OpenVMS Technical Journal V9

OpenVMS Mailboxes: Concepts, Implementation, and Troubleshooting

Bruce Ellis, President, BRUDEN-OSSG

Overview

This article intends to cover mailboxes from the basic concepts through advanced troubleshooting. If you are just starting with mailboxes, you might want to read from the beginning to the basic examples. If you have experience with mailboxes you may want to move ahead to the troubleshooting section. Hopefully, there is something for everybody in this article.

Much of the implementation details, starting at the "Mailbox Creation" section are discussed in more detail in the *HP OpenVMS I/O User's Reference Manual* Chapter 4 and under the $CREMBX section of the *HP OpenVMS System Services Reference Manual.*

Inter-process Synchronization and Communication

OpenVMS processes provide an environment in which programs can be executed. This environment includes software context, hardware context, and virtual address space. The "divide and conquer" approach to problem solving allows different programs, running simultaneously under different processes, to take on parts of a task concurrently. To support this design, the processes need methods to communicate with one another and to synchronize, or coordinate, activities between the processes.

OpenVMS provides several methods for interprocess synchronization and communication. Methods for interprocess communication include shared files, logical names, mailboxes, and global sections (shared virtual memory). Inter-process synchronization methods include common event flags, mailboxes, and lock management services.

Shared files are generally slow methods of communication. Logical names are potentially faster than shared files, but extensive use may fragment paged pool. Global sections are probably the fastest form of interprocess communication. The one major drawback to each of these methods is that there is no built-in signaling mechanism to notify the target process that there is a need to obtain the new data. In each case, the application could poll for new data, but this wastes CPU time and/or may cause delays in event notification.

For synchronization within a single system, common event flags have limited name space. You can only wait on one common event flag cluster at a time and there are only 32 (single-bit) event flags per cluster. Lock management system services are designed more for coordination of activities than signaling, although signaling mechanisms can be implemented using the lock management services. Mailboxes provide methods that allow processes to communicate with one another and to receive notification that there is data to be processed. In addition, there is an implicit queuing mechanism for multiple messages that have been written to the mailbox. The programming interface to mailboxes is simple to implement and can be written in just about any programming language, including DCL.

Mailbox Concepts

Mailboxes are pseudo-devices, similar to UNIX-style pipes. However, mailboxes allow bi-directional communication, i.e., a single process can read and write the same mailbox. Messages written to a mailbox are queued in first-in-first-out fashion. To implement pipe-oriented communication, channels can be assigned to a mailbox, such that the mailbox channel can only be written, or conversely, can only be read.

A mailbox can have multiple writers and multiple readers, although multiple reader designs are probably rarer than multiple writers. It is usually easier to implement a single reader of a mailbox (Figure 1).

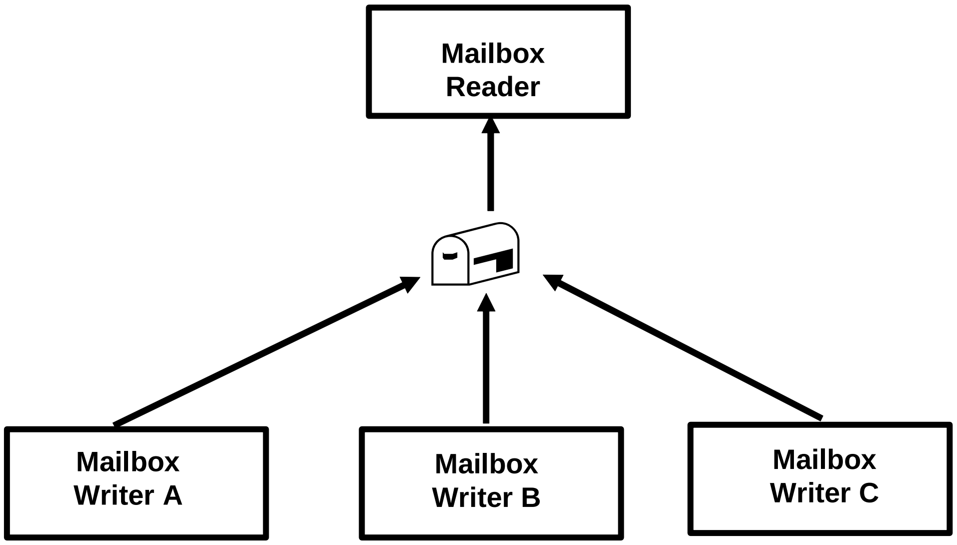

Figure 1. Sample Mailbox Design

When the writer wants to get messages back from the reader, it may use various methods, including creating a separate mailbox and passing along the mailbox unit number to the reader. The "reader" would assign a channel to the target mailbox unit and send a response as in Figure 2.

