Table 9.11 VxWorks Interprocess Communication Functions

Function	*Description*
semBCreate ( )	Creates a binary semaphore[1]
semMcreate ( )	Creates a mutex semaphore[1]
semCCreate	Creates a counting semaphore[1]
semDelete ( )	Deletes a semaphore
semTake ( )	Takes a semaphore
semGive ( )	Releases a semaphore
semFlush ( )	Resumes all waiting blocked tasks
msgQCreate ( )	Allocates and initializes a queue for the messages
msqQDelete ( )	Eliminates the message queue by freeing the memory
msgQSend ( )	Sends into a queue
msgQReceive ( )	Receives a message into the queue[2]
pipeDevCreate ( )	Creates a pipe device[3]
select ( )	A task waits for several kinds of messages, from pipes, for sockets and serial IOs[4]

[1] We specify an option SEM_Q_PRIORITY for the order in which the semaphore should be taken if there are a number of waiting tasks for same semaphore. Specify SEM_Q_FIFO for defining to take the semaphore in FIFO mode. The task waiting since longest gets that first.

[2] The calling task blocks if no message is available, else the message is read by the task. According to the option parameter, insertions into a queue can be an ordered one with priority as ordering parameter or for a FIFO based read.

[3] Statement STATUS = pipeDevCreate ('/pipe/name', max_msgs, max_length) will create a named pipe with maximum number of max_msgs messages in maximum pipe length max_length bytes.

[4] A task blocks if the message is not available at queue or socket and pipe and buffer are empty, when the task attempts to read the queue pipe, socket and IO buffer.

1. *Creating a binary semaphore for the IPCs.* The function 'SEM_ID semBCreate (*options, initialState*)' creates an ECB pointed by the SEM_ID. One of the two options mentioned next must pass on calling the function.

Passing parameters: (i) *Option(s):* One option, which can be selected is SEM_Q_PRIORITY. (ii) The other is SEM_Q_FIFO. Let us assume that at an instant, several tasks are in the blocked state and are waiting (pending) for a binary semaphore for its posting. A waiting task can take the semaphore in one of the two ways: (a) a task higher in priority than the other waiting ones takes the semaphore first and this becomes possible by SEM_Q_PRIORITY option, or (b) a task that was first blocked and reached the waiting state takes the semaphore first among the waiting ones and this becomes possible by the SEM_Q_FIFO option. The initial state of the binary semaphore passes by argument *initialState*. It is SEM_EMPTY when using the binary semaphore as an event-signalling flag. For the *initialState*, two options can be chosen: SEM_FULL in which the created semaphore's initial state is initialized as not available and already *taken;* and SEM_EMPTY, in which the created semaphore's initial state is initialized as available *not taken.* (Recall the use of semaphores SemKey and SemFlag in MUCOS.) There is mutex semaphore provision is different than the SemKey of MUCOS. In MUCOS, SemKey and SemFlag differ only initial values defined for them, the rest of the operations are identical. The semaphore *initialState* option defines the initial state when it was created.

Returning parameter: The function semBCreate ( ) returns a pointer, *SEM_ID. It returns NULL in case of an error for the ECB allocated to the *binary semaphore*, if none is available.

Example 9.21 explains the use of semBCreate. Let us assume that ISR_CharIntr is service routine on an interrupt when a byte becomes available at a port *A* and a task reads that byte after waiting for the semaphore availability from ISR_CharIntr.

Example 9.21

```
1. /* Include the VxWorks header file as well as semaphore functions from a library. */
# include "vxWorks.h"
# include "semLib.h"
# include "taskLib.h"
2. /* Task parameters declarations */
.
.
3. /* Declare a binary semaphore to be used as flag. */
SEM_ID semBCharIntrFlagID;
4. /* Create the binary semaphore and pass the options chosen selected to it. */
semBCharIntrFlagID = semBCreate (SEM_Q_PRIORITY, SEM_EMPTY); /* Higher priority waiting tasks
can take it first. Its initial state is not available. */
.
.
5. /* ISR creation codes */
6. /* Codes for ISR_CharIntr, for example, for puttig the port byte into a buffer */
.
.
7. /* At the end, make the binary semaphore SEM_FULL from SEM_EMPTY using semGive ( )*/
semGive (semBCharIntrFlagID); /* Section 9.34 explains SemGive */
 /* Other remaining codes for the ISR. */
.
.
 8. /* End of ISR_CharIntr Codes */
```

For using binary semephore, the following points are taken care.

(a) Declare initial value as SEM_EMPTY (not available)

(b) Use semTake ( ) in a task, which makes SEM_FULL to signal an event to another task

(c) Use semGive ( ) in waiting task for the event Example 9.21 showed the codes when using semBCreate ( ).

2. *Waiting for an IPC for binary or other type of semaphore release or check for availability of an IPC after release.* The function 'STATUS semTake (*semId, timeOut*)' is for letting a task wait till the release event of posting the binary or other type of semaphore. Wait till either *SemId* is posted (given) by a task or till time out occurs, whichever happens first. An exemplary use is semTake (*semBCharIntrFlagID*, WAIT_FOREVER). WAIT_FOREVER means timeout = –1 and the period is thus infinity. semTake is like OSSemPend function of MUCOS. (In MUCOS, time out = 0 means wait for ever.)

Passing parameters: (i) *semID* is the semaphore for which a suspended task waits; (ii) *timeOut* is the timeout period. One option that can be selected is WAIT_FOREVER if wait must be done for the posting of *semID*. The other option is NO_WAIT. Recall OSSemAccept function in MUCOS. Whenever this task is scheduled by the scheduler and this function is called, take the semaphore identified by *SemID* if set as available SEM_FULL. Third option, wait for timeout interval.

Returning parameter: The function semTake ( ) returns STATUS. It returns STATUS = OK in the case of success in taking the semID, else returns ERROR in the case of an error. After the semTake ( ) function unblocks a task, it again becomes available (empty or not taken).

3. *Sending an IPC after a binary or mutex or counting semaphore release (posting).* The function 'STATUS semGive (semId)' is for letting a task post (release) the binary or other type of semaphore. After this, a waiting-task can unblock. Which task unblocks depends on the option defined while creating the semaphore posted. Unblocking can be as per option SEM_Q_FIFO or as per SEM_Q_PRIORITY.

Passing parameter: *semID,* for which there is a wait by this or another task.

Returning parameter: The function semGive ( ) returns STATUS. It returns STATUS = OK in case of success in taking the semID, else returns 'ERROR' in case of an error that *semID* is invalid.

4. *Taking the semaphore multiple times till unavailable before the next posting.* Function STATUS semFlush (*semFlagID*) is for flushing. It will take the semaphore multiple times till unavailable before the next posting. It lets any waiting task not wait any further. It unblocks not only the calling but also all the other tasks waiting for taking the semaphore, *semFlagID.*

Parameter passing: The *semFlagID* passes as SEM_ID pointer at ECB that associates with the semaphore.

Returning parameter: The function semFlush ( ) returns the STATUS and makes the semaphore state from SEM_FULL (available) to SEM_EMPTY meaning unavailable. It returns STATUS = OK in case of the success in flushing of the semaphore and making semFlagID state SEM_EMPTY. Else, it returns 'ERROR' in case of an error that on time out the semaphore is unavailable or the semaphore identity is invalid. The semFlush ( ) function unblocks all the waiting tasks waiting for this *semFlagID* (semFlagID state = SEM_FULL earlier).

5. *Creating a mutex semaphore for the IPCs.* Mutex semaphore is needed when there is a critical section, which shares a data structure or uses a resource shared with the other tasks e.g., the bytes in a buffer between a sending task and a receiving task or using the flash memory for write operation or writing to a dislay device). There may also be sharing of the hardware devices or files between two tasks.

(a) An exemplary use in which we are using binary semaphore for mutual exclusion is as follows:
(i) SEM_ID semMKeyID;
(ii) semMKeyID = semMCreate (SEM_Q_PRIORITY, SEM_FULL).

(b) The function 'SEM_ID semMCreate' (*options*) is for creating an ECB pointed by the SEM_ID. The uses of semTake and semGive functions are as explained earlier. Let us assume that when entering a critical region in a task, semTake (semMReadPortAKey) executes and on leaving the critical region, semGive (semMReadPortAKey) for using the mutex semaphore, semMReadPortAKey. Here, we prevent the priority inversion situation (Section 7.8.5) by choosing an option. Another option is for selecting either SEM_Q_FIFO or SEM_Q_PRIORITY. However, when selecting *SEM_INVERSION_SAFE* we must select the option SEM_Q_PRIORITY. (Reason for using | sign in place of & in case of passing multiple options by a single argument was given before in Section 9.3.2.)

(c) Another example of using three options in semMCreate function argument is as follows. Here, the option prevents the priority inversion situation as well as protects the task from deletion by any other task until the semaphore is made empty (not taken) at the end of the critical region of a task. Three options are selected as follows.

```
SEM_ID semMReadPortAKey;
semMReadPortAKey = semMCreate (SEM_Q_PRIORITY | SEM_INVERSION_SAFE
| SEM_DELETE_SAFE);
```

Passing parameters: (i) The use of option between SEM_Q_PRIORITY and SEM_Q_FIFO is identical to the binary semaphore, which was described earlier. (ii) The use of SEM_INVERSION_SAFE makes the critical section using the mutex safe from priority inversion situation (Section 7.8.5). It means that the created semaphore initial state is initialized as SEM_FULL (available). (iii) Recall the use SEM_DELETE_SAFE. It protects deletion of this task when in the critical region.

Returning parameter: The function semMCreate ( ) returns a pointer, *SEM_ID for the ECB allocated to the *mutex semaphore.* It returns NULL in case of an error if none available.

Example 9.22 shows the codes using semMCreate ( ), semGive ( ) and semTake ( ).

Example 9.22

```
1. /* Include the VxWorks header file as well as the task and semaphore functions from a library. */
# include "vxWorks.h"
# include "semLib.h"
# include "taskLib.h"
/* Declare a semaphore key to be used as mutex. */
2. SEM_ID semMReadPortAKey;
/* Create the mutex and pass the options chosen selected to it. */

semMReadPortAKey = semMCreate (SEM_Q_PRIORITY | SEM_INVERSION_SAFE); /* This makes
the mutex semaphore full and available like SemKey set as 1 in MUCOS. */
 semGive (semMReadPortAKey);
3. /* Task1 creation codes */
.
.
4. /* Initial Codes for the Task 1 */
.
.
/* Task 1 Initial Codes ends */
5. /* Task while loop codes */
.
.
 6. semTake (semMReadPortAKey, WAIT_FOREVER); /* Critical Section (shared resource or data
section starts) */
.
. .
7. semGive (semMReadPortAKey); /* Critical Section (shared resource or data section ends) */
8. /* Remaining task 1 codes */
9. /* Task2 creation codes */
.
.
10. /* Initial Codes for the Task 2*/
.
.
/* Task 2 Initial Codes end */

11. /* Task 2 while loop codes */
.
.
 12. semTake (semMReadPortAKey, WAIT_FOREVER); /* Critical Section
(shared resource or data section starts) */
.
.
13. semGive (semMReadPortAKey); /* Critical Section (shared resource or data
section ends) */
14. /* Remaining task 2 codes */
/*****************************************************************/
```

For using mutex, the following points are taken care.

(a) The critical section that uses mutex for the resources protection should not be unnecessarily long and should be as short as possible.

(b) Declare initial value as SEM_EMPTY (available) and use options for SEM_DELETE_SAFE if some task uses taskDelete ( ) function when using semBCreate.

(c) Use option SEM_INVERSION_SAFE if some priority inversion situation is likely to arise and affect the system functioning

(d) Use semTake function in the same task at the beginning and semGive at the end of a critical region in which there are shared resources. semTake can be used recursively but the total number of times a semTake executes should be the same as the number of times semGive executes.

