

# **EFOS API Guide (Rev 1.0)**

**Copyright © 2004  
VINJEY Software Systems (P) Ltd**

(This page is intentionally left blank)

**Copyright © 2004 VINJEY Software Systems (P) Ltd.**  
**All rights reserved**

No part of this document covered by the copyright hereon may be reproduced, transmitted, transcribed, stored in a storage media, or translated into any language in any form or by any means without prior written permission from VINJEY Software Systems (P) Ltd, Bangalore, India.

**Disclaimer**

VINJEY Software Systems (P) Ltd reserves rights to make changes without prior notice to any products herein. VINJEY Software Systems (P) Ltd makes no representations or warranties with respect to the contents hereof. Further VINJEY makes no warranty, representation or guarantee regarding the suitability of product for any particular purpose.

**Trademarks**

All trademarks are the property of their respective owners

Contact Details

VINJEY Software Systems (P) Ltd,  
No. 38, 12<sup>th</sup> Main BTM I Stage, Bangalore,  
Karnataka, India.

Ph: +91-80-26689339, +91-9886338031

Email: [info@vinjey.com](mailto:info@vinjey.com)

Website: [www.vinjey.com](http://www.vinjey.com)

EFOS is a Real-Time Operating System (RTOS) designed with performance, ease of use and flexibility in mind. Typically many Real-time systems have stringent memory requirements and very high demand for efficiency. EFOS provides flexibility and ease of use for developer with features like multi-tasking, Inter-task communication at the same time meeting the memory and efficiency constraints. This document describes the Application Programming Interfaces available for all the standard modules in EFOS.

## Contents

Contents .....	4
1. INT APIs:.....	6
1.1 Int INT_intHook(Int intNumber, void *ep):.....	6
1.2 Int INT_disable(): .....	6
1.3.....	6
1.4 void INT_restore(Int oldState):.....	6
1.5 INT_enter:.....	7
1.6 INT_exit:.....	7
2. TASK APIs: .....	8
2.1 TASK_Handle TASK_create ( TASK_Attrs *taskAttrs ):.....	8
2.1.1 TASK_Attrs parameters: .....	8
2.2 TASK_Handle TASK_self():.....	9
2.3 void TASK_suspend(TASK_Handle taskHandle): .....	10
2.4 void TASK_resume(TASK_Handle taskHandle):.....	10
2.5 void TASK_disable():.....	10
2.6 void TASK_enable():.....	11
2.7 Int TASK_setPriority(TASK_Handle taskHandle, Int priority):.....	11
2.8 void TASK_yield(): .....	12
2.9 void TASK_sleep(Int sleepTime):.....	13
2.10 Int TASK_delete(TASK_Handle taskHandle) : .....	13
3. SEMA APIs: .....	14
3.1 SEMA_Handle SEMA_create(SEMA_Attrs *semaAttrs): .....	14
3.1.1 SEMA_Attrs Parameters:.....	14
3.2 Int SEMA_give(SEMA_Handle semaHandle):.....	15
3.3 Int SEMA_iGive(SEMA_Handle semaHandle):.....	15
3.4 Int SEMA_take(SEMA_Handle semaHandle, Int timeOut):.....	16
3.5 Int SEMA_count(SEMA_Handle semaHandle):.....	16
3.6 Int SEMA_delete(SEMA_Handle semaHandle): .....	16
4. PIPE APIs: .....	17
4.1 PIPE_Handle PIPE_create(PIPE_Attrs *pipeAttrs): .....	17
4.2 Int PIPE_ctrl(PIPE_Handle pipeHandle, Int cmd, void *arg):.....	18
4.3 UInt PIPE_read(PIPE_Handle pipeHandle, void *ptr, UInt length): .....	18
4.4 UInt PIPE_write(PIPE_Handle pipeHandle, void *ptr, UInt length):.....	19
4.5 PIPE_getFreeBytesCount(pipeHandle): .....	19
4.6 PIPE_getBytesAvailable(PIPE_Handle pipeHandle):.....	19

4.7 Int PIPE_close(PIPE_Handle pipeHandle):.....	19
5. MSG APIs:.....	20
5.1 MSG_Handle MSG_create ( MSG_Attrs *attrs ):.....	20
5.2 Int MSG_ctrl ( MSG_Handle msgHandle, Int cmd, void *arg ): .....	21
5.3 Int MSG_read ( MSG_Handle msgHandle, void *ptr ):.....	22
5.4 Int MSG_write ( void *ptr, MSG_Handle msgHandle ): .....	22
5.5 MSG_isMsgAvailable(msgHandle):.....	22
5.6 MSG_isMsgFull(msgHandle):.....	22
5.7 void MSG_close ( MSG_Handle msgHandle ): .....	23
6. Memory Manager APIs: .....	24
6.1 void * MM_alloc ( UInt16 segId, UInt16 size, UInt16 align ):.....	24
6.2 void MM_free ( UInt16 segId, void *ptr ):.....	24
6.3 void * Malloc ( UInt16 size ):.....	24
6.4 void Free ( void *ptr ):.....	24
7. Initialization APIs: .....	25
7.1 OS_init:.....	25
7.2 MM_init:.....	25
7.3 PIPE_init:.....	25
7.4 MSG_init: .....	25
7.5 OS_start: .....	26
8 Revision History .....	27

## 1. INT APIs:

This section describes about the INT APIs available in the EFOS. INT module is used for handling the interrupts in the system.