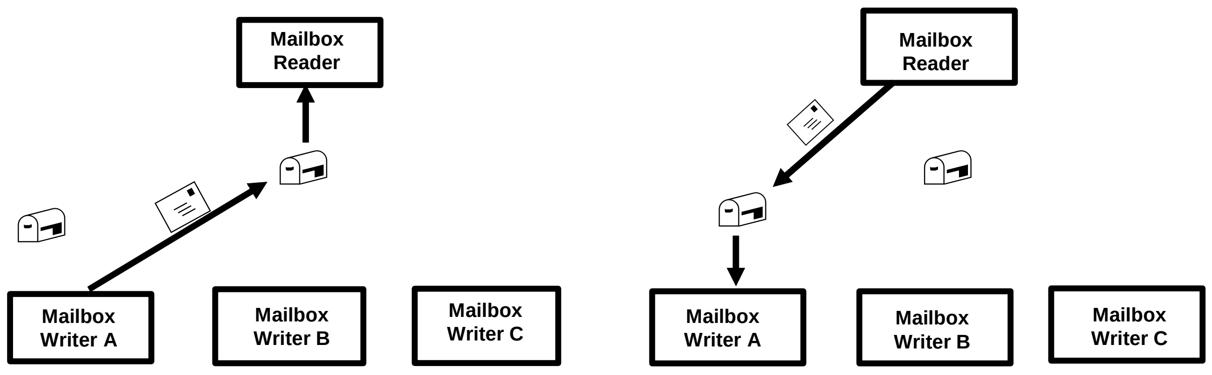

Figure 2. Communication between Mailbox Writer and Reader

In general, a mailbox read operation does not complete until there is a corresponding write operation. Similarly, write operations do not complete until there is a corresponding read operation. Applications that perform synchronous read or write operations will stall, waiting on an event flag (in the scheduling state LEF), until the counterpart operation is issued by another process.

Mailbox operations can be performed using high-level language I/O constructs, but more commonly are processed using the QIO (sys$qio) system service. The specifics of QIO operations on mailboxes are documented in chapter 4 of the *OpenVMS I/O User's Reference Manual.* Before you can issue a QIO on a device, a channel must be assigned. The channel identifies the device on your QIO system service calls. For more information on QIO and channels see chapter 23 of the *OpenVMS Programming Concepts Manual.*

Mailbox writes can be forced to complete immediately upon queuing by using a QIO system service function modifier (IO$M_NOW). This mechanism is different than performing an asynchronous QIO, in that the I/O request is not pending. What this means is that if a program that has issued a write using the IO$M_NOW modifier exits, its write stays queued to the mailbox, as long as some process on the system is interested in the mailbox (has a channel assigned to it). If a write was issued asynchronously without the IO$M_NOW modifier and the program exits, the write is canceled (when the channel to the mailbox is deassigned) and the write is lost.

Data that is written to the mailbox can be in any form and can vary in size. The mailbox driver simply treats the data as an array of bytes. The writer identifies the number of bytes being written to the mailbox. The size can vary from 0 to 64,000 bytes, dependent on the maximum message size assigned to the mailbox. The 64,000 byte limit is based on the fact that mailbox messages are allocated from non-paged dynamic memory (a.k.a. non-paged pool). Pool packets contain a word (16-bit) sized field to identify the amount of pool that the packet occupies.

The mailbox reader must supply a buffer that is large enough to hold the largest data item that will be written to the mailbox. To determine the number of bytes actually written, the reader should pass an I/O status block on a QIO to the mailbox driver. The reader can examine the size field in the I/O status block upon completion of the read.

Mailbox Creation

Before a mailbox can be used, it must be created. Mailbox creation is performed using the Create Mailbox (sys$crembx) system service. When a mailbox is created, it is assigned a name of the form MBA*u*, where *u* is a unit number assigned by OpenVMS. Prior to V8.2, OpenVMS limited the unit numbers on mailboxes to 9999. Additionally, mailbox creation and deletion required sequential scans of all existing units, which could be a slow process. V8.2, and greater, systems were modified to support up to 32,767 mailbox units. The I/O database was optimized to speed the creation and deletion of mailboxes.