(e) Do not use semGive ( ) for the mutex posting outside the critical region.

(f) Do not use semFlush ( ) (its use is illegal when using the mutex semaphore).

6. *Creating a counting semaphore for the IPCs.* VxWorks counting semaphore (Section 7.7.5) is similar to the POSIX semaphore (Section 7.8.3). It increments on posting (giving) and decrements on taking (on wait-over) the semaphore. Posting this semaphore up to 256 times is permitted before it is taken. The status becomes equal to the initial value of counting semaphore only when the number of times semaphore-given equals to the number of times it is taken. The counting semaphore helps in bounded buffer problem, ring-buffer problem and consumer–producer problem (Section 7.8.3). We have seen this in Example 9.18. If initial count = 0, then a task waiting for the semaphore blocks.

The function SEM_ID semCCreate (*options*, unsigned byte *initialCount*) is for creating an ' ECB pointed by the SEM_ID. One of the two options must be passed on calling a function. An exemplary use is as follows: 'semCCharIntrFlagID = semCCreate (SEM_Q_PRIORITY, SEM_EMPTY);'.

(a) SEM_ID semCID;

(b) SEM_ID = semCCreate (SEM_Q_PRIORITY, 0); /* To initial count = 0. */

Passing parameters: (i) One option that can be selected is SEM_Q_PRIORITY and the other is SEM_Q_FIFO. For the initial state, two options can be chosen: either initialCount should pass as 0 or it should be a fixed value. It depends on whether the semaphore is to be used for decrementing count for the tasks that are already blocked or (b) for incrementing counting.

Returning parameter: The function semCCreate ( ) returns a pointer, *SEM_ID. It returns NULL in case of an error for the ECB allocated to the *counting semaphore*. Null if none is available.

Recall MUCOS of Example 9.18, Steps 11 to 21. Example 9.23 shows how to use VxWorks semCCreate function. It also shows the use of the other VxWorks functions for task spawning and semaphores.

Example 9.23

```
1. /* Same as Steps 1 and 2 of Examples 9.21 and 9.22. */
2. /* Declare and Create Semaphores function, its identifying variables. */
/* Declare SemFlag1ID as the argument that passes to the task whenever called. Declare SemMKeyID and
SemCCountID as the mutex and counting semaphores. */
SEM_ID SemFlag1ID, SemMKeyID, SemCCountID;
3. /* Create Semaphore flag and declare unblocking of the tasks priority wise. Declare initially semaphore
flag unavailability. */
SemFlag1ID = semBCreate (SEM_Q_PRIORITY, SEM_EMPTY);
4. /* Create Semaphore mutex and declare unblocking of the tasks priority wise. Initially semaphore
mutex is available by default. */
SemMKeyID = semMCreate (SEM_Q_PRIORITY); /* SEM_Q_PRIORITY | SEM_DELETE_SAFE
```

two options can also be used. However that prolongs the execution time. We are not using safe option as taskDelete ( ) function is not used. Initially semaphore (mutex) is available by default. */
5. /* Create Semaphore for counting and declare unblocking of the tasks priority-wise.
unsigned byte *initialCount* = 0;
SemCCountID = **semCCreate (SEM_Q_FIFO,** *initialCount*);
unsigned short *COUNT_LIMIT* = 80; /* Declare limiting Count = 80 */
6. /* Declare and Create Semaphores task function, its variables and parameters. */
void Task_ReadPortA (SEM_ID *SemFlag1ID*);
int *readTaskID* = ERROR; /* Let initial ID till spawned be none */
int *Task_ReadPortAPriority* = 105; /* Let priority be 105 */
int *Task_ReadPortAOptions* = 0; /* Let there be no option. It waits for the SemFlag1ID from the ISR (Example 3.21). */
int *Task_ReadPortAStackSize* = 4096; /* Let stack size be 4 kB memory */
4. /* Create and initiate a task for reading at Port A. Task name starts with 't'. The task calling-function is Task_ReadPortA */
readTaskID = **taskSpawn** ("tTask_ReadPortA", *Task_ReadPortAPriority*, *Task_ReadPortAOptions*,
Task_ReadPortAStackSize, void (* Task_ReadPortA) (SEM_ID *SemFlag1ID*), SemMKeyID, SemCCountID, *&initialCount*, *COUNT_LIMIT*, 0, 0, 0, 0, 0, 0); /* Pass SemFlag1ID as the argument of task function and pass other arguments SemMKeyID and SemCCountID as arg0 and arg1. Remaining arguments are 0s. */

.
.
/* Other Codes */
.
.
5. /* The codes for the Task_ReadPortA redefined to use the key, flag and counter*/
static void Task_ReadPortA (SEM_ID *SemFlag1ID*) {
6. /* Initial assignments of the variables and pre-infinite loop statements that execute only once */
; /* Declare the buffer-size for the characters countLimit = 80 */
intCount = 0;
.
.
7. *while (1)* { /* Start an infinite while-loop. We can also use FOREVER in place of while (1). */
8. / *Wait for SemFlag1ID state change to SEM_FULL by semGive function of character availability check task */
***semTake* (SemFlag1ID, WAIT_FOREVER)**;
9. /* Take the key so that another task, *port decipher* does not unblock. That task needs SemMKeyID to unblock and run */
***semTake* (*SemMKeyID*, WAIT_FOREVER);** /* SemMKeyID is now not available and the critical region starts */
10. if (*Count* > = *COUNT_LIMIT*) {
.
.
};/* End of Codes for the action on reaching the limit of putting the characters into the buffer */

11. /* Codes for reading from Port A and storing a character at a queue or buffer*/
.
.
12. **semGive (SemCCountID);** *Count* ++; /*Let counting semaphore value increase because one character has been put into the buffer holding the character stream. initialCount incremented because of the need to compare later with the COUNT_LIMIT*/
13. **semGive (SemMKeyID);** /* Critical region ends. Release the mutex SemMKeyID to let next cycle of this loop start */
14. } /* End of while loop*/
15. } / * **End of the Task_ReadPortA function** */

A point to be noted is that there is no provision for setting the limit up to which a counting semaphore can be given (posted). It is fixed at 65,535 in MUCOS. Therefore, we have to use COUNT_LIMIT variable, and compare with the Count variable, which increments after each function 'semGive (SemCCountID)' call.

/**/

7. *Using POSIX semaphores.* POSIX semaphore functions can also be used for the VxWorks counting semaphores. The function 'semPxLibInit ( ) initializes' the VxWorks library to permit use of these. The functions are sem_open ( ), sem_close ( ) and sem_unlink ( ) to initialize, close and remove a named semaphore, respectively.

The functions sem_post ( ) and sem_wait ( ) unlock and lock a semaphore. The actions of these two are similar to semGive and semTake in the VxWorks counting semaphore or OSSemPost and OSSemPend functions of MUCOS semaphores. The function sem_getvalue ( ) retrieves the value of a POSIX semaphore. The actions of the function is similar to OSSemQuery in MUCOS.

POSIX semaphore functions sem_init ( ) and sem_destroy ( ) initialize and destroy. Destroy means de-allocate associated memory with the semaphore ECB. (This effect is not the same as first closing a semaphore and then unlinking it by sem_close and sem_unlink). Remember that no deletion safety like VxWorks mutex is available before using these functions. VxWorks semaphores have the additional following features. (i) Options of protection from priority inversion and task deletion. (ii) Single task may take a semaphore multiple times and recursively. (iii) Mutually exclusive ownership can be defined. (iv) Two options, FIFO and task priority for semaphores wait by multiple tasks.

sem_trywait ( ) is to try to lock a semaphore if not available and locked by other task.

8. *Creating a message queue for the message IPCs.* The function 'MSG_Q_ID msgQCreate (int maxNumMsg, int maxMsgLength, int q*Options*)' is used for creating an ECB pointed by the MSG_Q_ID. One of the two options mentioned next must pass on calling the function. The memory allocation to the buffer, which holds the bytes for messages is according to *maxNumMsg* and *maxMsgLength* (parameters for maximum number and length).

A point to be noted is that the message pointer passed into an array of message pointers in MUCOS (Example 9.20).

In a message queue, the maximum number of messages = $2^{31} - 1$ and the maximum number of bytes in each message is $2^{31} - 1$ bytes.

Passing parameters: (i) To the function, *maxNumMsg passes* the maximum number of messages that can be sent to the queue. (ii) *maxMsgLength* passes the maximum number of bytes permitted to be sent as a message. (iii) One option that can be selected is MSG_PRI_NORMAL, when the message is sent into the queue for receiving as a FIFO. The first message sent is then read first. The other option is MSG_PRI_URGENT. When the message is sent into the queue with this option, the message is received as LIFO. Urgent messages

like error logins are sent with this option selected. The last message sent is then read first. (iv) Another option that can be selected is MSG_Q_PRIORITY. The other is MSG_Q_FIFO. Let us assume that at an instant, several tasks are in the blocked state and are waiting (pending) for a message from the same queue for its posting (sending). A waiting task can take from the queue in one of two ways. (a) A task higher in priority than the other waiting ones, takes the message from the queue first. This becomes possible by MSG_Q_PRIORITY option. (b) A task, which first blocked and reached the waiting state, takes the message from queue first among the waiting ones. This is possible by MSG_Q_FIFO option.

Returning parameter: The function msgQCreate ( ) returns a pointer *MSG_Q_ID. It returns NULL in case of an error for the ECB allocated to the *message queue*. Null if none is available or on error.

Consider the *tasks* in a hand-held device. The *tasks* post the messages into a queue using msgQsend ( ). A buffer for writing into the flash memory in a task receives the bytes using MsgQReceive ( ).

9. *Sending an IPC after a message is sent into the queue.* The function 'STATUS msgQSend (*msgQId*, *&buffer*, *numBytes*, *timeOut*, *msgPriority*)' is used for letting a task send into the queue.

Passing parameters: (i) The queue identifies by msgQ*Id*. (ii) Message posts to an addressed *buffer*. The number of bytes sent into the buffer = numBytes. (iii) The timeOut is the period till posting of the message is awaited in case the queue is full. (iv) *msgPriority* is specified as MSG_PRI_NORMAL or MSG_PRI_URGENT, depending upon whether the message is inserted later on to retrieve in the FIFO or LIFO mode, respectively.

Returning parameter: The function msgQSend ( ) returns STATUS. It returns STATUS = OK in case of success in taking the msgQId, else returns 'ERROR' in case of an error that *msgQId* is invalid.

Example 9.24 shows how to use the queue functions for create and send.

Example 9.24

```
1.  /* Include the header files in Example 9.22 Step 1 as well as queue functions from a library. */
2.  # include "msgQLib.h"
3.   /* Declare message queue identity and message data type or structure. */
4.  MSG_Q_ID portAInputID;
5.  unsigned byte portAdata;
6.  void * message; /* Pointer for the message buffer */
7.   /* Create the message queue identity and pass the parameters and options chosen selected to it and let
     the maximum number of messages be 80 and message be of 1 byte each.
     Let us assume that Task_ReadPortA reads a byte from port A and sends it to another task that receives
     the messages from a queue after waiting for the queue message availability. */
8.  PortAInputID = msgQCreate (80,1, MSG_Q_FIFO | MSG_PRI_NORMAL)
9.  /* Task Creation Codes as in Example 9.23. */
10. .
11. .
12. /* Start Codes for Task_ReadPortA. */
13. void Task_ReadPortA {
14. .
15. while ( ) {
16. semTake (SemFlag1ID, WAIT_FOREVER);
17. /* Take the key to not let port-decipher task unblock and run. Therefore, that task also needs
     SemMKeyID for running. */
18. semTake (SemMKeyID, WAIT_FOREVER); /* SemMKeyID is now not available and
     the critical region starts */
```

```
19.  {
20.  .
21.  /* At the end, the send the byte, which is read at Port A. It is sent as a message to queue,
     portAInputID.
22.  *message = portAData;
23.  msgQSend (portAInputID, &message, 1, NO_WAIT, MSG_PRI_NORMAL);
24.  /* Other remaining codes for the task. */
25.  .
26.  .
27.  }
28.  }/* End of Task_ReadPortA Codes */
```

10. *Waiting in a queue for availability of message.* The function 'int msgQReceive (*msgQId*, &buffer, maxBytes, timeOut)' is used *for letting a task wait till sending (posting) of a message*. Wait till either *msgQSend* function sends the message in a task or till a time out occurs, whichever happens first.

