

EFOS User 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, 12th Main BTM I Stage, Bangalore,
Karnataka, India.

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

Email: info@vinjey.com

Website: 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.

Ease of Use:

EFOS is designed with simplicity in mind. It comes with a good number of examples that demonstrates the usage of different standard modules of EFOS. The APIs of EFOS are designed to be simple to reduce the amount of learning time for the developer. EFOS comes with a Graphical User Interface (GUI) that can be used for creating objects of different EFOS modules statically at the beginning of the program.

Flexible:

EFOS is application morphic. Each Real time system design requires certain features from the Real Time Operating System it uses. It is also quite evident that needs of each design is different from that of others. Also the requirements from the RTOS are different at different stages of product development life cycle. EFOS is flexible and it is configurable to the needs of the design.

This document gives an overview of different modules in the EFOS. It gives user level perspective how different modules of EFOS can be used

Contents

1. Interrupts:	7
1.1 Use of Interrupts:	7
1.1.1 I/O Devices Needs attention:	7
1.1.2 Increase processor efficiency:	7
1.2 Interrupt Stack:	8
1.3 Nested Interrupts:	8
1.4 How to write an ISR in EFOS:	9
1.5 Writing ISR in C Language:	9
1.6 List of EFOS APIs that can be invoked from an ISR:	10
2. Tasks:	11
2.1 TASK Basics:	11
2.1.1 Increase processor efficiency	11
2.1.2 Predictable Behavior	12
2.2 EFOS TASK:	12
2.2.1 Entry Point:	12
2.2.2 Task Stack and Context:	12
2.3 Task States:	13
2.4 Task Priority:	14

3. Semaphores:	15
3.1 Semaphores for Implementing Critical Sections:	15
3.2 Deadlocks:.....	17
3.3 Semaphores for Implementing Resource management:	18
3.4 Semaphores for Implementing Event Notification:	19
3.5 Semaphores and Task Linkage:	19
4. Memory Manager:	20
5. EFOS PIPE.....	21
5.1 Blocking Versus Non-Blocking PIPE_read:.....	21
5.2 Blocking Versus Non-Blocking PIPE_write:	22
6. EFOS MSG:	23
6.1 Blocking versus Non-blocking MSG_read:.....	23
6.2 Blocking versus Non-blocking MSG_write:	24

1. Interrupts:

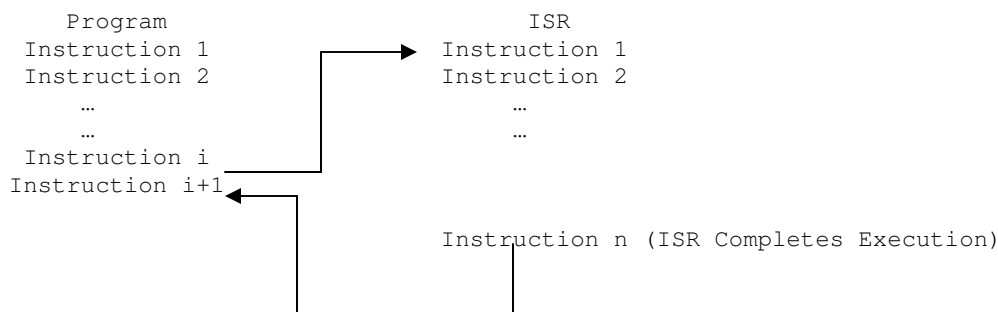
The concept of interrupts has been introduced in processors to increase the efficiency of the processor. As such when interrupts are not active the processor will continue to execute in a straight forward fashion. It involves fetching and execution of the instructions.

1.1 Use of Interrupts:

However in a system apart from CPU, multiple I/O devices will be used. Typically I/O devices will be slower in speed compared to that of processor. In this case interrupts in a system serves two purposes.

1.1.1 I/O Devices Needs attention:

When an I/O device needs attention it generates interrupt, processor will stop the current execution and start executing the Interrupt Service Routine (ISR). ISR will have the code to handle the I/O device. Once ISR is completed processor will resume execution from where it stopped before.