### 1.1 Int INT\_intHook(Int intNumber, void \*ep):

Each interrupt in the system will have a corresponding interrupt number. Each interrupt will have an address associated with it. Address represents where the entry point for the Interrupt Service Routine (ISR) is located. This API is used to change the entry point for a given interrupt number.

- intNumber will be the interrupt number to which we need to hook the entry point
- ep is the address of entry point to the Interrupt Service Routine (ISR)

INT\_intHook will return 1 on success and 0 on failure. Example usage of INT\_intHook is as given below

```
INT_intHook(11, hw_int9);
```

### 1.2 Int INT\_disable():

INT\_disable is used for disabling the interrupts. It returns a value that will hold the global interrupt status before disabling the interrupts. Example usage is

```
Int oldStatus;  
oldStatus = INT_disable();
```

### 1.3 void INT\_enable():

INT\_enable is used for enabling the interrupts. Example usage is

```
INT_enable();
```

### 1.4 void INT\_restore(Int oldState):

INT\_restore is used for restoring the global interrupt status to its earlier state. Example usage of INT\_restore is

```
Int oldStatus;  
oldStatus = INT_disable();  
....  
....  
INT_restore(oldStaus);
```

### **1.5 INT\_enter:**

INT\_enter is the assembly language macro should be used at the beginning of the Interrupt Service Routine (ISR). INT\_enter is used for saving the current context and switching from the task stack to the interrupt stack.

### **1.6 INT\_exit:**

INT\_exit is the assembly language macro should be used at the end of the Interrupt Service Routine (ISR). INT\_exit is used for restoring the context and switching from the interrupt stack to task stack.

## 2. TASK APIs:

This section gives information about the TASK APIs available in the EFOS.

### 2.1 TASK\_Handle TASK\_create(TASK\_Attrs \*taskAttrs):

TASK\_create is used for creation of new tasks in the system. TASK\_create takes attributes of the task as the argument for creation of a task. TASK\_create returns the handle to the task. It returns NULL in the event of failure. In EFOS scheduling of the task happens after the main() function exits. TASK\_create invokes scheduler, however the scheduling doesn't happen if TASK\_create is invoked before main() function exits.

```
typedef void (*TASK_EntryPt)(void *arg);

typedef struct __task_attrs
{
    TASK_EntryPt    entryPt;           /* Entry point for the Task */
    UInt            *stackPtr;        /* Stack Pointer for the Task */
    Int             priority;         /* Priority for the Task */
    TASK_Handle     objPtr;          /* Pointer to the task object */
    void            *arg;            /* Argument to the task */
#ifdef EFOS_DEBUG_LIST_ON
    char            *name;
#endif
}TASK_Attrs;
```

#### 2.1.1 TASK\_Attrs parameters:

TASK\_Attrs has following parameters

- entryPt:

TASK\_EntryPt is nothing but a function pointer that points to a function that takes a void pointer argument and returns none. Entry Point for a task is where the task begins its execution

```
typedef void (*TASK_EntryPt)(void *arg);

void task1(void *arg)
{
    /* task1 begins execution here */
    .....
    .....
}

TASK_Attrs taskAttrs;

taskAttrs.entryPt = task1;           /* Entry point for the task */
```

Consider we need to create a task which needs to start its execution from the beginning of function task1. In this case entry point for the task is task1

- **stackPtr:**

Each task needs to have its own stack. stackPtr holds the pointer to the stack. Consider the case wherein we need to create a stack of size 2000 bytes.

```
UInt16 task1Stack[1000];

TASK_Attrs taskAttrs;
taskAttrs.stackPtr=&(task1Stack[999]); /* Pointer to the task1 stack */
```

stackPtr will hold the initial value of the stack pointer register when the task gets created.

- **priority:**

Each task will have its associated priority value. Priority value determines the way the task scheduling happens.

- **objPtr:**

This is pointer to the task object that will be used by the task created.

- **arg**

This is a void pointer that will be passed as argument to the task.

- **name:**

When the Debug support is enabled, TASK\_Attrs has an additional argument name. name is pointer to a null-terminated character array that will hold the name of task. This information will be useful while debugging the applications. Size of the character array is restricted to a maximum size of 32 bytes.