Passing parameters: (i) Whenever the scheduler schedules this task and this function is called, there is a wait for a message pointed by *msgQId*. (ii) *buffer* is the address of the buffer. (iii) maxBytes is the maximum number of bytes acceptable per message by msgQReceive function. (iv) timeOut is the time out period. One option that can be selected is WAIT_FOREVER if wait must be done for sending the message by msgQSend function. The other option is NO_WAIT. Recall OSQFlush function in MUCOS. NO_Wait option is simply used for checking the availability of a message. Message is received from buffer if available else task runs other succeeding codes. In case of error message, for example, there is no waiting especially for the message. Only check for error message is needed.

Returning parameter: The function msgQReceive ( ) returns an integer for the number of bytes retrieved from the buffer address.

Example 9.25 shows how to use msgQReceive function.

Example 9.25

```
1. to 9. /* Codes as per Steps 1 to 9 in Example 9.24 */
10. /* The codes for the Task_MessagePortA */
static void Task_MessagePortA (void *taskPointer) {
11. /* Initial assignments of the variables and pre-infinite loop statements that execute once only*/
 int maxBytes = 80;
12. while (1) { /* Start an infinite while-loop. */
13. /* Wait for a Queue Message sending or availability. */
      msgQReceive (msgQId, message, maxBytes, WAIT_FOREVER); /* WAIT_FOREVER means
      timeout = -1 and the period is thus infinity. */
14. /* Other remaining Codes */
.
.
15. }; /* End of while loop*/
16. } /* End of the Task_MessagePortA function */
/*************************************************************************/
```

11. *Using POSIX queues.* Important points in using the POSIX queues are as followings.
    (a) The function mqPxLibInit ( ) initializes the VxWorks library to permit use of the POSIX Queues.
    (b) The functions mq_open ( ), mq_close ( ) and mq_unlink ( ) initialize, close and remove a named queue.
    (c) The function mq_setattr ( ) sets the attribute of a POSIX queue.
    (d) The functions mq_send ( ) and mq_receive ( ) unlock and lock a queue.
    (e) The function 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 mq_notify). This provision is extremely useful for a server task. A server task receives the notification from a client task through a signal-handler function (like an ISR).
    (f) The function mq_getattr ( ) retrieves the attribute of a POSIX queue.
    (g) The POSIX queue function mq_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.

VxWorks queues have the additional following features. (i) Time out option can be used. (ii) Two options, FIFO and task priority for queues wait by multiple tasks. POSIX queues have the additional following feature: task notification in case a single waiting task is available and there can be 32 message priority levels in place of one priority level URGENT in VxWorks.

12. *Creating a pipe device for read-write in IPCs.* When a task creates, a taskID allocates (Section 9.3.1). When using the task-related functions, the number facilitates task identity. Similarly, a pipe (Section 7.14) or socket (Section 7.15) or file (Section 8.6.2) when creates, a file descriptor data structure is created and a number, for example, fd is assigned to identify the device created. The number is assigned after examining a set of the numbers already allocated. When using the device-related functions, the number facilitates the device identity. The device function examples are open or read or write or get attribute or set or attribute close (Section 8.6).

A pipe in VxWorks is a FIFO queue, which is managed not by queue IPC functions but by the device-driver functions. VxWorks has management functions for a pipe-driver (like a device driver) *pipedrv*. This is analogous to the named pipe driver in Unix. Pipes also implement the unidirectional link between a set of tasks.

Function pipeDevCreate ('/pipe/*pipeName*', *maxMsgs*, *maxMsgBytes*) creates a pipe device named *pipeName* for maximum *maxMsgs* messages. Each message can be of maximum size *maxMsgBytes*. It enters into a list of devices on creation. devs ( ) function retrieves the list of devices with the device number allotted to each device including pipe devices.

Consider an example for creating a pipe named as *pipeUserInfo*. Assume that it can have a maximum of four messages: user name, password, telephone number and e-mail ID. Each of these can be of a maximum size of 32 bytes only. A global variable fd is an integer number for a file descriptor that identifies a device among a number of devices at the IO system. The device can be a file or pipe or socket or other device. Example 9.26 explains the codes for creating, writing and reading.

Example 9.26

```
1. # include "fioLib.h" /* Include the IO library functions. */
pipeDrv ( ); /* Install a pipe driver. */
2. /* Declare file descriptor. */
int fd;
 3. /* Mode refers to the permission in an NFS (Network File Server). Mode is reset as 0 for unrestricted
permission. */
int mode;
4. /* Create pipe named as pipeUserInfo for 4 messages, each 32 bytes maximum. */
```

pipeDevCreate ("/pipe/*pipeUserInfo*", 4, 32); mode = 0x0;
Messages can be written into a pipe by the function by first opening a pipe, device and then writing into that. The function for opening is open ('/pipe/*pipeUserInfo*', *rdwrFlag*, mode). We define flag = O_RDWR, which permits both read and write. Flag O_RDONLY permits the read only option and flag O_WRONLY permits the write only option. Remember that after opening a pipe, when we finish using it, we must use the function 'STATUS close ( )'. Writing to a pipe is analogous to writing on an IO device. To write, the coding is as follows.

```
5. /* Open read-write device using, a pipe named as pipeUserInfo with mode = 0 for unrestricted permission.
*/
fd = open ("/pipe/pipeUserInfo", O_RDWR, 0); /*A file descriptor is used for a file or pipe or socket or
serial device or other type of device. */
6. /* Write a message, info of lBytes. */
char [ ] info; Let the message be a string of characters. */
int lBytes;
lBytes =12;
write (fd, info, lBytes);
```

The message can be read from an open pipe by the function 'int read (fd, &buffer, *lBytes*)'. Reading from a pipe is analogous to reading from an IO device as per the file descriptor. To read, the coding is as follows.

```
7. int numRead; /* An integer to indicate the number of bytes successfully read.* /
lBytes =12; /* Let the Message bytes to read = 12*/
 numRead = read (fd, info, lBytes);
```

13. *Finding the set of opened devices at an instance from the number of devices in the system.* Recapitulate event functions in Section 8.4. Assume that all bits are cleared at the time of creation of an event flag group. A register saves the bits, which set on the occurrences of the events from the number of sources. Each event sets one bit. A task can wait for setting any or all events in the group.

Similarly, there is FD_SET. FD_SET sets a file descriptor function. Each device in a system having a number of devices set a bit in the *fdSet*. (A file descriptor is used for a pipe or socket or serial device or other type of device. Let a file descriptor fd = n; there is an array of bits in which the nth bit corresponds to fd = n.)

Now function, FD_SET (n, *&fdSet*) defines the data structure *fdSet*, when executed will make nth bit = set. FD_SET (m, *&fdSet*) makes the mth bit = set. FD_CLR (n, *&fdSet*) will make nth bit = clear. FD_ZERO (*&fdSet*) makes all bits of array = 0. FD_ISSET (n, *&fdSet*) returns true if nth bit in the array is set and false if reset at *fdSet*.

Now, let us examine how a task selects and finds the number of active devices at an instance. Task finds whether a pipe, *pipeUserInfo* is active or a *pipeResponse* is active. The function to select is 'int select (*numBitWidth*, *pointerReadFds*, *pointerWriteFds*, *pointerExceptFds*, *pointerTimeOut*). The arguments passed are the following: *numBitWidth* = number of bits to examine in the array of bits at two pointers, *pointerReadFds* and *pointerWritedFds*. Examination is as per the value at a structure that stores NULL if wait forever or a value for time out. Timeout is the number of system clock interrupts up to which the wait is done. The function select ( ) blocks till at least *one device* in the array of devices is ready or till time out, whichever happens first. Select clears all the bits that correspond to the devices that are not ready and returns the number of active devices. It returns *ERROR* on an error.

Summary

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

- The basic functions in the RTOSes and types of RTOSes.
- It is a necessity to use a well-tested and debugged RTOS in a sophisticated multitasking embedded system. MUCOS and VxWorks are the two important RTOSs.
- Code elegancy is one of the best in MUCOS and the provision of powerful functionalities is one of the best in VxWorks.
- MUCOS task creating and deleting, suspending and resuming functions are used for the task controlling and scheduling functions.
- There are functions for initiating the system timer in MUCOS. Starting a multitasking system by a first task and later suspending it forever is shown as a technique in programming for a multitasking system.
- MUCOS handles and schedules the tasks and ISRs and handles pre-emptive scheduling.
- There are delay and delay resume functions in MUCOS. These are shown to be useful for letting a low priority task run.
- MUCOS has the IPC functions for the event flag group, semaphore, mailbox and queue. The simplicity feature of MUCOS is that the same semaphore functions are used for binary semaphore, for event-signalling flag, for resource key and counting.
- MUCOS has mailbox functions and a simple feature that a mailbox has one message pointer per mailbox. There can be any number of messages or bytes, provided the same pointer accesses them.
- MUCOS has queue functions. A queue receives from a sender task an array of message pointers. Message pointers' insertion can be such that later on it can retrieve in the FIFO method as well as in the LIFO method from a queue. It depends on whether the post was used or post-front function was used, respectively. This helps in taking notice of a high-priority message at the queue.
- VxWorks is a popular broadly focused RTOS because of its powerful development tools, support to advanced processor architectures and device software optimization.
- VxWorks supports the multiple file systems, systems that enable advanced multimedia functionality and multitasking environment using VxWorks scheduler, POSIX scheduler or in-house developed scheduler.
- VxWorks supports ability to run two concurrent OSes on a single processing layer (e.g., VxWorks and Windows or VxWorks and embedded Linux).
- Instead of one create function, VxWorks has three functions: task create, task activate and task spawn (create and activate).
- VxWorks also provides for system timer functions, system auxiliary clock functions, watch dog timer functions, delay and delay resume functions.
- VxWorks handles and schedules the functions for the tasks and ISRs differently. It allocates highest priorities for the ISRs over the tasks and provides nested ISRs, and thus a common stack of the ISRs.
- VxWorks has an IPC called *signal*. It is used for exception handling or handling interrupts from the tasks. VxWorks has signal-servicing routines. A signal-servicing routine is a C function. It executes on occurrence of an interrupt or exception. A connect function connects the function with the interrupt vectors.
- Exceptions are the software interrupts. A signal setting is equivalent to a flag setting in case of hardware interrupts.
- VxWorks provides for pre-emptive scheduling as well as round robin time-sliced scheduling of tasks assigned equal priority.
- VxWorks provides for two ways in which a pending task among the pending tasks can unblock. One is as per the task priority and another is as a FIFO when accepting (taking) an IPC.
- VxWorks has three different semaphore functions for use as IPC for the event-signalling flag, resource key and counting semaphore. VxWorks also supports POSIX semaphores. VxWorks, instead of queuing the message pointers only, provides for queuing of the messages. Queues can be used as LIFO as in MUCOS. VxWorks also supports use of pipes and POSIX queues. VxWorks pipes are the FIFO queue that can be opened and closed like a file device. Pipes are like virtual IO devices that store the messages as FIFO.

Keywords and their Definitions

Counting semaphore : It is a semaphore that increments when an IPC is given by a task or a section of the task. It decrements when a waiting task unblocks and starts running.

