, float permittedSpeedError) (/* Codes
for filtering the calculated speed. If the calculated value in mm/ns is within plus minus limit of
permittedSpeedError, do not change it; else modify it with new value. */
float a;
a = (float) (speedNow/ unitChange);
if (calculatedSpeed > a + permittedSpeedError | | calculatedSpeed < a – permittedSpeedError) {return
(calculatedSpeed);} else return (a);
) /* End of macro for filtering the speed calculated to prevent vibrations and oscillation in controlling
vehicle. * /
23.
#define RangeRate (unsigned short * Port_RangeRate, int *speedNow, int *rangeNow, int *speedError,
int *rangeError, unsigned byte VehicleID) (
/ *Assembly codes for sending to Port_RangeRate and transmitting the range and rate parameters and
vehicleID. Port_RangeRate sends also *speedNow for display on speedometer at Port_Speed. /
.
.
); /* End of Macro to transmit range and rate parameters through Port_RangeRate. */
24. to 26. /* Code for Other Declaration Steps specific to various functions */
.
.
/* End of the codes for creation of tasks, semaphores, message queue, pipe tasks, and variables and all
needed function declarations */
/*****************************************************************************/
27. /* Start of Codes for task_Alignment. */
```

```
void task_Alignment (SemID SigReset, SemID SigAlign, byte *step, byte *deltaStep) {
.
28.
while (1) { /* Start of while loop. */
/* When cycle starts the semaphore is given. Wait for it. */
semTake (SigReset, WAIT_FOREVER); /* Wait for Cycle Reset Event flag. */
29. /* Codes for sending the step to the address of Port_Ranging. *
*step = *step + *deltaStep;
Out_Alignment (&Port_Align, *Step);
semGive (SigAlign);
}; /* End of while loop. */
} /* End of task_Alignment. */
/*****************************************************************************/
30. /* Start of codes for task_ReadRange. */
void task_ReadRange (SIGID SigAlign, SIGID SigRange, unsigned long *timeDiff) {
31. /* Initial assignments of the variables and pre-infinite loop statements that execute only once */
static unsigned short Port_Ranging = ....; /* Declare pointer to Port_Ranging. */
.
.
32. while (1) { /* Start an infinite while-loop. We can also use FOREVER in place of while (1). */
33. / *Wait for SigAlign state change to SEM_FULL by semGive function. */
semTake (SigAlign, WAIT_FOREVER);
/*Send signal to Port_Ranging. Port activates a radar flashing, records activation time, gets time of
sensing the reflected radar signal and finds time difference, timeDiff. Enable interrupts. */
RANGE (& Port_Ranging, &timeDiff);
);
semGive (SigRange);
};
34. } /* End of the codes for task_ReadRange. */
/*****************************************************************************/
35. /* Start of codes for task_Speed. */
void task_Speed (SIGID SigRange, SIGID SigSpeed, unsigned long *deltaT) {
36. /* Initial assignments of the variables and pre-infinite loop statements that execute only once */
static unsigned short Port_Speed = ...; /* Declare pointer to Port_Speed. */
.
.
37. while (1) { /* Start task infinite loop. */
semTake (SigRange, WAIT_FOREVER);
/* Codes for receiving the deltaT using the wheel counter and timer for later on calculating speedNow. */
SPEED (& Port_Speed, &deltaT);
semGive (SigSpeed);
    }; / * End of while loop. */
38. } /* End of codes for task_Speed */
/*****************************************************************************/
```

```
/* Start of codes for task_Range_Rate. */
void task_Range_Rate (SIGID SigSpeed, SIGID SigReset, SIGID SigACC, int avgTireCircum, unsigned
byte N_rotation, int CruiseSpeed, int stringRange, unsigned long * time-Diff, unsigned long *deltaT, int *
range-Error, int *speedError, int *range-Now, int *speed-Now) {
static unsigned short Port_RangeRate = ...; /* Declare pointer to Port_Ranging. */
.
.