1.1.2 Increase processor efficiency:

With the use of interrupts, processing now needs to devote time to the I/O device only when there is a need. On other valuable time it can do lots of useful processing. Thus interrupts in the system increase the system efficiency

Apart from I/O Interrupts processor can get Timer Interrupts, which happens in a periodic fashion

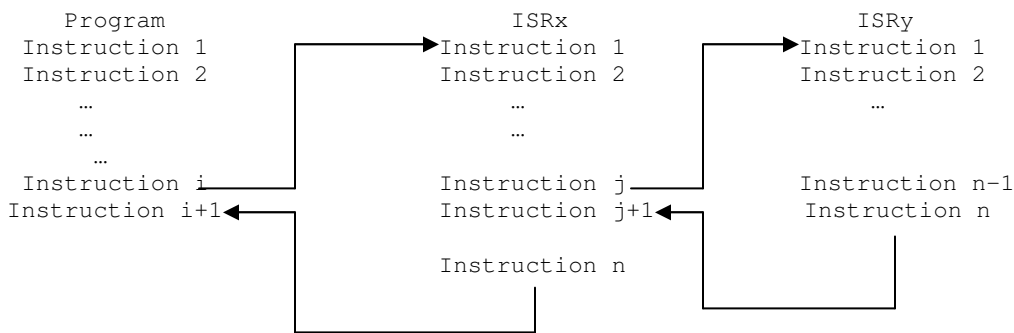
1.2 Interrupt Stack:

Interrupt stack is the stack that will be used by the interrupts. In a RTOS interrupts can use either separate stack (-Or-) use the stack of the task that got interrupted. In a system it is not deterministic to know which task will be interrupted. So if the task stack is used by the interrupt, size of all task's stack has to be incremented with the size of interrupt stack. Considering this wastage of memory, EFOS uses separate stack for interrupts. All the interrupts use one common stack. To understand how using separate interrupt stack saves memory, consider the following case. We have 5 tasks in the system and requirement for interrupt stack is 500 bytes. In that case we have clear advantage of 2000 bytes when we have separate interrupt stack

Using TASK Stack	Using Separate Interrupt Stack
Task1 Stack + 500 bytes for ISR Stack	Task1 Stack
Task2 Stack + 500 bytes for ISR Stack	Task2 Stack
Task3 Stack + 500 bytes for ISR Stack	Task3 Stack
Task4 Stack + 500 bytes for ISR Stack	Task4 Stack
Task5 Stack + 500 bytes for ISR Stack	Task5 Stack
	500 bytes for ISR Stack
<hr/> All Task Stack + 2500 bytes for ISR Stack	<hr/> All Task Stack + 500 bytes

1.3 Nested Interrupts:

Consider the case when Interrupt X has happened. Now the context will be saved and ISR_x will start its execution. When ISR_x is in the midway of its execution another interrupt Y happens. If Interrupt Y and global interrupt status is enabled ISR_y will interrupt ISR_x and start its execution. This concept is known as nested interrupts. Value of X can be same that of Y. This happens when same interrupt happens twice in a short duration.



Nested Interrupt can be avoided if ISR_x disables global interrupt. ISR_x can also choose to enable only selected set of interrupts. Nested Interrupts has both advantage and disadvantage. Advantage of using nested interrupt is it will reduce the interrupt latency time. Interrupt latency time is the difference between the time in which the interrupt happens and time in which ISR begins its execution. Say ISR_x doesn't support nesting of

interrupts then ISR_y will have to wait till ISR_x completes its execution there by increasing the interrupt latency. Thus nested interrupts helps in reducing the interrupt latency.

The disadvantage of nested interrupt is that the interrupt stack size has to be designed assuming the worst case. All the interrupts use one common stack. In order to support the nested interrupts the interrupt stack size has to be determined depending on the worst case nesting.

In EFOS each Interrupt Service Routine can choose to support nested interrupt or not in them. This provides the flexibility to the system designer to design his system depending upon his needs.