Event-signalling flag : A flag, which sets on occurrence of an event and resets on response to the event. A binary semaphore or event flag group bit can be used as the *event-signalling* flag.

Exception handling : Executing a function on receiving a signal. Error is also handled by using an exception-handling function.

FD set : The file descriptors of all devices exist into a data structure FD set. The bits corresponding to active devices are set and inactive devices are cleared.

File descriptor : A pipe or socket or file (Section 8.6.2) when creates, a file descriptor a data structure is created and a number, for example, fd is assigned to identify the device created.

File device : Memory block(s) in which the file read, file write, file open and file close functions operate as in case of file on a disk.

Mailbox : An IPC in the event control block into which a task or ISR posts a message pointer, which is retrieved by another task waiting for that.

Message queue : An IPC in the event control block into which a task posts the messages at the tail pointer or urgent messages at the front pointer, which are retrieved by another task waiting for that.

Pipe : A device from which one task gets the messages and the other task puts the messages. VxWorks *pipe* is a FIFO queue in which the IO device functions operate. Putting and getting messages from a pipe is like the one from a file.

POSIX queues : IPC queue functions as per POSIX standard functions.

POSIX semaphores : Semaphore functions as per the IEEE POSIX standard functions.

Resource key : A semaphore that resets on the start of execution of a critical region code and sets on finishing these.

Signal : Flag-like intimation to RTOS for development of certain situations during a run that need urgent attention by executing an ISR function-like signal handler.

Sophisticated multitasking embedded system : A system that has multitasking needs with multiple features and in which the tasks have deadlines that must be adhered to.

System timer : A system clock that can be set to interrupt at preset intervals. The time is updated regularly and the system interrupts regularly. RTOS also gets control of CPU to examine if any pre-emption or rescheduling is needed. Task priority provides priority for system timer functions, delay functions and delay resume functions.

Task delay : Let a task wait for a minimum time defined by the number of system ticks passed as an argument to the delay function.

Task spawning : Task creation and activation.

MUCOS : An RTOS µC/OS-II from Micrium of Jean J. Labrosse.

VxWorks : An RTOS from Wind River® Systems.

Task creation : Task is allotted a TCB and an identity. Creation also *initiates* and *schedules* on creation in MUCOS.

Task deletion : Task no longer has the TCB and is ignored till created again.

Task resumption : Task, which was delayed or suspended, can now be scheduled when the turn comes.

Task suspension : A task unable to run its codes further.

TCB : Task control block, which has the task parameters so that on task switching the parameters remain saved and when RTOS re-switches it back, the task can run from the point at which it left. Task is thus an independent process.

Well-tested and debugged RTOS : An RTOS, which is thoroughly tested and debugged in a number of situations.

Review Questions

1. What are the advantages of a well-tested and debugged broad-focussed RTOS, which is also well trusted and popular? (*Hint:* Embedded software has to be of the highest quality and there should be faster software development. Need for complex coding skills required in the development team for device drivers, memory and device managers, networking tasks, exception handling, test vectors, APIs and so on.)
2. How does a mailbox message differ from a queue message? Can you use message queue as a counting semaphore?
3. Explain ECB.

Practice Exercises

Note: Exercises 5 to 12 pertain to MicroC/OS-II and 14 to 23 to VxWorks.

4. Search the web (e.g., www.eet.com) and find the latest top RTOS products.
5. Draw five figures showing models for five examples 9.16 to 9.20 in Section 9.2 for event-flag semaphore, mutex, counting semaphore, mailbox and queue interprocess communication.
6. Draw the figures to show the models for interprocess communication at processes in digital camera, ACVM and orchestra playing robot examples in Sections 1.10.4, 1.10.2 and 1.10.7, respectively.
7. Classify and list the source files, which depend on the processor and those that are are processor-independent?
8. Design a table that gives MUCOS features.
9. MUCOS has one type of semaphore for using as resource key, as flag, as counting semaphore and mutex. What is the advantage of this simplicity?
10. How do you set the system clock using function void OSTimeSet (unsigned int counts)?
11. When do you use OS_ENTER_CRITICAL ( ) and OS_EXIT_CRITICAL ( )?
12. How do you set the priorities and parameters, OS_LOWEST_PRIO and OS_MAX_TASKS, for pre-emptive scheduling of the tasks?
13. A starting task is first created, which creates all the tasks needed, initiates the system clock and then that task is suspended. Why must this strategy be used?
14. VxWorks kernel includes both POSIX standard interfaces and VxWorks special interfaces. What are the advantages of special interfaces for the semaphores and queues?
15. How do you initiate round robin time-slice scheduling? Give 5 examples of the need for round robin scheduling.
16. How do you initiate pre-emptive scheduling and assign priorities to the tasks for scheduling? Give 10 examples of the need for pre-emptive scheduling.
17. How do you use signals and use function void sigHandler (int *sigNum*), signal (*sigNum, sigISR*) and intConnect [I_NUM_TO_IVEC (sigNum), sigISR, sigArg]? Give five examples of their uses.
18. How do you create a counting semaphore?
19. OS provides that all ISRs share a single stack. What are the limitations it imposes?
20. How do you create, remove, open, close, read, write and IO control a device using RTOS functions? Take an example of a pipe delivering an IO stream from a network device.
21. Explain the use of file descriptor for IO devices and files.
22. How do you let a lower priority task execute in a pre-emptive scheduler? Give four coding examples.
23. How do you spawn tasks? Why should you not delete a task unless memory constraint exists?
24. Write exemplary codes for using the POSIX functions for timer, semaphores and queues.

REAL-TIME OPERATING SYSTEM PROGRAMMING-II: WINDOWS CE, OSEK AND REAL-TIME LINUX FUNCTIONS

10

We have discussed the following important points relating to the RTOSes in the previous chapters.

- An RTOS has basic functions (services) of process (thread or task) and memory management, enables sharing of resources and data, enables use of timers and system clock, does time allocation and de-allocation to attain best utilization of the CPU time under the given timing constraints for the tasks, manages interprocess communication (IPC) (communication between the ISRs, tasks and OS functions) and IO subsystems, manages devices and device drivers and provides for real-time task-scheduling and interrupt-latency control. RTOS enables hard and soft real-time operations. RTOS provides a predictable timing behaviour of the system (in most cases in case) and a predictable task synchronization using the priorities allocation RTOS provisions for priorities inheritance.
- A programmer uses RTOS functions in application software and APIs. RTOS also enables asynchronous IOs. RTOS functions synchronize the concurrent

running of processes (tasks or threads), fast-level ISRs, slow-level interrupt service threads (ISTs).

3. **MUCOS and VxWorks are the two important RTOSes. MUCOS and VxWorks functions provide programming for the ISRs and tasks (processes).**
4. **Code elegancy and reliability is one of the best in MUCOS and provision of powerful functionalities is one of the best in VxWorks. VxWorks is a popular broadly focused RTOS because of its powerful development tools, support to advanced processor architectures and device software optimization. VxWorks supports the multiple file systems, systems that enable advanced multimedia functionality and multitasking environment using VxWorks scheduler, POSIX scheduler or in-house developed scheduler. VxWorks supports ability to run two concurrent OSes on a single processing layer.**

We will discuss the following popular RTOSes in this chapter.

1. *Windows CE for consumer electronic systems and devices*
2. *OSEK—a reliable RTOS for the automotive electronic system*
3. *Open source real time Linux*
4. *RTLinux*

10.1 WINDOWS CE

Windows CE (WCE) is an RTOS for handheld computers and mobile systems, developed by Microsoft. Microsoft designer perception for using the word CE is that CE stands for the properties that it is *compact, connectable, compatible, companion* and *efficient*. WCE can also be perceived as Windows for consumer electronics systems, however applications do not limit to consumer electronics systems.

WCE is nowadays one of the most popular OSes for the handheld systems.

An enhancement of WCE is Windows CE.NET. [Dot 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. A run-time environment converts the byte code instructions into the native machine and platform instructions. When different CPU embeds into the system, the different run-time environment is used. Therefore, bytes code can run on different platforms and be distributed. At run time, the .NET run-time verifies the executing native environment, data source and destination types, within range array indices and other functionalities. The code becomes robust.]

Windows CE.NET is used as a real-time operating system for handheld computers and mobile systems. WCE.NET is described in detail in Douglas Boling *Programming Microsoft WINDOWS CE.NET*", Microsoft, USA, 2003. Section 10.1.1 describes in brief the basic features and functions in WCE. Figure 10.1 shows the basic features.

10.1.1 Windows CE Features

WCE platform provides the following features.

1. Provides a Windows platform for the systems, which have resource constraints of power, memory, touch screen or display screen size and processing speeds. Windows platform enables a user to feel, look and interact with the system using GUIs in a manner similar to a PC running on Windows.
2. It is an open, scalable and small-footprint 32-bit OS.
3. Enables running of PocketPC applications such as Outlook, Explorer, Pocket Power-Point, Pocket-Word and Pocket-Excel for mailing, Internet, PPT and slide shows and office applications. PocketPC is a handheld PC based on WCE. Latest version CE 6.0 is for home as well as office systems and gives cellular networks connectivity. WCE using systems enable running of multimedia, voice user interfaces (VUIs), smartphone and game applications. (Voice user interfaces facilitate interaction and command inputs using stored voice or tunes, and voice-command inputs from user.)

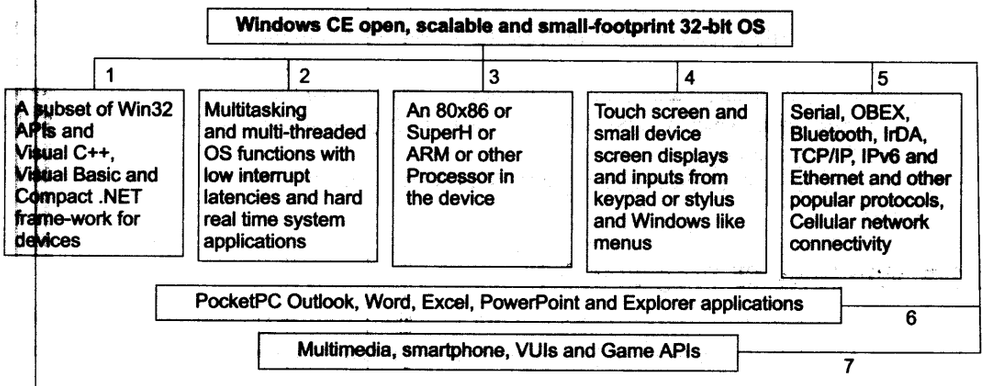


Fig. 10.1 Windows CE basic features

4. Processor of a WCE using system may be an 80x86 or SuperH or ARM or SH4 or MIPS. WCE system-performance fine tunes to the processor.
5. Functions as multitasking and multithreaded OS. Multitasking means that there are a number of processes which can run concurrently. Each process can have multiple threads and has at least one thread. A thread is a basic unit of computation using the resources which are provided by WCE. WCE provides support to 256 levels of thread-priorities. WCE also provides for adjustable time quantum for the threads. Threads having equal priorities are assigned a time slice (Section 8.10.2) each during the system run.
6. WCE has low interrupt latencies due to use of ISTs in addition to ISRs. The ISTs are put in priority queue of threads waiting for execution. (Sections 4.2.3 and 8.7.3). An IST is the slow-level interrupt service thread of a fast-level ISR. This gives WCE functionalities for hard real-time scheduling and interrupt latency control. WCE supports nested ISRs.