39. while (1) { /* Start task infinite loop . */
semTake (SigRange, WAIT_FOREVER);
40. /* .Codes for calculating rangeNow, speedNow, rangeError, speedError. */
*speedNow = (int) (unitChange * filter_speed ((float) (avgTireCircum * N_rotation) / (float) (*deltaT),
int *speedNow, float permittedSpeedError)));
*rangeNow = (*speedNow/unitChange) * (*timeDiff)/2.0; /* Divide by 2 because reflected signal travels
twice the distance in mm/ns. */
*speedError = cruiseSpeed - *SpeedNow;
RangeRate (& Port_RangeRate, *speedNow, *rangeNow, *speedError, *rangeError, VehicleID); / *Send
the parameters for transmission to other vehicles and Port_Speed for displays. */
if (alignment != true) {semGive (SigReset);
41. /* Code for loop of tasks of priorities 101, 103, 105 in which the values of rangeNow are calculated at
different step values by changing deltaStep and, finally, at that instance, alignment is declared as true, for
rangeNow is minimum. Front-end vehicle is now in line of sight. */
.
.
} else semGive (SigACC); /* After alignment is perfect the control algorithm is sent the event flag. */
}; / * End of while loop. */
42. } /* End of codes for task_Range_Rate. */
/*****************************************************************************/
/* Start of Codes for task_Algorithm. */
void task_Algorithm (SIGID SigACC, SIGID SigReset, boolean brakeStatus [numCars],
int RangeErrors [numCars], int Ranges [numCars], int SpeedErrors [numCars], int Speeds [numCars],
boolean emergency [numCars], unsigned byte VehicleID) {
.
.
43. while (1) { /* Start task infinite loop . */
semTake (SigACC, WAIT_FOREVER);
44. /* Assembly codes for getting errors of other vehicles through Port_RangeRate and other vehicles
Port_Brake statuses through Port_Brake. */
.
.
45. /* Assembly codes to read throttle position from Port_Throttle. */
.
.
46. /* Codes for cruise speed adaptive algorithm and code for string stability maintaining adaptive algorithm
to generate appropriate throttleAdjsut signal to Port_Throttle. */
.
.
```

```
47: /* Codes for Port_Brake action, if emergency = true. Port_Brake transmits the action needed
to other vehicles also. */
if (emergency [numCars] = =1) {
.
} else semGive (SigACC); /* After alignment is perfect the control algorithm is sent the
event flag. */
48; }/ * End of while loop. */
49; } /* End of codes for task_Algorithm. */
/*****************************************************************/
```

## 12.4 CASE STUDY OF AN EMBEDDED SYSTEM FOR A SMART CARD

Section 1.10.3 introduced the smart card system hardware and software. Section 12.4.1 gives the requirements and functioning of smart card communication system. Section 12.4.2 gives the class diagram. Figure 1.13 showed smart card-system hardware components for a contact-less smart. Sections 12.4.3 and 12.4.4 give the hardware and software architecture and synchronization model. Section 12.4.5 gives the exemplary codes.

### 12.4.1 Requirements

Assume a contact-less smart card for bank transactions. Let it not be magnetic. [The earlier card used a magnetic strip to hold the nonvolatile memory. Nowadays, it is EEPROM or flash that is used to hold nonvolatile application data.] Requirements of smart card communication system with a host can be understood through a requirement-table given in Table 12.4.

### 12.4.2 Class Diagram

Table 12.4 listed the functions and the different tasks. An abstract class is Task_CardCommunication. Figure 12.17 shows the class diagram of Task_CardCommunication. A cycle of actions and card-host synchronization in the card leads us to the model Task_CardCommunication for system tasks. Card system communicates to host for identifying host and authenticating itself to the host. ISR1_Port_IO, ISR2_Port_IO and ISR3_Port_IO are interfaces to the tasks. [A class gives the implementation methods of the interfacing routines.] The task_Appl, task_PW, task_ReadPort, and resetTask are the objects of Task_Appl, Task_PW, Task_ReadPort and Task_Reset, respectively. These classes are extended classes of abstract class Task_CardCommunication.