## 2.2 TASK\_Handle TASK\_self():

TASK\_self returns the task handle of the current executing task. Typically tasks don't know their own handles. So a task can use TASK\_self() to get its own handle. TASK\_self() is useful for TASK APIs like TASK\_setPriority as described in section 2.7. Example usage of TASK\_self is

```
TASK_Handle myHandle;
myHandle = TASK_self();
```

## 2.3 void TASK\_suspend(TASK\_Handle taskHandle):

TASK\_suspend is used for suspending a task that is runnable. TASK\_suspend should be used only for suspending the execution of a Runnable Task.

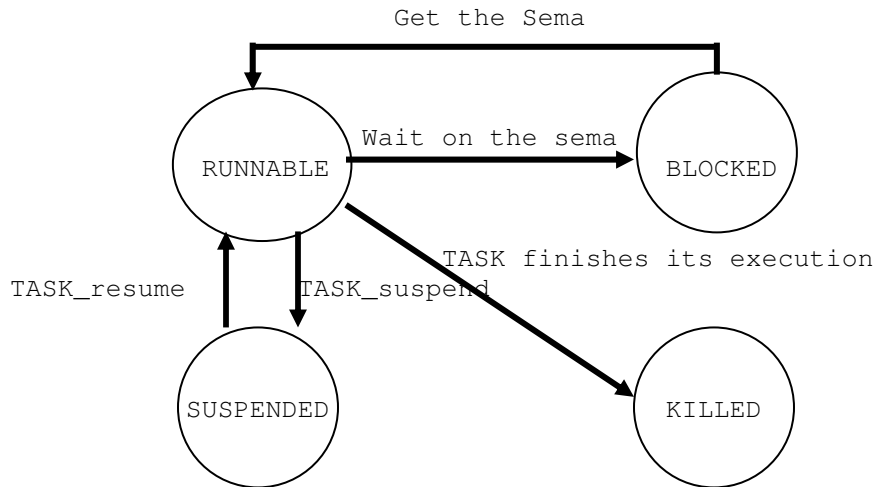


Figure above shows the states a task can be in. TASK\_suspend is used for changing state of task from Runnable to Suspended. When a task is in suspended state it won't get scheduled for execution. Example usage given here is used for a task to suspend itself

```
TASK_suspend(TASK_self());
```

## 2.4 void TASK\_resume(TASK\_Handle taskHandle):

TASK\_resume is used for making a task suspended using TASK\_suspend back to Runnable again. Figure above shows the states a task can be in. TASK\_resume is used for changing state of task from Suspended to Runnable. TASK\_resume should be used only on the tasks whose state is suspended. Example usage of TASK\_resume is

```
TASK_resume(taskHandle);
```

## 2.5 void TASK\_disable():

TASK\_disable is used to disable task scheduling in a region of the code. After TASK\_disable is executed task scheduling is disabled till it is enabled back. However Interrupt Service Routines will be executed when task scheduling is disabled. Example usage of TASK\_disable is

```
TASK_disable();
```

## 2.6 void TASK\_enable():

TASK\_enable is used for enabling back the task scheduling that has been disabled earlier. Task scheduling is enabled back after TASK\_enable is executed. TASK\_disable and TASK\_enable can be used as a pair to disable scheduling in a region of code. Example usage of TASK\_disable, TASK\_enable pair is as given below

```
TASK_disable();  
  
/* Region where task scheduling is disable */  
  
TASK_enable();
```

## 2.7 Int TASK\_setPriority(TASK\_Handle taskHandle, Int priority):

TASK\_setPriority is used for changing the priority of a given task. It takes input the task handle and the new integer priority level. Consider the following example that illustrates the usage of the TASK\_setPriority API.

```
void task1(void *arg)  
{  
    while ( 1 )  
    {  
        /* Display From Task1 */  
        TASK_setPriority(TASK_self(),1);  
    }  
}  
  
void task2(void *arg)  
{  
    while ( 1 )  
    {  
        /* Display From Task2 */  
        TASK_setPriority(task1Handle,3);  
    }  
}  
  
void main()  
{  
    /* Create task1 with priority value 3 and task2 with priority value 2 */  
}
```

When we execute this program we will get following messages.

```
From Task1  
From Task2  
From Task1  
From Task2
```

At beginning task1 starts execution as its priority value is higher compared to that task2. So we get the message "From Task1" in the beginning. However after printing the message task1 uses TASK\_setPriority to reduce its priority level. So task2 having priority value 2 starts its execution. So we get the message "From Task2". After that it restores

back the task1 priority to 3. So task1 resumes its execution. This process repeats so we get the pattern of the message given above.

## 2.8 void TASK\_yield():

When there is more than one task with equal priority, the task that has joined the runnable list first among them will get the CPU. However the task that gained control can use TASK\_yield API to give control to other equal priority tasks. Consider the following example that illustrates the usage of TASK\_yield API.

```
void task1(void *arg)
{
    while ( 1 )
    {
        printf("From Task1");
        TASK_yield();
    }
}

void task2(void *arg)
{
    while ( 1 )
    {
        DBG_puts("From Task2");
        TASK_yield();
    }
}

void main()
{
    /* Create Task1 and Task2 with priority value 2 */
}
```

When we execute this program we will get following messages.

```
From Task1
From Task2
From Task1
From Task2
```

When the scheduler executes for the first time it will find 2 tasks with same priority. However it will find that task1 has joined the runnable list first since it has got created ahead of task2. So task1 will begin its execution and we get message "From Task1". Then TASK\_yield function is called by the task1. So the control is switched to task2 and we get the message "From Task2". Since task2 also calls TASK\_yield, task1 resumes its execution. This process repeats so we get the above given message pattern.