7. Enables use of a subset of Win32 APIs, Visual C++, Visual Basic and a .NET Compact Framework (in Windows CE.NET). Software can be created using Microsoft Visual Studio 2005. The code is developed in Visual C++. Software for mobile devices (systems) with smartphone and cellular connectivity are developed using Windows Mobile SDK.
8. The programs can be tested on the familiar PC having 80x86 processor before embedding into the system. Emulation edition can be used to emulate an application on a PC. Evaluation kit is used to test the program before embedding into the actual system hardware with it or another processor.
9. Supports touch screen. A touch screen displays as well as accepts input through a stylus. (Stylus is a writing pencil-core-shaped object for a user of device or system as an alternative to mouse and keyboard. The user touches the tip at the displayed menu or displayed keypad on the screen to enter the commands or text, respectively.)
10. Network and communication protocols support, for example, OBEX (object exchange), Bluetooth, IrDA, UDP, TCP/IP, IPv6 and ethernet and many other popular protocols.
11. Shared source and source code access are provided by Microsoft. There is *componentization*. There are two software layers. One sublayer consists of Microsoft developed source codes of WCE kernel and is shared with the system or 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. Further, Microsoft gives freedom to modify kernel-level objects also without sharing them with the Microsoft.
12. Supports power manager, virtual memory, file-based registry and several file systems (e.g., flash memory-based file system). (Power manager is 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. Virtual memory are the addresses allocated for stored programs which may be of size more than the physical memory size. Section 8.6.2 explained the file system concept.)

Table 10.1 gives the new enhanced features of WCE in Windows CE 6.0, Windows Mobile 6 and Windows Automotive 5.0.

10.1.2 Windows CE Programming

Following are the differences in programming with WCE and Windows. WCE provisions for the following.

1. Win32 APIs subset only (e.g., no environment-related functions and environment blocks, no current directory information at the subset).
2. Small screen system.
3. Touch screen system.
4. No hard-disk, low RAM memory and use of ROM and flash memory in the system.
5. System processor can be x86 or ARM or SuperH, or any other.
6. Unicode 16-bit characters (unsigned *short*) so that international characters and languages can be used.
7. Reduced number of Windows controls in WCE compared with the personal computer.
8. New format Windows Controls (classes which support a number of GUI functions of command, menu, tool bars provided in one line due to small screen) and the new Controls [e.g., for date and time picker, calendar picker, edit to auto capitalize first character of a word when keying-in, virtual keyboard and organizer (e.g., task-to-do)].
9. Device drivers imported as the DLLs (Section 10.1.7).
10. Not support for the Handle inheritance and certain security attributes.
11. Componentization (Section 10.1.1).

Table 10.1 Windows CE 6.0, Windows Mobile 6 and Windows Automotive 5.0 Enhanced Features

S.No.	*Feature*	*Description*
1	Windows CE 6.0	Number of processes 2^{16} (earlier 32), lower virtual memory (VM) 2 GB (earlier 32 MB) addresses per process, upper 2 GB of the kernel VM space, device drivers running in both user mode and kernel mode (Section 8.1.2), system components which now run in kernel have been converted from EXEs to DLLs (Section 10.1.7), which get loaded into kernel space later at run time, new security infrastructure, 802.11i and 802.11e Wirelesss LAN support, new cell core for cellular networks and easy data connections, UDF 2.5 and exFAT file system, IDE integrated with Visual Studio 5.0 and targeted to enterprise specific tools such as industrial controllers and consumer electronics devices.
2	Windows Mobile, 6 second edition	Offfice Mobile 2007, smartphone with touch screen, improved Bluetooth stack, VoIP with AEC (acoustic echo cancellation), support to encryption of data stored in external removable storage cards, uses smart filter for fast files, e-mail, contacts and songs search, and can be set as modem for laptop.
3	Windows Automotive 5.0	Based on Windows CE 5.0 and building blocks for automobile off board service, automotive user interface toolkit (AUITK), expanded virtual memory support to enable the creation of complex 3-D graphics, and advanced navigation displays, enhanced power management and faster cold-boot times, real-time traffic updates, directions to the cheapest gas and improved performance.

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* (Sections 8.1.2 and 8.4) to the Windows application program. Messages are placed in queue (Section 7.12) for the Windows of the application program. The OS monitors all input sources [e.g., stylus tap, virtual (on-touch screen) or physical key press]. The OS notifies that a key has been pressed or a button has been clicked or command has been received for redrawing the Windows screen. [In Unix; it is the other way round. The application program asks for the input(s) from the OS for a character or commands or inputs from the keyboard.]

Window class instance defines a Window (object). The Window has basic coordinates x and y, and z-parameter. The z specifies whether Window is over and below other windows. The Window has specification for visibility (show or hide or no activate). The Window has specification for parent–child hierarchy. Windows procedures share the attributes, for example, Commandshow. Windows procedures are there to respond to requests and all notifications sent to the Windows.

WCE does not support Handle inheritance. [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 of a Window. It is an object, which is used as a 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*. The option pointer is pointer which points to a pointer of one of the several sets of the codes, which run on selecting the option. Windows support (WCE does not) Handle inheritance, which means a Handle can be extended to create a new Handle, which inherits the variables, properties and procedures of parent handle and adds, overrides and overloads new variables, properties and procedures.]

10.1.3 Windows and Windows Management

There are many Windows on a screen. A screen top (desktop) is a Window. A command-tool-task bar is a Window. A button is a Window. The Windows are related to each other. There may be a hierarchical (parent child) relationship in the Windows. There may be a sibling relationship or owner-owned relationship.

There is a top-level main Window. The main Window does not have a parent. The main Window can have child Windows. When a parent is moved or deleted, the all child Windows shall also move or delete. Child Window is invisible except at the edges. CreateWindowEx or CreateWindow creates a Window and uses the same messages and procedures as the main. A 32-bit style parameter dwStyle when set as WS_Child the child Window is created. An 8-bit style parameter bMenu parameter is used in child Window and equals the ID of that Window. (Prefix dw means double word and b means byte as data-types.)

Examples of management functions for the Windows are FindWindow (to find a Window and get Handle for that), GetParent to find the parent and GetWindow to query and get the owner, children and siblings.

10.1.4 Memory Management

Windows CE 6.0 permits virtual memory (VM) limit of 2 GB (earlier 32 MB) for each process and upper 2 GB VM space as the kernel VM space. Extended VM support enables the creation of complex 3-D graphics on WCE devices and therefore animation and gaming applications.

WCE provides for system memory between 1 MB and 64 MB and OS needs minimum 512 kB of memory and 4 kB 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*. The memory is allocated to the program from a pool of unused memory area called the heap. The application program that runs uses the heap and stacks. An application is allocated memory blocks (in place of the pages) from the heap and is in reserved virtual memory space region. A block in heap can also be freed later when not required. A heap can be a local heap of 188 kB or a separate heap in case of requirement of bigger number of memory blocks.

Object store (256 MB) is virtual RAM disk for permanent store, which is protected from turning off power. Individual file can use up to 32 MB in case of RAM as *object store*. PIM (personal information manager) data is also stored at the *store*. 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. A handheld PocketPC has a backup battery, which provides power to *object store* data and files. WCE at power-on searches the previously loaded *object store* at RAM and uses that object if available. The *object store* stores files, registry and WCE databases (Sections 10.1.5 and 10.1.6).

WCE saves in the ROM execute-in-place files. Execute-in-place file is a file in ROM for execution that cannot be opened and read by standard file functions open and read (Section 8.6.2).

WCE supports virtual and page memory. Virtual memory may be at the flash or disk. The application program uses the physical addresses at RAM. A virtual memory management system maps the virtual addresses of pages with the physical addresses of pages after the pages of the program has been loaded at RAM. A page is a fixed-sized memory unit, which is loaded from disk or flash to the RAM. WCE uses page size of 1 kB or 4 kB. It depends on the system processor. Three types of virtual pages are supported in WCE. Committed page is a page reserved for application and directly maps to the RAM address. A reserved page at virtual address cannot be used in the application. A free page can be used and is allocated during the run.

Static Allocations WCE allocates two allocations, one for read only and other for read/write data.

Stacks Stack stores the temporary variables and processor registers for the application and OS functions. WCE provides for separate stack for each thread (Section 7.2). WCE provides for 58 kB maximum stack size and 6 kB of stack for guarding the stack for underflow or overflow. An application can also specify the thread stack size.

10.1.5 Files and Registry

Files are created by a CreateFile function. It has the following arguments:

1. long pointer for character string,
2. 32-bit desired access parameter,
3. 32-bit shared mode specification,
4. long pointer for security attributes,
5. 32-bit to specify creation and distribution,
6. 32-bit specify flags and attributes and
7. Handle for template file.

The arguments are assigned as follows:

1. character string used for file name,
2. WCE file access parameter GENERIC_READ or GENERIC_WRITE or both,
3. WCE create file can set 32-bit shared mode specification as 0 or FILE_SHARE_READ, FILE_SHARE_WRITE,
4. WCE create file must use long pointer for security attributes as NULL,
5. WCE file creation and distribution specified by 32-bits CREATE_NEW, OPEN_EXISTING, CREATE_ALWAYS (new file after truncating existing), OPEN_ALWAYS (create new file if not existing else open) or TRUNCATE_EXISTING,
6. 32-bit specify WCE flags and attributes FILE_FLAG_WRITE_THROUGH, FILE_FLAG_RANDOM_ACCESS, FILE_ATTRIBUTE_NORMAL, FILE_ATTRIBUTE_READONLY, FILE_ATTRIBUTE_ARCHIVE, FILE_ATTRIBUTE_SYSTEM, FILE_ATTRIBUTE_HIDDEN, FILE_ATTRIBUTE_NORMAL and
7. Handle for template file is NULL as WCE does not support file template.

WCE files differ and have similarities in following respects for other Windows OS.

(1) Uses most of the Win32 file APIs.

(2) Uses standard file IO procedures CreateFile, OpenFile, ReadFile,WriteFile, SetEndOfFile and CloseFile (Section 8.6.2). These functions create, open, read, write, truncate and close the files (not for execute-in-place files in ROM).

(3) WCE files use the same attribute flags as in Windows. Examples are flags for read only, compressed, archive and system hidden.

(4) WCE used up to 256 storage devices or partitions on the storage devices. A installable file system driver can be installed for flash and other file systems.

(5) Uses object store as a default RAM-based file system,

(6) Current directory concept missing. A file name and complete path specification (maximum 260 bytes of MAX_PATH length) is required in WCE. There is no mention like C or D drive when using the files. A file has three-character extension format after the dot sign. The extension defines the file type. For example, .txt for text file.

(7) Memory-mapped files and objects are supported for reading the files as byte streams.

(8) Uses compact flash and RAM with backup battery.

WCE registry uses standard registry API. Registry API is an API for system database. It uses standard file registry functions. The functions are as follows: RegCreateKeyEx, RegOpenKeyEx, RegSetValueEx, RegQueryValueEx, RegDeleteKeyEx, RegDeleteValueEx and RegCloseKey.

Registry system has keys and their values as in a hash table. *Keys* can contain the keys. There are multiple-level keys. Permitted data types for registry are 32-bit numbers, string or free (for binary data).

10.1.6 Windows CE Databases

Table 10.2 gives the procedures (functions) and properties of WCE databases.

Table 10.2 Windows CE Functions in WCE Databases

S.No.	*Feature*	*Description*
1	Database format	Series of un-lockable records with saving of the property(ies) and data together in a record. No record contains another record within it. Each record has four-level indices for sorting.
2	Maximum number of records and record size	$2^{16} - 1$ and 2^{17}
3	Database record properties data types	Double 64-bit signed, boolean, collection of bytes, 0-temined unicode string, 16-bit signed, 16-bit unsigned, 32-bit signed, 32-bit unsigned, time and date structure
4	CeMountDBVol Windows CE function	To mount an external flash or media on a database volume, which stores the database files (not object-store files)
5	CeCreateDatabaseEx Windows CE function	To create new database
6	CeOpenDatabaseEx Windows CE function	To open created database
7	CeSeekDatabaseEx Windows CE function	To set pointer to database record
8	CeSetDatabaseInfoEx Windows CE function	To set the sort order of opened database
9	CeReadrecordPropsEx Windows CE function	To read a record of given property
10	CeWriteRecordProps function	To write new property of record
11	CeDeleteDatabaseEx function	To delete records, properties and complete database
12	CloseHandle ( )	To close the Handle
13	CeUnmountDBVol Windows CE function	To un-mount an external flash or media form a database volume, which was mounted earlier
14	Windows CE notification	Notify to a process that a thread has modified the database.