The program does not generally have knowledge of the mailbox device name that it is creating, as OpenVMS dynamically defines the name. To associate multiple processes to the same mailbox, processes usually identify the mailbox using a logical name. The logical name is passed by descriptor as the seventh argument to the sys$crembx system service call.

To create a mailbox at the DCL level you can use the command CREATE/MAILBOX.

Temporary and Permanent Mailboxes

The first parameter to sys$crembx is a flag that identifies whether the mailbox is a temporary or permanent mailbox. If the flag is set you get a permanent mailbox, otherwise you get a temporary mailbox.

Temporary mailboxes require TMPMBX privilege to create. They are deleted when all channels to the mailbox have been deassigned. The logical name passed to sys$crembx is cataloged in the logical name table identified by the logical name LNM$TEMPORARY_MAILBOX. By default, this logical name is assigned to LNM$JOB. Therefore, by default, using the logical name passed to the sys$crembx, system service associates processes in the same job to the same mailbox. Changes to this logical name are best made within the program, in user mode. Changes made at the DCL level may cause problems with SPAWN/ATTACH commands. If you do choose to change the logical name LNM$TEMPORARY_MAILBOX at the DCL level, make sure to change it in the LNM$PROCESS_DIRECTORY logical name table.

Permanent mailboxes require PRMMBX privilege to create and delete. They must be explicitly deleted using the sys$delmbx system service. The mailbox is actually deleted after all channels to the mailbox have been deassigned. The logical name passed to sys$crembx is cataloged in table identified by the logical name LNM$PERMANENT_MAILBOX. This logical name is set to LNM$SYSTEM by default.

If you are using DCL to create a mailbox, you can define that the mailbox will be temporary or permanent using the /TEMPORARY or /PERMANENT qualifiers, respectively. The default qualifier is /TEMPORARY. Just like the sys$crembx system service, you need TMPMBX privilege to create temporary mailboxes and PRMMBX privilege to create permanent mailboxes. You also need CMEXEC privilege to create a temporary mailbox. This privilege is required to allow the mailbox to be created in supervisor mode. The plan is to remove this restriction in the future. You may also need SYSNAM or GRPNAM privilege to create the logical name associated with the mailbox in the appropriate logical name table.

Permanent DCL-created mailboxes can be deleted using the DELETE/MAILBOX command. When all channels are deassigned the mailbox will go away. Currently, there is no supported way to deassign a channel to a DCL-created mailbox without logging out. Therefore, there is no supported way to delete a temporary mailbox without logging out. We at BRUDEN-OSSG, of course, have a method to get the channel deassigned.

Example 1. Viewing the Temporary and Permanent Logical Name Table Assignments

```
$ show logical/table=lnm$system_directory *mail*

(LNM$SYSTEM_DIRECTORY)

  "LNM$PERMANENT_MAILBOX" = "LNM$SYSTEM"
  "LNM$TEMPORARY_MAILBOX" = "LNM$JOB"
$
```

When the mailbox has been created, the channel number assigned to the mailbox is returned to the address passed as the second parameter to the sys$crembx system service. If multiple processes are going to be accessing the same mailbox and one process is guaranteed to create the mailbox, the rest of the processes can simply assign channels to the mailbox.

If the mailbox creator is not guaranteed to be a specific process, all processes can call the sys$crembx system service. After the mailbox has been created, the sys$crembx system service simply assigns a channel to the mailbox. Care should be taken to make sure that arguments to the sys$crembx system service match for all users of the same mailbox. If one process sets the prmflg (permanent flag) and another passes a zero for the argument, you end up creating two different mailboxes (one permanent and one temporary). Additionally, parameters used to size the mailbox and establish protections are assigned by the first process calling the service (the process that actually creates the mailbox).

Mailbox Protections

Protection on a mailbox is set when the mailbox is created. The fifth argument to sys$crembx identifies the protection mask. If the protection mask is 0, the template protection mask is used. This mask defaults to allowing all access to all UIC categories. In the protection mask, bits <15:12> identify world, bits <11:8> group, bits <7:4> owner, and bits <3:0> system access. The categories for each mode are LPWR (Logical, Physical, Write, and, Read). Bits clear allow access. Bits set deny access. Logical access is required for any other form of access. Physical access is ignored. A setting of the hex value 0xF000 would allow all access for System, Owner, and Group, denying access for the World category. The setting 0xF200 would write access for the Group category and all access for the World category.