1.4 How to write an ISR in EFOS:

Interrupt Service Routine can be written in C (-Or-) Assembly Language in EFOS. For writing Interrupt Service Routine in C refer to the next section. EFOS provides two assembly language macros INT_enter and INT_exit that has to be used as header and footer for Interrupt service routines in EFOS. These macros do two important things

- Save and Restore the current context
- Switching back and forth from Task Stack to Interrupt Stack

A standard EFOS Interrupt Service Routine looks like following

```
_hw_int10:  
    INT_enter  
    ; Enable the Interrupts that can interrupt me  
    ; Enable the Global Interrupt Flag  
    ; Interrupt code  
    INT_exit
```

Enabling the interrupts that can interrupt the ISR and global interrupt flag is optional. It has to be done if the ISR decides to support nesting of interrupts.

1.5 Writing ISR in C Language:

Writing the Interrupt Service Routine in C is fairly straight forward. ISR Assembly language code has to invoke the required C function according to the calling conventions followed in your platform. Refer to the Getting Started Guide for your platform for information regarding this.

```
_hw_int10:  
    INT_enter  
    ; Enable the Interrupts that can interrupt me  
    ; Enable the Global Interrupt Flag  
    ; Invoke C Function according to C Language calling conventions  
    INT_exit
```

1.6 List of EFOS APIs that can be invoked from an ISR:

EFOS provide rich set of APIs for doing different operations with the kernel. However only certain set of EFOS APIs can be invoked from an ISR. Given below is the list of APIs that should not be invoked from an ISR

```
TASK_create  
TASK_self  
TASK_yield  
SEMA_take  
SEMA_give  
SEMA_create  
MSG_create  
MSG_read, MSG_write (If blocking)  
PIPE_create  
PIPE_read, PIPE_write (If blocking)
```

2. Tasks:

2.1 TASK Basics:

As we had seen earlier interrupts are used for increasing the processor efficiency. Typically Tasks are used in the system for two purposes

- a. Getting maximum efficiency out of the processor
- b. Getting predictable behavior in the system for a given application

2.1.1 Increase processor efficiency

A processor is designed to fetch and execution in a sequential fashion. When an interrupt happens it suspends its execution and executes the Interrupt service routine. Once the interrupt service routine is completed it resumes executions from where it got suspended earlier. Typically a register like program counter (PC) will be used for holding the information regarding the instruction to be executed. For a system with just one task, the execution involves sequence of instructions within that task like given below.

```
void taskA()
{
    getKey();
    ...
    ...
}

void getKey()
{
    /* Wait for the Key press
    and get the key data */
    ...
    ...
}
```

Now consider a case wherein getKey function has to wait for an I/O event to occur. In our case getKey waits for a key press in the keypad (-or-) keyboard. The moment user presses the key, keyboard interrupt will be generated and getKey can get the data regarding the key pressed. However till the user presses key taskA can't do any processing. Here is where multi-tasking comes into picture. It allows other tasks similar to taskA to be executed while taskA is waiting for a key press. Thus it helps in getting more efficiency out of the processor.

```
void taskA()
{
    getKey();
    ...
    ...
}

void getKey()
{
    /* Wait for the Key pres
    and get the key data */
    ...
    ...
}

void taskB()
{
    while ( 1 )
    {
        do X
        do Y
        ...
    }
}
```

2.1.2 Predictable Behavior

Now consider the scenario where in we have to process the user key press as soon as possible. But at the same time we have to increase the efficiency of the processor. Here comes the key concept of EFOS, priority. What we need here is when the taskA is waiting for key press taskB can do its processing. However when the key is pressed immediately taskA has to continue its execution. This scenario can be achieved by simply keeping the taskA priority value about that of taskB priority value. This way we can achieve both predictable behavior and achieve higher efficiency from the processor.

This kind of predictable behavior is very important in the real-time systems. Consider the following scenario. A Reactor is controlled by many tasks in the system. A panic button is designed in the system so that as soon as the panic button is pressed the reactor has to shut off. This can be achieved by creating a task which holds the highest priority in the system. The functionality of the task is to wait for the panic button press and shut down the reactor.