## **2.9 void TASK\_sleep(Int sleepTime):**

TASK\_sleep can be used for blocking the current task for sleepTime amount of time. When the current task invokes TASK\_sleep its state changes from Runnable state to blocked state. After sleepTime gets passed task moves back from Blocked state to Runnable state.

## **2.10 Int TASK\_delete(TASK\_Handle taskHandle) :**

TASK\_delete call can be used for deleting a task at the run-time. TASK\_delete takes input the TASK\_Handle. EFOS always maintains a list of tasks available in the system. TASK\_delete can be used for removing a task from the list of tasks. TASK\_delete can be used to delete a task in any state. If a task wants to delete itself it can do it with as shown below

```
TASK_delete(TASK_self());
```

### 3. SEMA APIs:

This section gives information about the SEMA APIs available in the EFOS.

#### 3.1 SEMA\_Handle SEMA\_create(SEMA\_Attrs \*semaAttrs):

SEMA\_create is used for creating the semaphores. It returns the handle for the semaphore that is created. It returns NULL when the creation is un-successful.

##### 3.1.1 SEMA\_Attrs Parameters:

- count: Each semaphore is associated with a count value at any given time. The count value in the SEMA\_Attrs structure defines the initial value of the count.
- objPtr: This is pointer to the semaphore object that will be used by the semaphore created.
- name: When the Debug support is enabled, SEMA\_Attrs has an additional argument name. name is pointer to a null-terminated character array that will hold the name of semaphore. This information will be useful while debugging the applications. The size of the character array is restricted to a maximum size of 32 bytes.

```
typedef struct __sema_attrs
{
    Int                count;
    SEMA_Handle        objPtr;
#if ( EFOS_FEATURES_MASK & EFOS_DEBUG_LIST_MASK )
    char               *name;
#endif
} SEMA_Attrs;
```

Example usage of Semaphore creation is

```
SEMA_Handle semaHandle;
SEMA_Attrs semaAttrs;
SEMA_Obj semaObj;

semaAttrs.count = 1;
semaAttrs.objPtr = &semaObj;
#if ( EFOS_FEATURES_MASK & EFOS_DEBUG_LIST_MASK )
semaAttrs.name = "MY_SEMA";
#endif

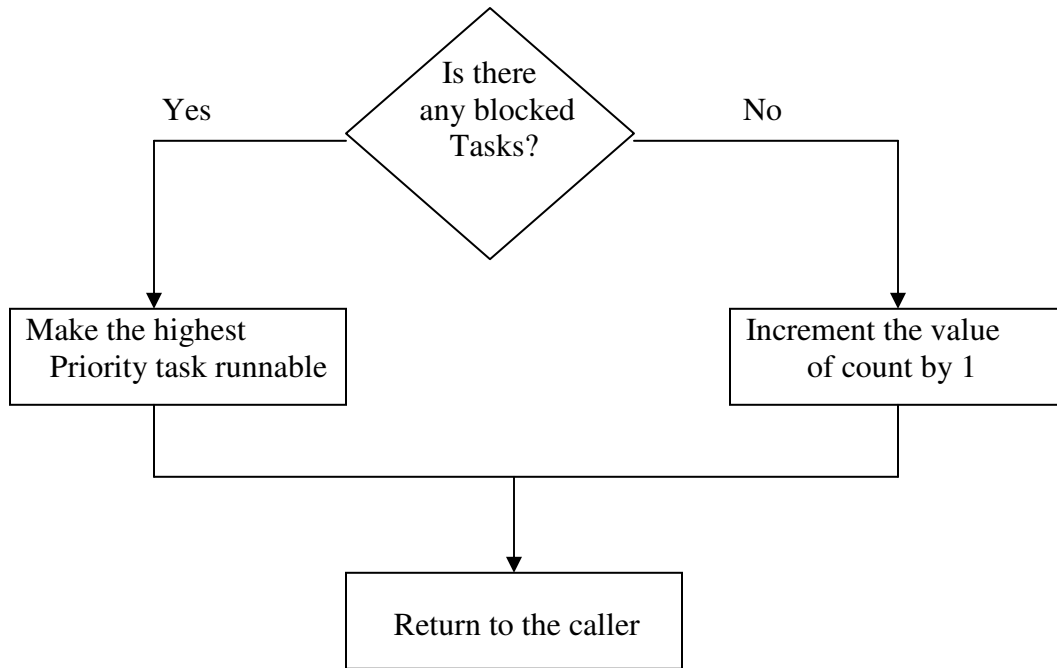
semaHandle = SEMA_create(&semaAttrs);
```

This example is used for creating a semaphore whose initial count value is 1 with name MY\_SEMA.