Example 2. Sample Call to sys$crembx Disabling World Access to a given Mailbox

```
/* Assign a channel to the mailbox. */
        status = sys$crembx(0,&mbx_chan,0,0,0xF000,0,&mbx,0,0);
        check(status);
```

Example 3. Viewing the Protections from the Mailbox Created in Example 2.

```
$ SHOW DEVICE MBA28282:/FULL

Device MBA28282:, device type local memory mailbox, is online, record-oriented
    device, shareable, mailbox device.

    Error count                    0    Operations completed               0
    Owner process                 ""    Owner UIC                [JAVA,ELLIS]
    Owner process ID       00000000    Dev Prot        S:RWPL,O:RWPL,G:RWPL,W
    Reference count               1    Default buffer size                256

$
```

In addition to the sys$crembx argument for protections, there is an IO$M_SETPROT function modifier on the IO$_SETMODE function that accepts a protection mask on the P2 argument to the sys$qio system service. You can also set up objects rights on your mailbox.

DCL-created mailboxes have protections assigned using the /PROTECTION qualifier on the CREATE/MAILBOX command.

Read/Write Only Channels

Within a program you can force read-only or write-only access on a mailbox channel (similar to a unidirectional pipe), using the flags CMB$M_READONLY or CMB$M_WRITEONLY (defined in $CMBDEF/cmbdef.h) in the eighth argument to sys$crembx. If you are assigning a channel, the flags AGN$M_READONLY or AGN$M_WRITEONLY can be used to restrict access. The restriction is only in effect for I/O requests issued within a given application.

The closest equivalent to a sys$assign system service call from DCL is an OPEN command. The CREATE/MAILBOX command does not implicitly perform an OPEN command. So, before processing a mailbox, it must have been created by some process and must be opened by all processes accessing the mailbox. DCL-created mailboxes support read-only mailboxes through the OPEN/READ command, but not write-only mailboxes.

Mailbox Sizing

To understand sizing issues that relate to mailboxes we should take a different view of a mailbox. When a mailbox is created, OpenVMS creates a data structure called a Unit Control Block (UCB) in non-paged pool. The UCB has a specialized layout that supports mailbox operations. The UCB maintains queues. There is a message queue for messages written to the mailbox. There is a reader queue that tracks read I/O requests to the mailbox. The data structures queued to reader queue are called I/O Requests Packets (IRPs).

The UCB also maintains queues to allow processes to be notified of unsolicited read or write operations (read with no pending write or write with no pending read). Processes are notified of these events through the delivery of an Asynchronous System Trap (AST), known as an *attention AST*. A similar attention AST can be delivered when space becomes available in a full mailbox.

There are also queues that allow your process to be notified when a new read or write channel is assigned to a mailbox.

The point of this discussion is that when you create a mailbox, regardless of how you size it, you are only creating the UCB for the mailbox. The sizing parameters limit the use of non-paged pool space to describe messages that are queued to the UCB. So, a more accurate view of a mailbox with three write requests and no current read looks like figure 3. The "MBOX" headers describe the layout of the message block. These symbolic offsets may not be available in earlier versions of OpenVMS. A view of a mailbox with no active writes and one read looks like figure 4.