10.1.7 Processes, Threads and IPCs

WCE has Win32 executable files, called modules in two forms, one .exe files and the other the DLLs (dynamic-linked libraries). The .exe files are compiled files and are first loaded in system memory for execution. The DLLs are loaded at run time on request from the .exe file or another DLL. The loading is by using a list of DLLs and the functions list within the DLLs at an import table in the .exe file. The DLL also has import table for other DLLs and their functions.

WCE is a multitasking multithreaded (Sections 7.1, 7.2 and 7.3) OS. In a multitasking OS, an application consists of the number of processes. The process is an instance of an application. There can also be multiple copies of the same processes. In a multitasking multithreaded OS, a process can create number of threads which execute concurrently. There is at least one thread which is created per process. There is context switching between the threads of the same processes or from one process thread to another process thread. Basic unit of execution under control of kernel is thread. Each thread has separate context (for CPU register including PC and stack pointer) and stack for saving context when thread blocks and for retrieving on thread activation.

At least four WCE processes load on start up. These are FileSys.exe (for file system functions), GWES.exe (for GUI functions), Device.exe (for loading and maintaing device drivers) and NK.exe (for kernel functions).

Table 10.3 gives the procedures and properties of processes and threads.

Table 10.3 Properties and Windows CE Functions for Threads and Processes

S.No.	*Feature*	*Description*
1	Thread properties	Threads of the same process share the memory space and manage access permissions. The Handles are used for synchronization objects (interprocess communication objects), file and memory objects. A primary thread can create secondary threads.
2	Thread priorities	Each thread assigned priorities one among eight levels[1]: system-level threads and device drivers (ISTs) use upper 248 levels of priorities. Total number of priority levels is 256 and priority 0 value is highest and 255 is lowest. Higher priority thread pre-empts lower priority thread. Priority inversion (Section 7.8.5) is taken care of during execution.
3	Maximum number of processes and memory size	Windows CE 6.0 supports number of processes 2^{16} (earlier 32) with each process virtual memory limit of 2 GB (earlier 32 MB) and up to lower 2 GB (earlier 32 MB). Also there is upper 2GB (earlier 32 MB), which is the kernel VM space.
4	CreateProcess WCE function	To create a new process with four parameters required in the arguments (long pointers for the ApplicationName, Commandline (a unicode string), and ProcessInformaton[2] and 32-bit CreationFlags[3] (to specify initial state on loading).
5	TerminateProcess Windows CE function	To terminate a process. Two arguments are Handle for process and 32-bit exit code for the process. The exit code for the process is obtained from GetExitCodeProcess function.
6	OpenProcess Windows CE function	To open a process using processId as argument. ProcessId is obtained from GetWIndowThreadProcessId .
7	CreateThread WCE function	To create new process with following parameters required in the arguments: 32-bit StackSize, long pointer for THREAD_START_ROUTINE (an address of routine for starting thread execution), long pointer Parameter for application-specified thread parameter(s), 32-bit CreationFlags[4] (to specify initial state on loading) and 32-bit ThreadId (thread ID).
8	ExitThread Windows CE function	To terminate a thread using one argument, which is 32-bit exit code for the thread. The exit code for the thread is obtained from GetExitCodeThread function.

(*Contd*)

S.No.	Feature	Description
9	SetThread-Priority	To set thread priority among eight levels. Handle for thread and *priority* value to be set are the arguments. Value = 250 for thread set to normal priority level.
10	CeSetThread-Priority	To set thread priority value among 256 values between 0 and 255. Handle for thread and *priority* value are the arguments.
11	GetThread-Priority	To get thread priority level. Handle for thread is the argument.
12	CeGetThread-Priority	To get thread priority value among 256 levels between 0 and 255. Handle for thread is the argument.
13	CeSetThread-Quantum	To set thread time slice value. Handle for thread and 32-bit *time slice* in milliseconds are the arguments.
14	CeGetThread-Quantum	To get thread time slice value. Handle for thread is the argument.
15	Sleep	To delay the thread execution for a period. 32-bit time slice in milliseconds is the argument.
16	SuspendThread	To suspend a thread. Handle for thread is the argument.
17	ResumeThread	To resume a suspended thread. Handle for thread is the argument.

[1] Eight levels are idle (= normal – 4), above idle (= normal – 3), lowest (= normal – 2), below normal (= normal – 1), normal, above normal (= normal +1), highest (= normal + 2) and time critical (= normal +3).

[2] ProcessInformaton consists of a Handle object for process, a Handle object for thread, 32-bit processId and 32-bit threadId.

[3] CreationFlags = 0 for standard process, = CREATE_SUSPEND for create and suspend, = CREATE_NEWCONSOLE for creating a new console, = DEBUG_PROCESS for process passing debug information to calling process and = DEBUG_ONLY_THIS_PROCESS for process passing debug information only from this process and not from CHILD PROCESSES.

[4] CreationFlags = 0 for standard thread, = CREATE_SUSPEND for create and suspend (Must use ResumeThread if created thread is suspended) and = STACK_SIZE_PARAM_IS_A_RESERVATION for creating thread with reserved stack size.

Exceptions, Notifications and IPC Objects Synchronization WCE provides for exception handling signals and notification signals (Section 7.10). Notification examples are notifications from timer, serial device detect, power up, system event. Notification can generate a dialog and a sound or run a notification responding function.

WCE also provides Handles for the IPC objects. An IPC object in a multitasking or multithreading system is used to generate a synchronization object and the object gives the information about certain sets of computations finishing one process or thread and to let the other process or thread waiting for that object to get information about finishing the computations that take note of the information. An IPC object is released (sent) which means that a process (thread or scheduler, task or ISR) generates some information by or value or generates an output so that it lets another process waiting for that object in order to take note or use the object. A kernel provisions for the functions for creating, releasing and waiting the IPC objects (Section 7.9).

IPC objects in a multitasking or multithreading system are events (Section 8.4), semaphores including mutex semaphores (Section 7.11) and message queues (Section 7.12). WCE provides for the events, semaphores including mutex semaphores and message queues for threads synchronization. Table 10.4 gives these for exceptions, notifications, events, semaphores, mutex and message queues.

10.1.8 Inputs from Keys, Touch Screen or Mouse

A keyboard is used to enter many characters, commands or large text. Physical keyboard is inconvenient in a handheld device. There is a soft keyboard. It controls and simulates the virtual keyboard on touch screen. An

application can get the input either from pysical keyboard or from soft keyboard. A function SetFocus is used to specify the focused Window so that the input directs to that Window. Windows sends a series of messages for the Window in focus. Every key or action has an assigned value. For example, a virtual key value is VK_LBUTTON which passes a value 01 on a stylus tap. A virtual key value is VK_RETURN, which passes a value 0D when the Enter key is pressed.

Table 10.4 Windows CE Functions for Exceptions, Notifications, Events, Semaphores, Mutex and Message-Queues

S.No.	*Feature*	*Description*
1	Signalling (exception) functions	RaiseException using the 32-bit exception code, exception flags, number of arguments and 32-bit constant for array of long pointer for the arguments. Each argument passes the data to the exceptional responding routine (like ISR).
2	Signalling (notification) functions	1. CeSetUserNotificationEx using the Handle for notification, CE_NOTIFICATION_TRIGGER object[1] and CE_USER_NOTIFICATION object (object pointer defines action flags, dialog title, dialog text, sound and other details). 2. CeGetUserNotificationEx using the Handle for notification and long pointer for CE_USER_NOTIFICATION object. 3. CeClearUserNotificationEx using the Handle for notification to acknowledge a notification by notification responding function.
3	Critical section functions	InitializeCriticalSection (to initialize a critical section), EnterCriticalSection (to enter a critical section), LeaveCriticalSection (to exit a critical section),TryEnterCriticalSection (to try to enter a critical section) and DeleteCriticalSection (to delete a critical section).
4	Semaphore functions (for threads of a process)	CreateSeamphore (to create the semaphore), ReleaseSemaphore (release semaphore to let waiting thread code unblock), CreateMutex (to create the mutex), ReleaseMutex (release mutex to let waiting thread code unblock).
5	Message queue functions	CreateMsgQueue (to create the message queue), OpenMsgQueue (to open a message queue), ReadMsgQueue (to read from the queue), GetMsgQueueInfo (to query the queue), WriteMsgQueue (to write into the queue), CloseMsgQueue (to close an open message queue).
6	Event functions (for threads of a process)	CreateEvent (to create the event), SetEvent (event set to signal occurrence of the event and do not auto-reset till waiting thread unblocks) (event auto-resets on unblocking of thread), ResetEvent (to force reset of the event and unblock the thread waiting for it), PulseEvent (to set the event and then reset the event by unblocking all waiting threads for that event), SetEventData using event-handle and 32-bit data in the arguments, GetEventData using event-handle to get the event data.
7	Wait Single object functions	WaitForSingleObject using object Handle and 32-bit waiting time value in milliseconds in the arguments.

(*Contd*)

S.No.	Feature	Description
8	Wait for multiple objects	WaitForMultipleObjects using count number for objects (events or mutexes), pointer to array of object Handles, boolean WaitAll (if true then wait for all, in WCE must be set to false) and 32-bit waiting time value in milliseconds in the arguments. Each object handle is a long pointer. Waiting time value = INFINITE disables the timeout specification For wait for the multiple objects.
9	Wait for message objects	MsgWaitForMultipleObjectsEx using count number for message objects, long pointer to array of object Handles, boolean WaitAll (if true then wait for all, in WCE must be set to false), 32-bit waiting time value in milliseconds in the arguments and 32-bit flags for WakeMask.[2]

[1] CE_NOTIFICATION_TRIGGER object pointer defines type and notification details of notification type, notification size, notification event, notification application, notification arguments, notification start time and notification end time.

[2] WakeMaskFlags = QS_ALLINPUT for any message received, QS_TIMER for a WM_TIMER (Windows Manager timer) message, QS_PAINT for a WM_PAINT Windows manager paint, QS_SENDMESSSAGE (a sent message outside the list received), QS_POSTMESSSAGE (a posted message outside the list received), QS_MOUSE a mouse move or click or stylus tap received, QS_MOUSEMOVE a mouse move or stylus move received, QS_MOUSEBUTTON a mouse click or stylus tap received.

A keyboard function example is SHORT GetKeyState (int iVirtKey) for querying a keyboard key. An application can simulate key-event.

Inputs from Touch Screen or Mice Touch screen for input is equivalent to a single button mice input. Further, the mice has a cursor. When a mice is pressed, the window is sent the message WM_LBUTTONDOWN on left button down and release of WM_LBUTTONUP on left button up event.

WM_MOUSEMOVE message is sent when the stylus is moved within the same window. When the stylus is dragged outside the in-focus window, the WM_MOUSEMOVE messages stop. If SetCapture procedure is called then, WM_MOUSEMOVE messages continue. ReleaseCapture stops sending the messages of WM_MOUSEMOVE.

GetMouseMovePoints sends the messages for each point traced by stylus on the screen from a start to end. GetMouseMovePoints integrated with handwriting recognizer application can be used handwriting on the PocketPC to write the text or commands or messages.

WM_LBUTTONDBLCLK message is sent on the double tap of the stylus. For each message the parameters lParameter = two 16-bit screen tap horizontal and vertical position values x and y, and wParameter = 16 bits for the flags corresponding to which key shift or control held down or not.

Right button click of mice is simulated using stylus when ALT key is held down while tapping.