### 3.2 Int SEMA\_give(SEMA\_Handle semaHandle):

SEMA\_give checks whether there is any blocked tasks in the semaphore. If there is any blocked task in the semaphore it makes the highest priority blocked task runnable and returns to the caller. If there is no blocked task in the semaphore it simply increment the value of count by 1 and returns to the caller. Consider the case when there is more than one task that holds the highest priority. In that case the task among them that has joined the blocked list first will be made runnable. SEMA\_give returns 1 on success and 0 on failure. Example usage of SEMA\_give is

```
ret_value = SEMA_give(semaHandle);
```



### 3.3 Int SEMA\_iGive(SEMA\_Handle semaHandle):

SEMA\_iGive is similar to that of SEMA\_give. The difference is that SEMA\_give is used in the application context and SEMA\_iGive is used in the ISR context. It is important to note that SEMA\_give is should not be used in the ISR context and SEMA\_iGive should not r be used in the application context. Internally the difference between SEMA\_give and SEMA\_iGive is that SEMA\_give invokes scheduler immediately, whereas SEMA\_iGive invokes the scheduler when the ISR's execution is completed. SEMA\_iGive returns 1 on success and 0 on failure. Example usage of SEMA\_iGive is

```
ret_value = SEMA_iGive(semaHandle);
```

### **3.4 Int SEMA\_take(SEMA\_Handle semaHandle, Int timeOut):**

SEMA\_take decrements semaphore count value by 1 if the current count value is not 0. If count is 0, the task that invokes the SEMA\_take gets blocked. In that case the task is said to be blocked on the semaphore. timeOut denotes the amount of time that the task can remain blocked in the semaphore at maximum. Once the time specified expires SEMA\_take returns 0 to denote time-out has occurred. When the timeOut value is -1, it represents Infinite time out. Example usage of SEMA\_take is

```
result = SEMA_take(semaHandle,-1);
```

### **3.5 Int SEMA\_count(SEMA\_Handle semaHandle):**

SEMA\_count is used for determining the count value of the semaphore. SEMA\_count can be used to determine whether the subsequent SEMA\_take will block or not.

### **3.6 Int SEMA\_delete(SEMA\_Handle semaHandle):**

SEMA\_delete is used for deletion of semaphore that has been created earlier. It takes input the semaphore handle that has been returned as result of SEMA\_create. SEMA\_delete has to be used with caution. Deletion of a semaphore on which some tasks are blocked can lead to disaster. SEMA\_delete returns 1 on success and 0 on failure.

## 4. PIPE APIs:

### 4.1 PIPE\_Handle PIPE\_create(PIPE\_Attrs \*pipeAttrs):

PIPE\_create is used for creation of pipe with specified attributes. PIPE\_create returns the handle to the PIPE Object. It returns NULL in the event of failure. PIPE\_create takes input pointer to the PIPE\_Attrs structure that contains the attributes to the pipe.

```
typedef struct __pipe_attrs
{
    UInt          pipeLength;          /* Length of the pipe */
    PIPE_Handle   objPtr;              /* Pointer to the PIPE Object */
    void          *bufPtr;             /* Pointer to the PIPE Buffer */
    Int           isRdBlockingCall;    /* Read is blocking call (-Or-) not */
    Int           isWrBlockingCall;    /* Write is blocking call (-Or-) not */
#ifdef EFOS_FEATURES_MASK & EFOS_DEBUG_LIST_MASK
    char          *name;               /* Name of the Pipe */
#endif
} PIPE_Attrs;
```

- pipeLength is used to define the length of the pipe in bytes.
- isRdBlockingCall is used to define whether the PIPE\_read call is blocking (-Or-) non-blocking.
- isWrBlockingCall is used to define whether the PIPE\_write call is blocking (-Or-) non-blocking.
- objPtr is the pointer to the pipe object that the created pipe will use.
- bufPtr is the pointer to the buffer that will be used by the pipe. PIPE will use the buffer starting with address bufPtr for pipeLength bytes as it buffer.
- name: When the Debug support is enabled, PIPE\_Attrs has an additional argument name. name is pointer to a null-terminated character array that will hold the name of pipe. This information will be useful while debugging the applications. The size of the character array is restricted to a maximum size of 32 bytes.