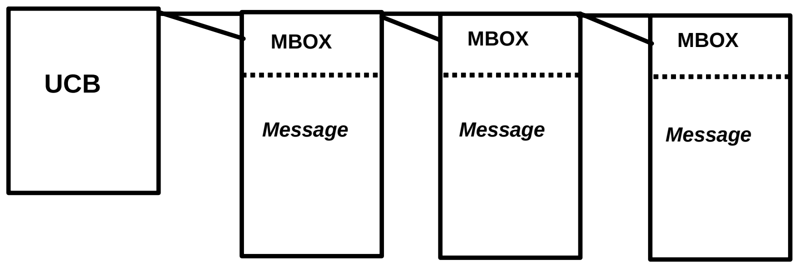

Figure 3. Mailbox with three Pending Writes

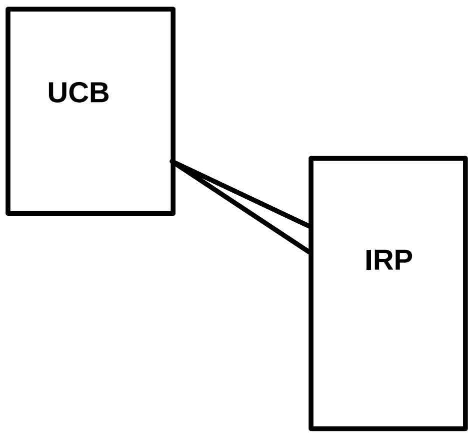

Figure 4. Mailbox with one Pending Read

When a mailbox is created, the third argument to sys$crembx is the maximum message size and the fourth argument is the mailbox buffer quota. The maximum message size restricts the size of an individual message that can be written to the mailbox. This setting can be used to set the size of the input buffer by the reader. If this size is not specified, it is set by the system parameter DEFMBXMXMSG. On the CREATE/MAILBOX command, the /MESSAGE_SIZE qualifier specifies the maximum message size.

The buffer quota is effectively the "size" of the mailbox. It is the maximum number of bytes that can be written to the mailbox. Setting a large buffer quota does not cause any space to be allocated from non-paged pool. What it does do, is allow that many bytes to be potentially allocated from non-paged pool to support mailbox writes. When an attempted write would cause a given mailbox to exceed its buffer quota, the mailbox is considered full and the write will either stall or fail. On the CREATE/MAILBOX command, the BUFFER_SIZE qualifier specifies the mailbox size.

If the buffer quota is not specified on the call to the sys$crembx system service, the setting for the system parameter DEFMBXBUFQUO is used to size the mailbox. The maximum advertised setting for this parameter is 64,000 bytes. You can override checks in SYSGEN and set the parameter to a higher setting, if you are running V7.3-1 or greater. This should be done with great caution, as it will affect the default size of all mailboxes that do not specify a non-zero buffer quota parameter on a call to sys$crembx. You can alternatively, and more safely, set a buffer quota parameter larger than 64,000 bytes as a buffer quota parameter on mailbox creation for select mailboxes.

The key thing to keep in mind when setting larger buffer quota settings is that you do not exhaust non-paged pool. If you are going with higher settings for buffer quotas, compensate with correspondingly larger settings for the system parameters NPAGEDYN and NPAGEVIR.

You can monitor mailbox space usage using the IO$_SENSEMODE function to the sys$qio system service. This function receives no function dependent parameters (P1-P6). It returns the number of messages queued to the mailbox in the iosb$w_bcnt field of the I/O status block. It returns the number of message buffer bytes in the iosb$l_dev_depend field of the I/O status block. You can obtain the buffer quota and remaining buffer using the DVI$_MAILBOX_INITIAL_QUOTA and DVI$_MAILBOX_QUOTA items through the sys$getdvi system service. Example 4 shows a program that obtains and displays information on mailbox usage. There is a sample SDA extension in the SYS$EXAMPLES directory, named MBX$SDA.C, that you can build and obtain more complete information on all mailboxes on the system. We will discuss troubleshooting full mailboxes later in this article.

Example 4. Sample Program to Monitor Mailbox Usage

The following program is implemented as a foreign command. It accepts a mailbox name and displays the number of outstanding messages queued to the mailbox, the bytes in use, bytes available, and mailbox size.

```
$ type mbx_usage.c
// Sample program to display total and available mailbox space.
// Implemented as a foreign command.  Mailbox name is passed in on the command
// line
// Author: Bruce Ellis, BRUDEN-OSSG
#include <stdio.h>
#include <starlet.h>
#include <dvidef.h>
#include <iodef.h>
#include <iledef.h>
#include <iosbdef.h>
#include <descrip.h>
#include <string.h>
#include <ssdef.h>
#include <efndef.h>
#define check(S) if(!((S)&1)) sys$exit(S)

#define MBX 1
#define EXPECTED_ARGS 2
int     main(int argc, char **args)
{

        struct dsc$descriptor_s  mbx_name;
        unsigned int    mbx_size;
        unsigned int    mbx_avail;
        ile3    dvi_list[] = {{sizeof(mbx_size),DVI$_MAILBOX_INITIAL_QUOTA,
                                &mbx_size},
                        {sizeof(mbx_avail),DVI$_MAILBOX_BUFFER_QUOTA,
                                &mbx_avail}, {0,0}};
        iosb    ios;
        int     status;
        short chan;
// If we do not have a mailbox name, exit
        if(argc != EXPECTED_ARGS)
        {
                sys$exit(SS$_NOSUCHDEV);
        }
// Set up mailbox name descriptor
```

```
        mbx_name.dsc$w_length = strlen(args[MBX]);
        mbx_name.dsc$a_pointer = args[MBX];

// Get mailbox information
        status = sys$getdvi(EFN$C_ENF,0,&mbx_name,dvi_list,&ios,0,0,0,0);
        check(status);
        check(ios.iosb$w_status);
// Get more information from QIO
        status = sys$assign(&mbx_name,&chan,0,0,0);
        check(status);
        status = sys$qiow(0,chan,IO$_SENSEMODE,&ios,0,0,0,0,0,0,0,0);
        check(status);
        check(ios.iosb$w_status);
// Display info.
        printf("Mailbox size: %d\nRemaining bytes in mailbox: %d\n",
                mbx_size,mbx_avail);
        printf("Number of messages in the mailbox: %hd\nNumber of message bytes: %d\n",
              ios.iosb$w_bcnt, ios.iosb$l_dev_depend);
        return(SS$_NORMAL);
}

$
```