Windows Controls Each Window uses a number of classes, called *Controls*. A control has a number of user-interface elements. The user-interface examples are *button*, *radio* and *checkbox*. The user-interface elements are predefined for a Control and there exists a Windows Control library. Predefinition and library help in each application window has same feel and look.

A Control is also a Window and is created by CreateWindowEx or CreateWindow. A control may be static control. It displays a text (as per defined alignment) or icon or bitmap. A control is scroll bar control.

Most powerful Control for user interface is *Button*. Button appearance can be set. An owner window can also draw the owned button.

1. Simple button is a push-button. When the stylus taps the button, it generates a WM_COMMAND message with a 16-bit parameter wParameter for the flag BN_CLICKED. BN_CLICKED flag specifies that button is clicked.
2. Checkbox button is a square box with blank or filled circle or a label using which the user specifies a choice by tap stylus at that point and which toggles between the blank and the filled.The Checkbox toggles between two states.
3. Radio button is a button to allow the user to select among the interrelated choices and when one selects the other may unselect. Application checks or unchecks a radio button.
4. Group box button. It is an empty box with a text label. The text in the box gives the interface for programming.

A Control is list box control. It is used for selecting among the list of items displayed by text. WCE supports a constant string data style in list box control. The style is called LBS_EX_CONSTSTRINGDATA. Only the pointer to the string saves at the Window and not the string. The application is supposed to manage that string.

A Control is Edit control. It is used for keying in the text and editing it. The keyed text is in upper cases when ES_UPPERCASE style is set. When ES_LOWERCASE is the style, the edit text appears as lower cases. The keyed text is visible as *** when ES_PASSWORD style is set.

A Control is combo-box control. One can use two or more controls in the combo box. A combo box in WCE is drop-down or drop-down list. Drop-down is an edit-text field control with a button on the right side. When this button is clicked a list box for selection appears. Drop-down list is a list of texts each with a button on the right side. The stylus taps at any one of it to choose.

Windows Menus WCE menus are at the menu bar or command bar control. The CreateMenu, AppendMenu, InsertMenu are the procedures in WCE to create, append or insert a menu item. CreatePipupMenu is a procedure to nest the menus. Window generates WM_COMMAND message and the ID parameter of the menu item is sent.

10.1.9 Communications and Networking

WCE serial port is a stream device driver, which is also opened by CreateFile (Section 10.1.4). WCE has the functions for clearing errors, querying status, timeout values setting and getting, querying the serial driver and controlling the communication. Table 10.5 gives the Windows CE serial communication functions.

WNet API is an API for networking support to Windows. It has a feature of accessing network resources that does not depend on platform and implementation of network functions. WCE supports a subset of WNet. Table 10.6 gives the WNet API subset network connection functions.

10.1.10 Device-to-Device Socket and Communication Functions

Devices, such as mobile phone or PocketPC establish synchronization with the neighbouring devices and computers, and form a personal area network (PAN). Examples of protocols used in PAN are Bluetooth and IrDA (Infrared Data Association).

An API is Winsock API for sockets programming support to Windows. The TCP/IP, Bluetooth and IrDA network sockets are programmed using Winsock. Winsock supports streaming sockets and datagram (Section 3.11.3) connections. (Streaming sockets and datagram difference is that there is connection between two APIs at different devices, and between two specific addresses at two APIs at different devices, respectively.) Winsock has a feature of accessing PAN resources through the sockets that does not depend on platform and implementation of socket functions (Section 7.15). WCE supports a subset of Winsock 1.1 and 2.0 API. Table 10.7 gives the Winsock API subset in WCE for device-to-device socket communication functions.

Table 10.5 Windows CE Serial Communication Functions

S.No.	*Feature*	*Description*
1	CreateFile WCE function	Creates the port for communication. Returns a Handle for serial COM1 port. The arguments used are TEXT ("COM1"), GENERIC_READ | GENERIC_WRITE, 0, NULL, OPEN_EXISTING, 0, and NULL.
2	ReadFile	Reads from the port for communication. Returns an integer. The arguments used are Handle returned on creation, pointer to character, pointer to 8-bit number of bytes read, NULL.
3	WriteFile	Reads from the port for communication. Returns an integer. The arguments used are Handle returned on creation, pointer to character, pointer to 8-bit number of bytes read, NULL.
4	TransmitCommChar	Send character into queue for port transmission. (Control characters can be inserted into the stream). Returns a *boolean* for successful or unsuccessful transmission. The arguments used are Handle returned on creation and character for transmission.
5	Set CommMask	To set communication mask. The arguments used are Handle returned on creation and 32-bit for event mask to specify clear to send, break, data set ready, error, receive line signal detect, character received, a receive-event's flag received, transmit buffer empty.
6	Get CommMask	To get communication mask. The arguments used are Handle returned on creation and long pointer for 32-bit event mask.
7	WaitCommEvent	To wait for event. Handle for file, long pointer for 32-bit event mask and NULL (for long pointer for overlap) are the arguments.
8	SetCommState	To set communication state. Handle for file and long pointer to device control block (DCB) structure are the arguments. DCB defines 32 bits for DCB length, baud rate, binary flag, parity flag and 24 other flags.

Table 10.6 WNet API Network Connection Functions

S.No.	*Feature*	*Description*
1	WNetAdd-Connection	Maps the network (remote) resource. Returns a 32-bit code for no error or error. The arguments used are one Window handle, three long pointers for *network resource* and string for password and user names and one 32-bit value for the flags. The *network resource* is a structure, which contains long pointers for remote name and local name.
2	WNetConnection-Dialog	To dialog. The argument used is a long pointer for connection dialog structure.
3	WNetCancel-Connection	Disconnects the network (remote) connection added earlier. Returns a 32-bit code for no error or error. The arguments used are one long pointer for name (local or remote), 32-bit value for the flags, boolean to specify forced disconnection when files or devices are open and not closed.

(*Contd*)

S.No.	Feature	Description
4	WNetDis-connectDialog	To dialog on disconnection. The arguments used are a Window handle and 32 bits for resource type. Resource type may be printer or disk or any other that is available. There is another overloading WNetDisconnectDialog, which has one argument, a long pointer for disconnection dialog structure.
5	WNetGet-Connection	Queries the network (remote) resource connection. Returns a 32-bit code for no error or error. The arguments used are long pointers for strings for name (local or remote) and user name and long pointer for 32-bit value, which specifies the length of remote buffer characters.
6	WNetGetUser	Queries the user name. The arguments used are long pointers for strings for local and remote names and long pointer for 32-bit value, which specifies the length of remote buffer.
7	WNetGetUniversalName	Queries the name as per universal naming convention. The arguments used are long pointer for string for local path, 32-bit info-level, long pointers for buffer address and 32-bit buffer size.
8	SetCommState	To set communication state. Handle for file and long pointer to device control block (DCB) structure are the arguments. DCB defines 32 bits for DCB length, baud rate, binary flag, parity flag and 24 other flags.

Table 10.7 Winsock API Subset in WCE for Device-to-Device Socket Communication Functions

S.No.	Feature	Description
1	Socket function	To create new Socket. is like a Handle. The function has three parameters required as the arguments: three integers, one for addressed family specification (e.g., AF_BT, AF_IRDA or AF_INET for Bluetooth or IrDA or TCP/IP, respectively), second for addressed socket type specification (e.g., SOCK_STREAM or SOCK_DGRAM for stream or datagram socket, respectively) and third for protocol (e.g., BTHPROTO_RFCOMM for Bluetooth RF communication).
2	Bind function	To find the desired server. Three arguments are SOCKET (of server), constant structure for addressed socket information (e.g., SOCKADDR_BTH, SOCKADDR_IRDA or SOCKADDR_INET for Bluetooth or IrDA or TCP/IP, respectively) and integer for name length.
3	Accept function	To accept client connection at the server in listen mode. Three arguments are SOCKET (of server) already in listen mode, structure for addressed buffer information (for SOCKADDR_BTH, SOCKADDR_IRDA or SOCKADDR_IN for Bluetooth or IrDA or TCP/IP, respectively) and integer for buffer length.

(*Contd*)

S.No.	Feature	Description
4	Connect function	To connect a newly created client socket to server (client does not call bind and accept function). Three arguments are connecting (client) SOCKET, constant structure for addressed socket information (e.g., SOCKADDR_BTH, SOCKADDR_IRDA or SOCKADDR_INET for Bluetooth or IrDA or TCP/IP, respectively) and integer for name length.
5	Listen function	To connect to server for data after binding. Two arguments are SOCKET and integer for queue size (= SOMAXCONN for maximum size) for pending connection.
6	Send function	To send from a socket. Four arguments are SOCKET (of sender), constant char for addressed buffer information (for SOCKADDR_BTH, SOCKADDR_IRDA or SOCKADDR_INET for Bluetooth or IrDA or TCP/IP, respectively) and two integers one for length and other for flags.
7	Recv function	To receive from a SOCKET. Four arguments are SOCKET (of receiver), constant char for addressed buffer information (for SOCKADDR_BTH, SOCKADDR_IRDA or SOCKADDR_INET for Bluetooth or IrDA or TCP/IP, respectively) and two integers one for length and other for flags.
8	Shutdown	To shut the send and receive functions on a close of the socket connection. Two arguments are SOCKET and integer for closing how (= SD_BOTH or SD_SEND or SD_RECEIVE for shutdown of both send or receive or send function only or receive function only, respectively)
9	Close socket	To close the socket connection. The argument is SOCKET to be closed.

10.1.11 Win32 API Programming

The development of Windows for the GUIs is often the most important part of application development in a computer or embedded system or handheld system which has a screen or touch screen for interaction with a user. GUIs facilitate interaction and inputs from user after graphic screen displays of menus, buttons, dialog boxes, text fields, labels, check box and radio buttons and others. Win32 API programming is thus a very important part of any application development. Win32 has large number of APIs in a PC. However, only a subset is required for handheld devices and small screen size systems. A subset of Win32 APIs is provided in WCE.

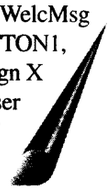
When an application is developed, a Windows displays the messages in the central region, title, command, tool and status bars. The Windows also displays commands (buttons) so that a stylus tap (or mouse click) sends the selected command using menu and buttons. The Windows also show the icons for maximizing, minimizing and closing at right-hand side top corner. Windows also show icon for Help (to help the user) and a ? sign icon (to show more buttons on a tap there). WCE has single-line controls for command, tool and status bars. A stylus tap or mouse click sends the menu choice to the application. WCE has new format for the Windows Controls (command, menu, toolbar bars) and new Controls (data, time, calendar) and organizer (e.g., task-to-do).

The following example shows a simple application of Win32 API. The example shows how simple it is to create the screen windows and show the commands (buttons) for further action in an application.

Example 10.1

int WINAPI WinMain (HINSTANCE hPresentinstance; HINSTANCE hPreviousinstance, LPWSTR lpCommandline, int iCommandshow) {MessageBox (NULL, TEXT ("Welcome"), TEXT ("WelcMsg"), MB_DEFBUTTON1, MB_DEFBUTTON2, MB_ICONQUESTION);
return 0; } /* After third argument the last argument(s) is one or more among the series of flags which can be used for showing the buttons or icons in MessageBox Windows bar. The buttons and icons must be those as provided for in the procedure MessageBox. */

1. The Presentinstance is a parameter to identify the present instance, Previnstance is a parameter to identify the previuosinstance (WCE always assumes it to be zero).
2. Commandline is a unicode string (it specifies the functions of the program).
3. Commandshow is an integer to specify *state* of the program, which defines a configuration of main window. The state parameter is passed from the parent application to a new application. The state configuration in a personal computer can be the one which shows minimized, maximized or normal icons. WCE allows only three states and configuration of WCE Windows is as per variables show without activate (SW_SHOWNOACTIVATE), show hidden (SW_HIDE) and show normal (SW_SHOW). Default value of Commandshow is used as per the value for the main Window show command.
4. MessageBox creates a window over the main window. The window shows messages in the box until window is closed. It shows: (i) no other Windows because first argument is NULL; (ii) text message Welcome in the unicode message window (at center) and text Unicode message caption (title) WelcMsg at left corner in the command-cum-tool-status bar; (iii) buttons as per definitions MB_DEFBUTTON1, MB_DEFBUTTON2 in the middle of command bar and icon of ? ; (iv) at the end of bar, a sign X icon is created at the right corner in the bar. The X enables the closing of the window by the user on tapping on the touch screen or mouse click. [MessageBox is used here in place of printf otherwise a driver console.dll needs to be added to enable printing on console (screen).]