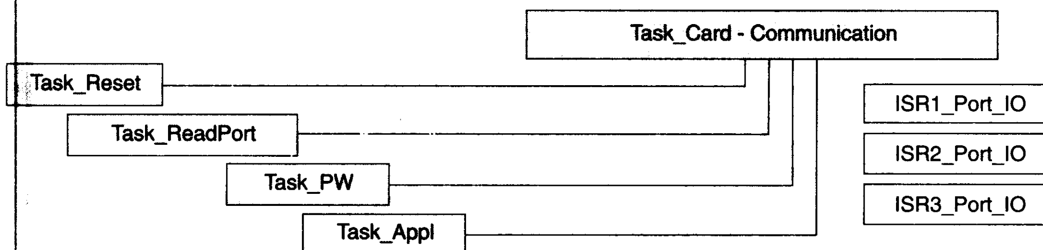



Fig. 12.17 Class diagrams of Task_CardCommunication

**Table 12.4 Requirements of smart card communication system with a host**

| *Requirement* | *Description* |
|---|---|
| Purpose | 1. Enabling authentication and verification of card and card holder by a host and enabling GUI at host machine to interact with the card holder/user for the required transactions; for example, financial transactions with a bank or credit card transactions. |
| System Functioning | 1. The card inserts at host machine. The radiations from the host activate a charge pump at card. The charge pump powers the SoC circuit, which consists of card processor, memory, timer, interrupt handler and Port_IO.<br>2. On power up, system-reset signals resetTask to start. The resetTask sends the messages—*requestHeader* and *requestStart* for waiting task task_ReadPort.<br>3. task_ReadPort sends requests for host identification and reads through the Port_IO the host-identification message and request from host for card identification.<br>4. The task_PW sends through Port_IO the requested card identification after system receives the host identity through Port_IO.<br>5. The task_Appl then runs required API. The *requestApplClose* message closes the application.<br>6. The card can now be withdrawn and all transactions between card-holder/user now takes place through GUIs using at the host control panel (screen or touch screen or LCD display panel). |
| Inputs | 1. Received header and messages at IO port *Port_IO* from host through the antenna. |
| Signals, Events and Notifications | 1. On power up by radiation-powered charge-pump supply of the card, a *signal* to start the system boot program at resetTask.<br>2. Card start *requestHeader* message to task_ReadPort from resetTask.<br>3. Host authentication request *requestStart* message to task_ReadPort from resetTask to enable requests for Port_IO.<br>4. *UserPW* verification message (notification) through Port_IO from host.<br>5. Card application close request *requestApplClose* message to Port_IO. |
| Outputs | Transmitted headers and messages at Port_IO through antenna. |
| Control Panel | No control panel is at the card. The control panel and GUIs activate at the host machine (for example, at ATM or credit card reader). |
| Design metrics | 1. *Power Source and Dissipation*: Radiation powered contact-less operation.<br>2. *Code size*: Code-size generated should be optimum. The card system memory needs should not exceed 64 kB memory. Limited use of data types; multidimensional arrays, long 64-bit integer and floating points and very limited use of the error handlers, exceptions, signals, serialization, debugging and profiling.<br>3. *File system(s):* Three-layered file system for the data. One file for the *master file* to store all file headers. A header has strings for file status, access conditions and file-lock. The second file is a *dedicated file* to hold a file grouping and headers of the immediate successor elementary files of the group. The third file is the *elementary file* to hold the file header and file data.<br>4. *File management:* There is either a fixed length file management or a variable file length management with each file with a predefined offset.<br>5. *Microcontroller hardware*: Generates distinct coded physical addresses for the program and data logical addresses. Protected once writable memory space.<br>6. Validity: System is embedded with expiry date, after which the card authorization through the hosts disables.<br>7. *Extendibility*: The system expiry date is extendable by transactions and authorization of master control unit (for example, bank servee).<br>8. *Performance*: Less than 1 s for transferring control from the card to host machine. |

(*Contd*)

| Requirement | Description |
|---|---|
| | 9. *Process Deadlines*: None.<br>10. *User Interfaces*: At host machine, graphic at LCD or touchscreen display on LCD and commands for card holder (card user) transactions.<br>11. *Engineering Cost*: US$ 50000 (assumed).<br>12. *Manufacturing Cost*: US$ 1 (assumed). |
| Test and validation conditions | 1 Tested on different host machine versions for fail proof card-host communication. |

1. Task_CardCommunication is an abstract class from which extended to class (es) derive to read port and authenticate. The tasks (objects) are the instances of the classes Task_Appl, Task_Reset, Task_ReadPort and Task_ReadRange.
2. Task_ReadPort interfaces ISR1_Port_IO.
3. The task_PW is object of Task_PW and interfaces ISR2_Port_IO. Task_Appl interfaces ISR3_Port_IO.

### 12.4.3 Hardware and Software Architecture

Smart card hardware was introduced in Section 1.10.3. Figure 12.18 shows hardware inside the card.