Compile and link the program.

```
$ cc mbx_usage
$ link mbx_usage
```

Setup a foreign command symbol to run the program.

```
$ mbu:== "$sys$login:mbx_usage"
```

View a sample mailbox.

```
$ mbu MBA28605
Mailbox size: 100000
Remaining bytes in mailbox: 100000
Number of messages in the mailbox: 0
Number of message bytes: 0
$
```

Find OPCOM. Note: OPCOM reads from the mailbox MBA2:

```
$ show system/process=opcom
OpenVMS V8.3  on node ALPH40  18-NOV-2006 21:49:01.13  Uptime  55 02:47:35
  Pid    Process Name    State  Pri      I/O       CPU       Page flts  Pages
20400410 OPCOM           HIB      8      406   0 00:00:00.24       688     43
```

No current activity on MBA2:

```
$ mbu MBA2
Mailbox size: 65535
Remaining bytes in mailbox: 65535
Number of messages in the mailbox: 0
Number of message bytes: 0
```

Suspend OPCOM.

```
$ set process/suspend/id=20400410
```

Send some data to MBA2:

```
$ spawn/nowait request "Please service this request!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!"
%DCL-S-SPAWNED, process ELLIS_14466 spawned
$ spawn/nowait request "Please service this request!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!"
%DCL-S-SPAWNED, process ELLIS_28545 spawned
$ spawn/nowait request "Please service this request!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!"
%DCL-S-SPAWNED, process ELLIS_19638 spawned
$ spawn/nowait request "Please service this request!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!"
%DCL-S-SPAWNED, process ELLIS_16419 spawned
$ spawn/nowait request "Please service this request!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!"
%DCL-S-SPAWNED, process ELLIS_48996 spawned
```

View OPCOM's mailbox. Note, the activity.

```
$ mbu MBA2
Mailbox size: 65535
Remaining bytes in mailbox: 64907
The number of messages queued seems to be off by 1.  Note: a user mode AST
has been queued to OPCOM to service the completion of the first request.  The
AST could not be delivered because OPCOM was suspended.  Therefore, the
first message has been pulled from the queue to be serviced, dropping the
message count by 1.
Number of messages in the mailbox: 4
Number of message bytes: 628
$
Resume OPCOM and clean out the mailbox.
$ set process/resume /id=20400410
$ mbu MBA2
Mailbox size: 65535
Remaining bytes in mailbox: 65535
Number of messages in the mailbox: 0
Number of message bytes: 0
$
```

Mailboxes and Quotas

When a temporary mailbox is created, the creating process has the buffer quota charged against its buffered byte limit (BYTLM). In the case of permanent mailbox, no process is charged for the buffer quota. Since the quota for the mailbox has been handled, individual I/O requests are not charged against the job's BYTLM. However, for all sys$qio calls that do not use the IO$M_NOW function modifier, the process' buffered I/O limit (BIOLM) is charged. Writes issued with the IO$M_NOW modifier are not charged against the process's BIOLM, since they may persist beyond the life of the program and possibly the process that issued them.

Mailbox Processing

As we mentioned earlier, mailboxes can be read and written using high-level language constructs, but are more commonly read and written using sys$qio system service calls. If you are not familiar with programming calls to sys$qio, you should invest some time reading the *OpenVMS Programming Concepts Manual*. Common mistakes that beginners make when coding sys$qio calls include:

- Using a call to sys$qio, instead of using sys$qiow. The sys$qiow has an implicit wait until the call has been serviced. Using sys$qio calls work fine, as long as you implement waits at some point in your program, usually through a call to sys$synch. With no explicit or implicit waits, messages are queued up to the mailbox, causing it to fill and the application to hang.
- Not passing and checking the I/O status block (IOSB) parameter. Status returned on the call to sys$qio indicates whether the call was issued properly. It does not indicate whether the I/O request completed properly. Completion status is returned in the low word of the IOSB structure. This is described in the synchronization section of the *OpenVMS Programming Concepts Manual*.