Example usage of PIPE\_create is given below

```
PIPE_Attrs pipeAttrs;
PIPE_Handle pipeHandle;
PIPE_Obj pipeObj;
unsigned char pipeBuf[1000];

pipeAttrs.pipeLength = 1000;
pipeAttrs.objPtr = &pipeObj;
pipeAttrs.bufPtr = pipeBuf;
pipeAttrs.isRdBlockingCall = 1;
pipeAttrs.isWrBlockingCall = 0;
#ifdef EFOS_FEATURES_MASK & EFOS_DEBUG_LIST_MASK
pipeAttrs.name = "MY_PIPE"
#endif

pipeHandle = PIPE_create(&pipeAttrs);
```

## 4.2 Int PIPE\_ctrl(PIPE\_Handle pipeHandle, Int cmd, void \*arg):

PIPE\_ctrl is used for controlling the behavior of the pipe. PIPE\_ctrl takes input pipeHandle that is the return value of the PIPE\_create, command and command argument as its arguments. PIPE\_ctrl takes input the following commands

```
#define PIPE_CMD_RD_BLOCKING           1
#define PIPE_CMD_RD_NON_BLOCKING      2
#define PIPE_CMD_WR_BLOCKING         3
#define PIPE_CMD_WR_NON_BLOCKING     4
```

PIPE\_CMD\_RD\_BLOCKING command doesn't take any arguments. It is used for changing the PIPE\_read call blocking.

PIPE\_CMD\_RD\_NON\_BLOCKING command doesn't take any arguments. It is used for changing the PIPE\_read call non-blocking.

PIPE\_CMD\_WR\_BLOCKING command doesn't take any arguments. It is used for changing the PIPE\_write call blocking.

PIPE\_CMD\_WR\_NON\_BLOCKING command doesn't take any arguments. It is used for changing the PIPE\_write call non-blocking.

PIPE\_ctrl command should not be used when PIPE\_read (-Or-) PIPE\_write call is executing.

## 4.3 UInt PIPE\_read(PIPE\_Handle pipeHandle, void \*ptr, UInt length):

PIPE\_read is used for reading the data from the pipe. It takes input the pipeHandle that is the return value of the PIPE\_create, pointer to the input buffer and number of bytes to be read. PIPE\_read can be either blocking (-Or-) non-blocking depending upon the attributes supplied during the pipe creation. This attribute can be changed using PIPE\_ctrl call at later point of time. If PIPE\_read is blocking call it will return only when the requested number of bytes is read successfully. If PIPE\_read is non-blocking call it is not guaranteed that the requested number of bytes will be read. PIPE\_read returns the number of bytes read onto the input buffer. In case of blocking PIPE\_read one can use PIPE\_getBytesAvailable to determine the number of bytes available in the pipe to ensure that subsequent PIPE\_read will not block. Example usage of PIPE\_read is

```
bytesRead = PIPE_read(pipeHandle, inputBuf, 256);
```

In case PIPE\_read is a blocking call it can be used for reading data more than the size of the pipe.

#### **4.4 UInt PIPE\_write(PIPE\_Handle pipeHandle, void \*ptr, UInt length):**

PIPE\_write is used for writing the data to the pipe. It takes input the pipeHandle that is the return value of the PIPE\_create, pointer to the output buffer and number of bytes to be written. PIPE\_write can be either blocking (-Or-) non-blocking depending upon the attributes supplied during the pipe creation. This attribute can be changed using PIPE\_ctrl call at later point of time. If PIPE\_write is blocking call it will return only after all the data is being successfully written. If PIPE\_write is non-blocking call it is not guaranteed that all the output data will be written. PIPE\_write returns the number of bytes written onto the pipe. In case of blocking PIPE\_write one can use PIPE\_getFreeBytesCount to determine the number of bytes that can be written without getting blocked. Example usage of PIPE\_write is

```
bytesWritten = PIPE_write(pipeHandle, outputBuf, 256);
```

In case PIPE\_write is a blocking call it can be used for writing data more than the size of the pipe.

#### **4.5 PIPE\_getFreeBytesCount(pipeHandle):**

This is a macro that can be used to determine the amount of free space that is available in the Pipe. Example usage of PIPE\_getFreeBytesCount is

```
bytesFree = PIPE_getFreeBytesCount(pipeHandle);
```

#### **4.6 PIPE\_getBytesAvailable(PIPE\_Handle pipeHandle):**

This is a macro that can be used to determine the amount of data available in the Pipe. Example usage of PIPE\_getBytesAvailable is

```
bytesAvailable = PIPE_getBytesAvailable(pipeHandle);
```

#### **4.7 Int PIPE\_close(PIPE\_Handle pipeHandle):**

PIPE\_close is used for closing the pipe that has been already created. PIPE\_close takes input the handle to the pipe that needs to be closed. Example usage of PIPE\_close is

```
PIPE_close(pipeHandle);
```

## 5. MSG APIs:

### 5.1 MSG\_Handle MSG\_create ( MSG\_Attrs \*attrs ):

MSG\_create is used for creation of MSG object with specified attributes. MSG\_create returns the handle to the MSG object. It returns NULL in the event of failure.

```
typedef struct __msg_attrs
{
    Int      msgSize;          /* Size of the Message */
    Int      msgCount;        /* Number of Messages */
    Int      rdBlocking;      /* whether MSG_read call is blocking call */
    Int      wrBlocking;      /* whether MSG_write call is blocking call */
    Int      rdTimeOut;       /* Time-out when MSG_read is blocking call */
    Int      wrTimeOut;       /* Time-out when MSG_write is blocking call */
    MSG_Handle objPtr;
    Void     *bufPtr;
#ifdef EFOS_FEATURES_MASK & EFOS_DEBUG_LIST_MASK
    Int8     *name;
#endif
} MSG_Attrs;
```