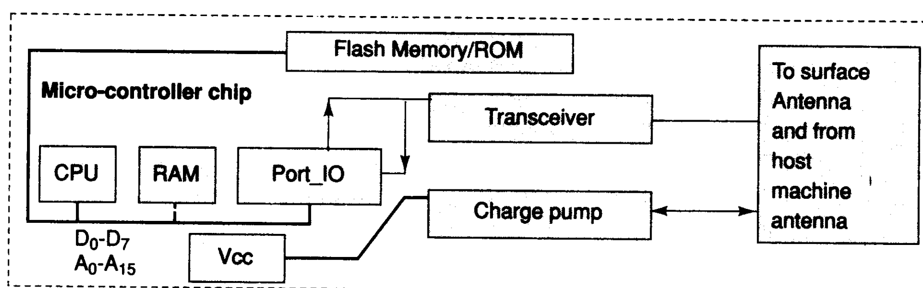



**Fig. 12.18** Smart card hardware

Software using Java Card™ provides one solution. [Refer 5.7.5] JVM has thread scheduler built in. No separate multitasking OS is thus needed when using Java because all Java byte codes run in JVM environment. Java provides the features to support (*i*) security using class java.lang.SecurityManager), (ii) cryptographic needs (package java.security*). Java provides support to connections, datagrams, IO streams and network sockets. Java mix is a new technology in which the native applications of the card run in C or C++ and downloadable applications run in Java or Java Card™. The system has OS and JVM both.

SmartOS is an assumed hypothetical OS in this example, as RTOS in the card. Remember that a similar OS function name is used for understanding purposes identical to MUCOS but actual SmartOS has to be different from MUCOS. Its file structure is different. It has two functions as follows: The function unsigned char [ ] SmartOSEncrypt (unsigned char *applStr, *EnType type*) encrypts as per encryption method, *EnType* = "RSA" or "DES" algorithm chosen and returns the encrypted string. The function unsigned char [ ] SmartOSDecrypt (unsigned char *Str, DeType *type*) encrypts as per deciphering method, DeType = "RSA" or "DES" algorithm chosen and returns the deciphered string. SmartOSEncrypt and SmartOSDecrypt execute after verifying the access conditions from the data files that store the keys, PINs and password.

Table 12.5 gives the tasks for the card OS in this case study.

### 12.4.4 Synchronization Model

Following are the actions on the card placed near the host machine antenna in a machine slot.

Step *1:* Receive from the host, on card insertion, the radiation of carrier frequency or clock signals in case of contact with the card. Extract charge for the system power supply for the modem, processor, memories and Port_IO (card's UART port) device.

**Table 12.5** List of tasks, Functions and IPCs

| *Task Function* | *Priority* | *Action* | *IPCs pending* | *IPCs posted* | *String or System or Host input* | *String or System or Host Output* |
|---|---|---|---|---|---|---|
| resetTask | 1 | Initiates system timer ticks, creates tasks, sends initial messages and suspends itself. | None | SigReset, MsgQStart | SmartOS call to the main | *request-Header*; *requestStart* |
| task_Read Port | 2 | Wait for resetTask Suspenson, sends the queue messages and receives the messages. Starts the application and seeks closure permission for closing the application. | SigReset, Messages from MsgQStart, MsgQPW, MsgQAppl, MsgQAppl-Close | SePW | Functions Smart OS-Encrypt, SmartOS-decrypt, ApplStr, Str, close-Permitted | *request-password,* request-*Appl*, request-ApplClose |
| task_PW | 3 | Sends request for password on verification of host when SemPW = 1. | SemPW | *MsgQPW, SemAppl* | *request-Password* | – |
| task_Appl | 8 | when SemPW = 1. runs the application program. | SemAppl | *MsgQAppl.* | – | – |

Step *2:* Execute codes for a boot up task on reset resetTask. Let us code in a similar way as the codes in Example 11.1 for Firsttask. The codes begin to execute from the *main* and the main creates and initiates this task and starts the SmartOS. There it is the resetTask, which executes first.

The steps taken by the task synchronization model are as follows:

1. **resetTask** is the task that executes like Firsttask in Example 11.1. It suspends permanently after the following: (a) It initiates the timer for the system ticks, which are reset at 1 ms. (b) It creates three tasks, task_ReadPort, task_PW and task_Appl of the system that are described below. (c) For a waiting task, task_ReadPort, it sends into a message queue MsgQStart, the request header string *requestHeader*. The latter specifies the bank-allotted PIN to the user. (d) It also sends another string *requestStart* a request for host PIN at the I/O Port Port_IO in order to identify the host. (e) It posts a semaphore flag, SigReset, and suspends itself so that system control does not return to it till another reset.