Reading Mailboxes

Mailboxes are read through sys$qio using one of the function codes: IO$_READVBLK, IO$_READLBLK, or IO$_READPBLK. For mailboxes, there is no difference between the function codes. The sys$qio system service provides a uniform interface to all devices. Other devices will give different meaning to the three functions within the context of the device.

When using one of the read functions, the input buffer is passed by address in the P1 parameter to sys$qio. The size of the buffer is passed in the P2 parameter. The size should allow for the maximum message size allowed for the mailbox. If the message size allowed on a read is smaller than the amount of data written, a status of SS$_BUFFEROVF is returned in the IOSB status field. The data beyond the end of the buffer is lost.

The actual number of bytes written to the mailbox may be less than the size of the read buffer. The actual number of bytes written to the mailbox is returned in the iosb$w_bcnt field of the IOSB.

If the data in message buffers is larger than you are anticipating in the input buffer, you can preserve the data in the message buffer using the function modifier IO$M_STREAM on the read. Subsequent reads will pick up the remnant data in the message buffer.

When a read is posted on a mailbox, it will not complete until a corresponding write is issued. This can cause the application to hang if there is no current writer. In many cases, this behavior is fine and desired. In cases where the writer may have failed, this behavior may cause functional problems in the application. There are several ways to deal with this potential problem, including:

- Using the function modifier IO$M_NOW with a read function. If there are no pending writes, the read will complete immediately with a zero byte read. In my opinion, this is usually an undesirable option. It causes convoluted and potentially poor performing code.
- Using the function modifier IO$M_WRITERCHECK with a read function. This request will return a status of SS$_NOWRITER if there is no data in the mailbox and there are no write channels assigned to the mailbox. This option only works if the channel assigned by the process using it was assigned as a read-only channel. A variation of this method can be implemented using IO$M_WRITERCHECK with an IO$_SENSEMODE function.
- Using the function modifier IO$M_WRITERWAIT with the IO$_SETMODE function. The event flag set can be checked or an AST can be delivered to the process notifying it that there is a write channel assigned. As in the last bullet, this method only works with unidirectional mailboxes.
- Using a sys$setimr and an asynchronous sys$qio, then waiting for a "logical or" of the event flags. You can use the sys$readef system service to determine whether the timer expired or the read completed first and then process accordingly.

To determine whether a writer has completed a multi-write transmission, the cooperating processes can use the IO$_WRITEOF function in the context of the writer, and the reader can check for a status of SS$_ENDOFFILE in the IOSB.

On a read function, the device dependent field of the IOSB contains the process identification (PID) of the writer, unless the writer is a system process.

DCL READ commands issued on mailboxes will read their contents and store them in symbols. Be cautious of performing READ (and WRITE) commands interactively. They block execution of the supervisor mode control Y AST.

Writing Mailboxes

Mailboxes can be written using the sys$qio function codes IO$_WRITEVBLK, IO$_WRITELBLK, or IO$_WRITEPBLK. Just as on writes, these function codes have identical meanings. The write functions support a IO$M_READERCHECK function modifier that operates in similar fashion to the IO$M_WRITERCHECK on read functions.

Simple mailbox writes do not complete until a corresponding mailbox read is issued. A write to a mailbox can be forced to complete using the function modifier IO$M_NOW. A message written with this function modifier will be queued to a mailbox and control will be returned to the writer. As long as the mailbox is not deleted, the message will stay in the mailbox until it is read. You need to take caution that some process has a channel assigned to a temporary mailbox, or the mailbox will be deleted when the writer program runs down.

The IO$M_NOW function modifier should be used with care to prevent possibly filling the mailbox.

To notify the reader that we are done transmitting data, you can send an "end of file" by using the IO$_WRITEOF function code. The only effect of using this function is that a status value of SS$_ENDOFFILE is returned to the reader's IOSB. This technique is an optional method to signal that one stream of data is complete. The reader could terminate on detection of this status or could start processing another stream.