Elements of the MSG\_Attrs structure is given below

- msgSize: Message queue is used for sending and receive message of a particular size. This attribute defines the size of each message.
- msgCount: This attributes defines the maximum number of message that a message queue buffer can hold at a given instance of time.
- rdBlocking flag is used to denote whether the MSG\_read call is blocking (-Or-) non-blocking.
- wrBlocking is used to define whether the MSG\_write call is blocking (-Or-) non-blocking.
- objPtr: This is pointer to the MSG Object that will be used by the message queue created.
- bufPtr: This is pointer to the buffer that will be used by the message queue. Message queue will use the buffer starting with address bufPtr with length of for ( msgSize \* msgCount ) bytes for storing the messages
- rdTimeOut is the maximum amount of time MSG\_read will block for a message when MSG\_read is a blocking call. If value of rdTimeOut is -1 it represents infinite timeout.
- wrTimeOut is the maximum amount of time MSG\_write will block to write a message when MSG\_write is a blocking call. If value of wrTimeOut is -1 it represents infinite timeout.
- name: When the Debug support is enabled, MSG\_Attrs has an additional argument name. name is pointer to a null-terminated character array that will hold

the name of message queue. This information will be useful while debugging applications. The size of the character array is restricted to a maximum size of 32 bytes.

```
MSG_Attrs msgAttrs;
MSG_Obj msgObj;
MSG_Handle msgHandle;
unsigned char msgBuf[64];

msgAttrs.msgSize = 16;
msgAttrs.msgCount = 4;
msgAttrs.rdBlocking = 1;
msgAttrs.wrBlocking = 1;
msgAttrs.objPtr = &msgObj;
msgAttrs.bufPtr = msgBuf;
msgAttrs.rdTimeOut = -1;
msgAttrs.wrTimeOut = -1;
msgAttrs.msgName = "MY_MSG";

msgHandle = MSG_create(&msgAttrs);
```

## 5.2 Int MSG\_ctrl ( MSG\_Handle msgHandle, Int cmd, void \*arg ):

MSG\_ctrl is used for controlling the behavior of the message queue. MSG\_ctrl takes input msgHandle that is the return value of the MSG\_create, command and command argument as its arguments. MSG\_ctrl takes input the following commands

```
#define MSG_CMD_RD_BLOCKING          1
#define MSG_CMD_RD_NON_BLOCKING     2
#define MSG_CMD_WR_BLOCKING         3
#define MSG_CMD_WR_NON_BLOCKING     4
#define MSG_CMD_RD_TIME_OUT        5
#define MSG_CMD_WR_TIME_OUT        6
```

MSG\_CMD\_RD\_BLOCKING command doesn't take any arguments. It is used for changing the MSG\_read call blocking.

MSG\_CMD\_RD\_NON\_BLOCKING command doesn't take any arguments. It is used for changing the MSG\_read call non-blocking.

MSG\_CMD\_WR\_BLOCKING command doesn't take any arguments. It is used for changing the MSG\_write call blocking.

MSG\_CMD\_WR\_NON\_BLOCKING command doesn't take any arguments. It is used for changing the MSG\_write call non-blocking.

MSG\_CMD\_RD\_TIME\_OUT command doesn't take any arguments. It is used for changing the value of MSG\_read time out.

MSG\_CMD\_WR\_TIME\_OUT command doesn't take any arguments. It is used for changing the value of the MSG\_write time out.

MSG\_ctrl command should not be used when MSG\_read (-Or-) MSG\_write call is executing.

### **5.3 Int MSG\_read ( MSG\_Handle msgHandle, void \*ptr ):**

MSG\_read is used for reading a message from the given message queue. It takes pointer to input buffer, where the message has to be stored and the message handle as parameters. MSG\_read can be blocking (-Or-) non-blocking call depending on the attributes supplied during the MSG creation. This attribute can be changed using MSG\_ctrl call at later point of time. If MSG\_read is blocking call it will return only when the message is read successfully (-Or-) if time out happens. If MSG\_read is non-blocking call it is not guaranteed that the message will be read. MSG\_read returns 1 on reading the message onto input buffer. It returns 0 on failure to read the message. When MSG\_read is blocking call it will block when there is no message in the message queue for the specified time out value. One can use MSG\_isMsgAvailable to determine whether any message exists in the message queue to ensure that the subsequent MSG\_read call won't block. Example usage of MSG\_read is

```
MSG_read ( inBuf, msgHandle );
```