2.2 EFOS TASK:

Tasks and semaphore forms the basic building block of the EFOS. Each task will have its own entry point, stack and context. At any given point of time a task will be having one of following four states

```
TASK_RUNNABLE  
TASK_BLOCKED  
TASK_SUSPENDED  
TASK_DEAD
```

Each task in EFOS has an associated priority with it. EFOS schedules the execution of tasks depending upon the priority. Now let us look at each of them in a greater detail

2.2.1 Entry Point:

Entry point is defined as the place where the task has to begin its execution. Entry point is nothing but a function pointer which takes a void argument and returns none

```
void (*TASK_entryPt)(void *arg);
```

2.2.2 Task Stack and Context:

Stack is the space where the local variables are stored. It is important that each task has its own stack so that when a task gets preempted and resumes execution it can start from where it left before in the same state.

```

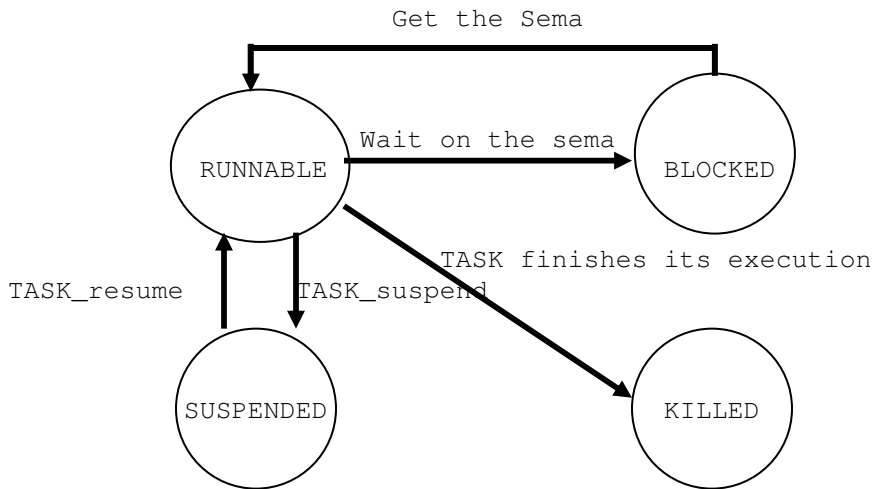
void taskA()      void fn1()      void fn2()      void fn3()
{
  ...
  fn1 ();
  ...
}
{
  ...
  fn2 ();
  ...
}
{
  ...
  fn3 ();
  ...
}

```

... → taskA Preempted

Consider the following case with a task, taskA whose entry point is function taskA. When the function fn3 is executing taskA gets blocked (-or-) preempted. So some other task will take over the processor and starts execution. However the local variables of the functions taskA, fn1, fn2 and fn3 will be stored its stack. When taskA resumes execution it has to retain the same values of the local variables. So each task has to have its own stack and context.

2.3 Task States:



A task can only be in one of the following four states. A task is in RUNNABLE state when it is ready to run. However in a system there can be more than one task that will be in the RUNNABLE state. Only one among them will use the processor at a given instance of time. The task that will use the processor will be defined by the priority of the task. A Task is said to be in Blocked state when it is waiting for a resource (-Or-) event to happen. For instance when getKey is waiting for a key press, it is essentially waiting for the event key press to happen. When the task acquires the resource (-Or-) the event happens it comes back to the RUNNABLE state.

A Runnable task can be suspended temporarily and kept in the SUSPENDED state. On other hand a suspended task can be moved back to the Runnable state at any time. TASK_suspend is used for moving a task from Runnable state to Suspended state. TASK_resume is used for restoring the suspended task back to the runnable state.

A Task moves to the DEAD State when it finishes the execution. DEAD State is the terminal condition for a task.