This example uses Handle. INSTANCE is a Handle object. In the present case, a handle is a reference to an interface, which handles a Window instance.

This example uses prefixes before the objects and variables as follows. Prefix H before object INSTANCE indicates that it is a Handle object. Prefix LP before WSTR indicates that it is a long pointer. Prefix W before STR indicates that the string is a 16-bit unsigned word. This method or prefixing helps in easier understanding of the objects and variables by a programmer.

10.1.12 Creating Windows

An application can create its own Windows instead of creating the Windows using MessageBox as in the previous example or a similar function in the Win32 subset of WCE. There are several Windows procedures that can create its own Windows. Following are the examples.

1. CreateWindowEx, which creates main window.
2. MainWndProc, which creates application window.
3. For example, WM_Paint to draw the window background and put text within it at the specified position after first creating a client rectangle.

Drawing on Screen WCE does not support full Win32 graphics API and different mapping modes in Windows. WCE does not support coordinate transformations. A text is written using DrawText procedure. WCE always sets device context in MM_TEXT mapping mode.

Windows application does not write directly to the screen. It requests a Handle. The handle draws and displays device context. Device context specifies the application Windows. Windows sends the pixels to screen using the device context. A device context is a tool, which Windows use to manage the access to the display and printer. Two attributes of device context are colours for background and foreground. Text alignment attributes of device context are left, right, top, centre, bottom, no update and update of current point of device context and baseline alignment. Font of the displayed text from the device context can be specified. Font can also be created for an application as alternative to WCE default fonts.

A bitmap is a graphical object. Bitmaps can be drawn. The bitmaps are used to create, retrieve images, manipulate and draw at the device context. WCE supports format of bitmap in four colours. WCE permits 1, 2, 4, 8, 16 and 24 values and provides for compaction.

WCE provides for drawing lines, rectangle, circle, ellipse, round rectangle and polygon shapes and has a pen tool. WCE provides for fill functions for the draw, for example, gradient fill (shade changing on moving vertically up to down or horizontally from left to right) and hatched filling.

10.2 OSEK

RTOSes described in Sections 9.1 to 9.3 do not suffice for automotive systems, which require other necessary features. Embedded software in the automotive system needs special features in its OS over and above the MUCOS or VxWorks features and MS DOS and UNIX. Special OS features needed are as follows.

1. *Language* can be application-specific, need not be just C or C++ and *data types* should also be application-specific and not RTOS-specific. In VxWorks, for example, STATUS is RTOS specific. This is not permitted, as it could be the source of a bug and thus unreliable.
2. *OS*, every method, class and run-time library should be scalable. This optimizes the memory needs.
3. *Tasks* can be classified into four types. This provides a clear-cut distinction to a programmer: which class to use for what modules in the system.
   (a) Basic with one task of each priority and single activation. It is called BCC 1 (Basic Conformance Class 1).
   (b) Extended with one task of each priority and single activation. It is called ECC 1 (Extended Conformance Class 1). Extended task means, for example, a task created by FirstTask in Example 9.8.
   (c) Basic with multiple tasks of each priority and multiple times activation during run. It is called BCC 2.
   (d) Extended with multiple tasks of each priority and multiple times activation during run. It is called ECC 2.
4. OS can schedule ISRs and tasks in distinct ways. (VxWorks scheduler also does os.)
5. Interrupt system disables at the beginning of the service routine and enables on return. This lets the task run in real-time environment.
6. Task can be scheduled in real-time.
7. Task can consist of three types of objects, *events* (semaphore), *resources* (statements and functions) and *devices*. There are port devices also. An exemplary device is *alarm*. It displays the pictograms, messages and flashing messages. It sounds buzzer and beeps.
8. Timer, task or semaphore objects creation and deletion cannot be allowed. A run-time bug may lead to uncalled deletion of a timer or semaphore. That is the potential source of a problem and thus unreliable.
9. IPC, message queue posting by a task, is not allowed as a waiting task may wait indefinitely for its entire message needs. RTOS queue types, waiting infinitely or for a time out for a message can be a potential source of trouble and thus unreliable. Similar risks may arise with semaphore as a resource key or counter. These are therefore not used.

10. Before entering a critical section and on executing a service routine, all interrupts must disable and enable on return only (Refer to Section 7.8).

There is incompatibility of control units made by different automobile manufactures due to different interfaces and protocols. Software in the automotive electronics must also be standard.

A structured and modular software implementation based on standardized interfaces and protocols as proposed by OSEKVDX is a necessity. This gives the portability and extendibility, and thus the reusability of existing software. Presently, the important software standards and guidance are AMI-C (Automotive Multimedia Interface Collaboration) [*http://www.ami-c.org*], MISRA-C (Motor Industry Reliability Association standard for C language software guidelines for automotive systems) [www.misra.org.uk] and OSEK/VDX for RTOS, communication and network management. (Refer to website *http://www.osek-vdx.org* and also to a book *Programming in the OSEK/VDX Environment* by Joseph Lemieux from CMP Books, Oct. 2001.)

OSEK is an acronym for Offene Systeme und deren Schnittstellen für die Elektronik in Kraftfahrzeugen (eng., "Open Systems and their interfaces for the Electronics in Motor vehicles"). A German automotive company consortium (BMW, Robert Bosch GmbH, DaimlerChrysler, Opel, Siemens and Volkswagen Group) and the University of Karlsruhe founded OSEK in 1993. Later VDX (Vehicle Distributed eXecutive) from Renault and PSA Peugeot Citroën joined the consortium.

OSEK/VDX is a body for defining and specifying three standards for an embedded OS.

1. One is for real-time execution of ECUs (electronic control units) software and base for the other OSEK/VDX modules using *MODISTARC* (methods and tools for the validation of OSEK/VDX-based distributed architectures and for conformance testing of the OSEK/VDX implementations.
2. Second is for *communications stack* for data exchange within and between control units.
3. Third is for *network management* protocol for automotive embedded systems configuration determination and monitoring.

OSEK/VDX has also produced other related specifications. There are two standard alternatives for OS and communication systems. One is for normal requirements (standard operating system/communication specifications). Second is for the special requirements. Special globally synchronized fault-tolerant architectures are covered using OSEK time specifications. Figure 10.2 shows OSEK basic features. OSEK provides for functional extendibility. One can integrate new application functions into a single control unit together with other APIs. OSEK provides for APIs porting. There is easy transfer of application functions from one hardware ECU platform to another with only minor modifications.

OSEK specifies that extendibility and portability should be independent of the source of APIs and co-existence of software from different sources must be possible. It shall be remarked that OSEKVDX does not prescribe the implementation of OSEKVDX modules, that is, different ECUs may have the same OSEKVDX interfaces, but different implementations, depending on the hardware architecture and the performance required.

OSEK defines three standards.

1. OSEK-OS for OS, which has greater reliability. It is because, in an OS based upon OSEK, the previous ten points are taken care of.
2. OSEK-NM architecture for network management. As in OS, tasks are divided into four types, and the NM divides the architecture into two types. (i) Direct transfer and interchange of network messages; (ii) indirect transfer and interchange, both between the nodes.
3. OSEK-COM architecture for IPCs between the same CPU control unit tasks and between the different CPU control unit tasks. Between different unit tasks, the data link and physical layers exists. Different CPU physical layers connect by CAN bus architecture.

OSEK OS standard provisions for greater reliability compared with VxWorks or MUCOS. The MUCOS provides OSEK/VDX extension and provides the programmer the user a certified OSEK/VDX application programming interface. Reliability is introduced by the interface because the extension supports the OS conformance classes BCC1 and ECC1. The COM conformance classes CCCA and CCCB are provided for

internal communication. Extension does not permit creation and deletion of tasks during run. Extension defines each task of different priority and activates it only once in the codes. Extension does not use message queues and uses semaphores as event flag only with no task having run-time deletion or creation of these.

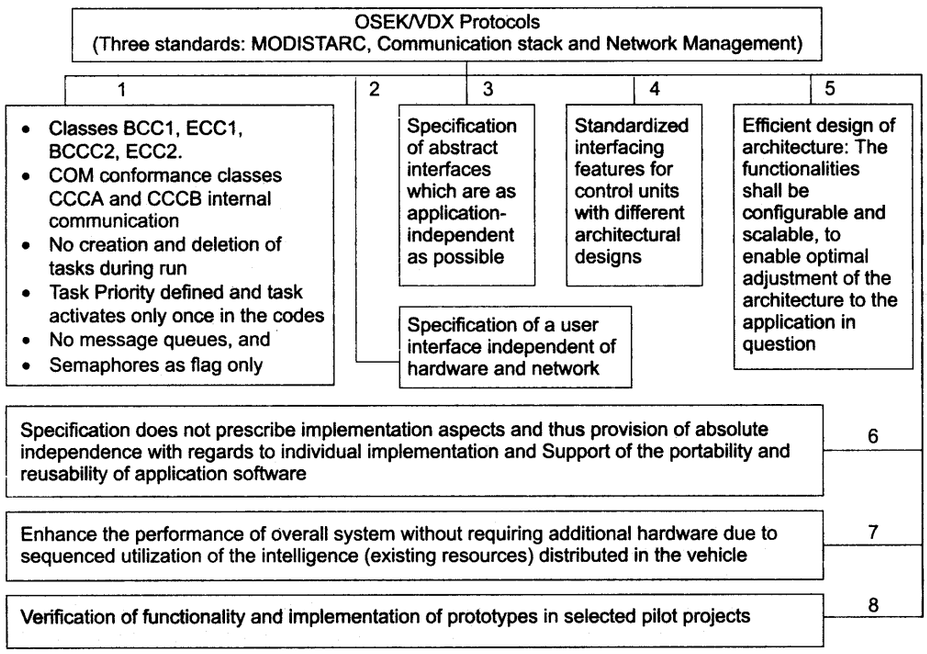


Fig. 10.2 OSEK basic features

10.3 LINUX 2.6.x AND RTLINUX

Linux is increasingly used in embedded systems and real-time enhancements of that have been introduced as Linux 2.6.x, the enhancements provide preemptive scheduling, high resolution timers and preemptive interrupt service (handler) threads. Linux latest version is Linux 2.6.24, released in January 2008. Reasons for the OS popularity are that it is a freeware, has device driver features, has expandability of the kernel codes at run time and has provision of registering and deregistering device-driver modules. The modules are scheduled like the processes. Sections 10.3.1 and 10.3.2 describe the open-source real-time Linux and RTLinux (a real-time Linux) (both open source and profession).

10.3.1 Real Time Linux Functions

The OS named Linux is after Linus Torvalds, father of the Linux OS. Embedded Linux systems combine the Linux kernel with a small set of free software utilities. The glibc is often replaced as the C standard library by less resource-consuming alternatives such as dietlibc, uClibc or Newlib.

Embedded Linux application program (tasks) makes the system calls or message passing (Section 8.1.2) for the functions at a kernel for. 1. Process management, 2. Memory management (e.g., allocation, de-allocation, pointers, creating and deleting the tasks), 3. File system, 4. Shared memory, 5. Networking system functions,