### **5.4 Int MSG\_write ( void \*ptr, MSG\_Handle msgHandle ):**

MSG\_write is used for writing a message to the given message queue. It takes input the MSG\_Handle and the output buffer, which holds the message. MSG\_write can be blocking (-Or-) non-blocking call depending on the attributes supplied during the MSG creation. This attribute can be changed using the MSG\_ctrl call at later point of time. If MSG\_write is blocking call it will return only after the message is successfully written (-Or-) if time out occurs. If MSG\_write is non-blocking call it is not guaranteed that message will be written. MSG\_write returns 1 on writing the message onto Message Queue. It returns 0 on failure. When MSG\_write is a blocking call it will block only when the Message queue is full for the specified timeout value. One can use MSG\_isMsgFull to determine whether the message queue is not full to ensure that subsequent MSG\_write call won't block. Example usage of MSG\_write is

```
MSG_write ( outBuf, msgHandle );
```

### **5.5 MSG\_isMsgAvailable(msgHandle):**

This is a macro that can be used to determine whether there is any message available in the message queue (-Or-) not. Example usage of MSG\_isMsgAvailable is

```
msgAvailable = MSG_isMsgAvailable ( msgHandle );
```

### **5.6 MSG\_isMsgFull(msgHandle):**

This is a macro that can be used to determine whether the message queue is full.  
Example usage of MSG\_isMsgFull is

```
isMsgQFull = MSG_isMsgFull ( msgHandle );
```

### **5.7 void MSG\_close ( MSG\_Handle msgHandle ):**

MSG\_close is used for deleting the message object that has been created earlier.  
Example usage of MSG\_close is

```
MSG_close ( msgHandle );
```

## **6. Memory Manager APIs:**

This section gives information regarding the APIs relating to the memory manager module in EFOS.

### **6.1 void \* MM\_alloc ( UInt16 segId, UInt16 size, UInt16 align ):**

MM\_alloc is used for allocation of a memory of specified size, alignment on a given input segment. MM\_alloc returns NULL on failure and returns the pointer to the memory on success. segId is the id that represents the memory segment. size represents the size of the memory requested. align represents the alignment requested. Example usage of MM\_alloc is

```
void *ptr;
/* Allocate 100 bytes of alignment 4 in segment with id 0 */
ptr = MM_alloc ( 0, 100, 4 );
```

### **6.2 void MM\_free ( UInt16 segId, void \*ptr ):**

MM\_free is used for freeing up the memory that has been allocated earlier. MM\_free takes input the segment id and the pointer returned by the MM\_alloc. It is used for freeing up the memory that has been allocated earlier using MM\_alloc. Example usage of MM\_free is

```
MM_free ( 0, ptr );
```

### **6.3 void \* Malloc ( UInt16 size ):**

Memory manager holds the default memory segment for allocation of memory for Malloc and Free calls. Malloc is used for allocation of memory from the default segment of specified input size. Malloc returns NULL when the allocation has failed and returns the pointer to memory on success. An alignment of 8 bytes is use for Malloc calls. Example usage of Malloc is

```
void *ptr;
ptr = Malloc ( 100 );
```

### **6.4 void Free ( void \*ptr ):**

Memory manager holds the default memory segment for allocation of memory for Malloc and Free calls. Free takes input the pointer returned by Malloc earlier. Free is used for freeing up the memory allocated by the Malloc earlier. Example usage of Free is

```
Free ( ptr );
```

## **7. Initialization APIs:**

These are the set of initialization APIs that get invoked in the when OS is initialized. However Initialization APIs for the Add-on modules has to be invoked by the user.

### **7.1 OS\_init:**

OS\_init is used to initialize all the modules in the kernel library of the EFOS. OS\_init has to be invoked before any EFOS kernel API is invoked. Example usage of OS\_init is

```
OS_init();
```

### **7.2 MM\_init:**

MM\_init is used to initialize the memory manager module in the memory manager library. MM\_init has to be invoked before any memory manager API is invoked. Example usage of MM\_init is

```
MM_init();
```

### **7.3 PIPE\_init:**

PIPE\_init is used to initialize the pipe module in the pipe library. PIPE\_init has to be invoked before pipe API is invoked. Example usage of PIPE\_init is

```
PIPE_init();
```

### **7.4 MSG\_init:**

MSG\_init is used to initialize the message queue module in the message queue library. MSG\_init has to be invoked before message queue API is invoked. Example usage of MSG\_init is

```
MSG_init();
```

## 7.5 OS\_start:

This is the last stage before application begins its execution. Roles of OS\_start is as given below

- Create a Low-Priority System Task

OS\_start creates a low-priority system task that is used by the EFOS. Priority level 0 is used by this low-priority system task and reserved for its usage. No application Task should use the priority level 0.

- Begin the execution application Tasks

All the Application Tasks won't begin its execution till the end of OS\_start. After creating the Low-Priority System Task OS\_init begins the execution of Application Tasks that are created in the main.

Application Tasks take over the control after this step.

## ***8 Revision History***

Revision Number	Comments	Date
1.0	First Version of the document	April 27, 2004