2.4 Task Priority:

Each task has an associated priority attached with it. Task priority value is nothing but an integer number. A task is said to have a higher priority when the integer priority value is higher. For instance consider two tasks taskA and taskB with priority values 3 and 2 respectively. In this case taskA is said to have higher priority compared to that of taskB. Priority is the key concept in defining the predictable behavior of the system and allocation of CPU among multiple tasks in RUNNABLE state. A task with highest priority among the runnable task will get the CPU. Consider the case when there are many tasks with highest priority. In that case one among them that has joined the runnable list first will get the CPU.

When there is more than one task that is in the runnable list first among them will be given the CPU. However a task can yield to give the control to another equal priority task. Also task priority can be changed dynamically at the run-time.

3. Semaphores:

Semaphores are used for the synchronization purposes in the system. Along with the tasks, semaphores forms critical component in the EFOS.

A Semaphore is initialized with an initial value of the count at the time of creation. Apart from the creation API semaphore module provides three other important APIs.

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.

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_iGive is similar to that of SEMA_give but will be used in the ISR context.

Semaphores are typically used for implementing three features in the application

- a. Critical Sections
- b. Resource management
- c. Event notifications.

In the following sections describe about how semaphores are used for implementing above mentioned features

3.1 Semaphores for Implementing Critical Sections:

Before we get into how semaphores are used for implementing the critical sections let us see what does critical section means.

```

    Int16 sum;

    void taskA()
    {
1   |   Int16 n, taskASum;
    |   ...
    |   taskASum = getSum(5);
    |   /* Print the sum value */
7   |   ...
    |   }

    void taskB()
3   |   {
    |   ...
    |   taskBSum = getSum(10);
    |   ...
5   |   ...
    |   }

    void getSum(Int16 n)
    {
    |   sum = 0;
    |   for ( i = 0 ; i < n ; i++ )
    |   {
    |       sum+=i;
    |   }
    |   }

```

In the above example two tasks exists with priority values 2 and 3 respectively. Shared global variable sum is used by both tasks. Consider the case when following sequence happens

- (i) taskA begins execution because taskB is blocked. It invokes getSum with argument 5.
- (ii) Few iterations of getSum are executed for taskA, during this time taskB become RUNNABLE and preempt taskA.
- (iii) taskB resumes execution and it invokes getSum with argument 10.
- (iv) getSum returns the value of sum as 55 and taskB continues execution
- (v) After sometime taskB gets blocked
- (vi) taskA resumes the execution, but unfortunately the value of sum has already been corrupted.
- (vii) The value that taskA will get as sum will be incorrect

Astute reader would have noted that sum can be used as a local variable to solve this problem. This example is used to present a simple example that illustrates the need of critical section.

Critical section is a section of code in which only one task should execute at a given point of time. In the above code it is evident that the code inside the getSum is critical section code. When getSum uses semaphore for implementing critical sections it can be re-written as given below

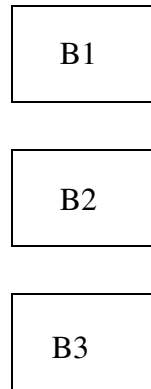
```

void getSum ( int n )
{
    int i;
    SEMA_take(&gsCricSectSema,-1);
    sum = 0;
    for ( i = 0 ; i < n ; i++ )
        sum += i;
    SEMA_give(&gsCricSectSema);
}

```


3.3 Semaphores for Implementing Resource management:

Semaphores are also useful for implementing the resource management. Consider a printer system as an example. Task taskFill's job is to fill the data that has to be printed onto specified buffers. 3 equal sized buffers are kept for this purpose.



Here buffer is the resource. When the resource doesn't exist taskFill should be in blocked state. It has to become runnable as soon as buffers resource is available. When the content of a buffer is printed on the paper, it becomes free and available for the taskFill function.

```
fillBuffer()                                dataPrinted()
{
    while(1)
    {
        SEMA_take(&fillSema,-1);
        /* Copy buffer with data
        to be printed */
    }
}

fillTask()
{
    while ( 1 )
    {
        bufIdx = 0;
        getData(); /* Get the data to be filled onto the buffer */
        fillBuffer();
        bufIdx++;
        if ( bufIdx == 3 )
            bufIdx = 0;
    }
}

dataPrinted()
{
    SEMA_give (&fillSema);
}
```

Function getData is used to get the data to be filled onto the buffer. Once the data is available data is copied onto the buffer of bufIdx using fillBuffer.

Initially when fillTask begins execution, buffers B1, B2, B3 will all be filled with data. But even if we have more data, fillTask will get blocked in semaphore fillSema. This is because the resource doesn't exist. As soon as content on buffer is printed upon

dataPrinted will be called which will release/give the resource. taskFill will take up the resource and fills in with the data to be printed.

Thus semaphores are used for resource management.

3.4 Semaphores for Implementing Event Notification:

Semaphores can also be used for event notification. Consider the case of key press being the event and getKey is the function that waits for the event key press and returns the value of the key pressed.

```
Int8 getKey()
{
    SEMA_wait(keySema,-1);
    /* Read and Return the result */
}

void keyIsr(Int16 intNumber, void *ptr)
{
    /* Read the value of key pressed and store it in buffer */
    /* Notify the get key about the key pressed Event */
    SEMA_iGive(keySema);
}
```

This can be achieved using the above mentioned code. getKey(), when invoked will be waiting for the key press. When the key is pressed keyIsr will be invoked. It will store the information regarding the key pressed and notify the getKey of the event. Then getKey reads the information and returns the result. Thus semaphores can be used for event-notification.

3.5 Semaphores and Task Linkage:

Semaphores and tasks are two basic building block modules in EFOS. We have seen before task can be in one of four states RUNNABLE, SUSPENDED, BLOCKED (-Or-) DEAD. A Task moves from Runnable state to Blocked State when it invokes SEMA_take with a semaphore whose count is 0. Similarly the task becomes runnable when it acquires the resource it needs. In this way task's state is controlled by semaphores.

4. Memory Manager:

Memory Manager is used for managing the heap section of the memory. Memory manager in EFOS can handle multiple memory segments. Memory Manager is available as an add-on module in EFOS.

A memory segment is defined as continuous chunk of memory region. A memory segment is defined by two parameters

- a. Base Address
- b. Segment Size

Base address holds information about where the memory segment starts from. Segment size as its name suggests represents the size of the memory segment. In EFOS each memory segment is assigned a unique segment number. This number is used as argument to the MM_alloc and MM_free calls. MM_alloc/MM_free calls are similar to that of standard malloc/free calls. One difference between malloc/free calls and MM_alloc and MM_free calls is that MM_alloc and MM_free calls take segment index and alignment as extra arguments.

5. EFOS PIPE

Pipe is one of the inter-task communication modules available in the EFOS. Pipe is available in EFOS as an add-on module. Pipe can also be used for data communication between ISR and application buffer



A Pipe can be created of any given size. In case of Inter-task communication there will be one task that will be writing data to the pipe (-Or-) providing input to the pipe. There will be another task that will be reading the data from the pipe (-Or-) taking output from the pipe. writerTask is one task that writes data onto the pipe. readerTask is one that reads the data from the pipe. PIPE provides two standard APIs for reading and writing data to and from the pipe. PIPE_read/PIPE_write can be blocking (-Or-) non-blocking.

5.1 Blocking Versus Non-Blocking PIPE_read:

PIPE_read takes two inputs

- (i) Input Buffer, Pointer onto which the data has to be read into.
- (ii) Number of bytes to be read

Now two conditions exist

- (a) The number of bytes requested is available on the pipe
- (b) The number of bytes requested is not available on the pipe

In case of (a) the data is read from the pipe and copied onto the given pointer irrespective of whether it is blocking PIPE_read (-Or-) non-blocking PIPE_read.

In case (b) the behavior differs between blocking and non-blocking read. In case of non-blocking read whatever data is available in pipe is read onto the input buffer. Number of bytes read is returned as the return value. In case of blocking read the reader task will be blocked till the requested number of bytes is read from the pipe. It is important to note that requested number of bytes can be more than that of the pipe size.

5.2 Blocking Versus Non-Blocking PIPE_write:

PIPE_write takes two inputs

- (i) Output Buffer, Pointer from which the data has to be copied from
- (ii) Number of bytes to be written

Now two conditions exist

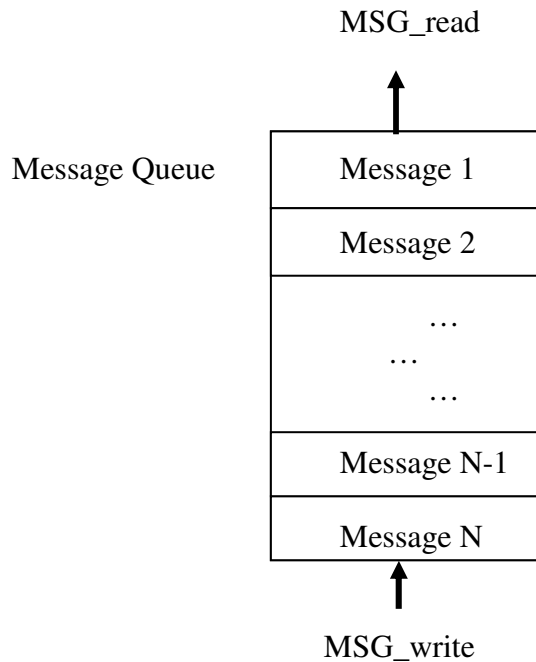
- (a) Free space available in pipe is more than the number of bytes to be written
- (b) Free space available in pipe is less than the number of bytes to be written

In case of (a) data is copied from the output buffer onto the pipe irrespective of whether it is blocking (-Or-) non-blocking write.

In case of (b) the behavior differs between blocking and non-blocking write. In case of blocking write, whatever data that can be copied from the output buffer to pipe is copied onto pipe. Number of bytes copied is returned as the return value. In case of non-blocking write the writer task will be blocked till all the data to be written is copied completed. It is important to note that output buffer size can be bigger than that of the pipe size

6. EFOS MSG:

MSG is one of the Inter-task communication mechanisms available in the EFOS. Inter-task communication is used for communication between two tasks. MSG is available as an add-on module in EFOS.



Two attributes that define the MSG object is

- (i) Size of the message queue, this represents the maximum number of messages that it can hold at a given instance of a time.
- (ii) Message Size, gives the details regarding the size of each message.

Messages are read from the Message Queue in FIFO (First In First Out) fashion. FIFO essentially means that messages are read in the same order as it is written onto the queue. Two APIs MSG_read and MSG_write are used for reading and writing data onto the message queue.

6.1 Blocking versus Non-blocking MSG_read:

MSG_read can be either blocking (-Or-) non-blocking. When MSG_read is invoked two conditions exists

- (i) A new message is already waiting

(ii) No new message is waiting

In the case (i) the result is same for blocking and non-blocking MSG_read. The first message that is waiting in the queue will be read. In the case of (ii) the behavior differs depending upon whether MSG_read is blocking (-Or-) not. Incase of non-blocking MSG_read it returns error denoting that no new message is waiting in the queue. Incase of blocking MSG_read the task that invokes the MSG_read gets blocked with a given timeout. It becomes runnable when the new message is written onto the queue (-Or-) timeout expires.

6.2 Blocking versus Non-blocking MSG_write:

MSG_write can be either blocking (-Or-) non-blocking. When MSG_write is invoked two conditions exists

- (i) Message Queue has space for a new message
- (ii) Message Queue is full.

In the case (i) the result is same irrespective of whether it blocking write (-Or-) non-blocking write. It appends the message to the end of the queue and returns success. In the case (ii) the behavior differs depending upon whether the MSG_write is blocking (-Or-) not. Incase of non-blocking MSG_write it returns error denoting that message queue is full. Incase of blocking write the task that invokes MSG_write gets blocked with a given timeout. It gets unblocked when a message is read (-Or-) timeout expires.

7 Revision History

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