On a write function, the device dependent field of the IOSB contains the PID of the reader of the mailbox, except when the function modifier IO$M_NOW is used. In this case, the field contains 0, as the mailbox has not necessarily been read by the time the write completes.

The DCL WRITE command can be used to write to a mailbox. The qualifier /NOWAIT implements the function modifier IO$M_NOW on a WRITE.

When you issue a close on a DCL-created mailbox, there is effectively an IO$_WRITEOF function performed.

Simple Mailbox Examples

At this point, it would probably be good to take a look at a couple of simple examples that use mailboxes for communication. Example 5 illustrates a simple mailbox writer program. The program reads strings from sys$input and sends them to the mailbox named DATA_MBX. When an end of file is read from sys$input, a sys$qio is issued with a function of IO$_WRITEOF. Example 6 illustrates a simple mailbox reader program. The program reads from the data mailbox and sends output to sys$output until a write using the function code IO$_WRITEOF is detected. Example 6a is a DCL version of examples 5 and 6.

The logical name LNM$TEMPORARY_MAILBOX is assigned to the logical name LNM$GROUP in user mode. This practice allows the programs to be run from two different interactive sessions. Sample runs of the programs are shown in each example. Note: the runs from the writer were run in parallel with the reader.

Example 5. Simple Sample Mailbox Writer

```
$ type mailbox_writer.c
/*
        Simple mailbox writer.  Reads lines from standard input until
        EOF and sends to a mailbox named DATA_MBX.

        Author: Bruce Ellis, BRUDEN-OSSG
*/
#include <starlet.h>
#include <iodef.h>
#include <ssdef.h>
```

```
#include <stdio.h>
#include <iosbdef.h>
#include <descrip.h>
#include <iledef.h>
#include <lnmdef.h>
#include <string.h>
#include <lib$routines.h>

#define MBX_PROT 0xF000
#define MAX_MSG 1024
#define BUF_QUO 60000
#define LIST_END 0
#define check(S) if(!((S)&1)) sys$exit(S)

int     main(void)
{

        iosb    ios;
        int     status;

        $DESCRIPTOR(ptable,"LNM$PROCESS_DIRECTORY");
        $DESCRIPTOR(lnm,"LNM$TEMPORARY_MAILBOX");
        char    equiv[ ] = "LNM$GROUP";
        ile3    lnm_items[ ] = {{strlen(equiv),LNM$_STRING,equiv},{LIST_END}};
        $DESCRIPTOR(mbx,"DATA_MBX");
        short   chan;
        int     efn;
        char    in_buffer[BUFSIZ];
/* Create a logical name to allow the next temporary mailbox's name
   we create to be placed in the group logical name table.
*/
        status = sys$crelnm(0,&ptable,&lnm,0,lnm_items);
        check(status);
/* Create/assign a channel to the data mailbox. */
        status = sys$crembx(0,&chan,MAX_MSG,BUF_QUO,MBX_PROT,0,&mbx,0,0);
        check(status);
/* Get an available event flag number. */
        status = lib$get_ef(&efn);
        check(status);

/* Read from standard input and send to mailbox until EOF. */
        while(gets(in_buffer))
        {
        /* If input buffer is too large, abort. */
                if(strlen(in_buffer)>MAX_MSG)
                {
                        sys$exit(SS$_BUFFEROVF);
                }
                status = sys$qiow(efn,chan,IO$_WRITEVBLK,&ios,0,0,
                        in_buffer,strlen(in_buffer),0,0,0,0);
                check(status);
                check(ios.iosb$w_status);
        }
/* Send an EOF to the mailbox. */
        status = sys$qiow(efn,chan,IO$_WRITEOF,&ios,0,0,
                0,0,0,0,0,0);
        check(status);
        check(ios.iosb$w_status);

        return(SS$_NORMAL);
}

$
$ cc mailbox_writer
$ link  mailbox_writer
$ r mailbox_writer
Bruce Ellis was here
Welcome to Mailboxes from BRUDEN-OSSG
We have lot's of great guys and a great Guy on board.
Control-Z was entered on the next line.
 Exit
$
```

Example 6. Simple Sample Mailbox Reader

```
$ type mailbox_reader.c
/*
        Example of a simple mailbox reader.
        The program reads from a mailbox named DATA_MBX and
        displays the data on sys$output until the writer issues
        an IO$_WRITEOF function.

        Author: Bruce Ellis, BRUDEN-OSSG
*/

#include <starlet.h>
#include <iodef.h>
#include <ssdef.h>
#include <stdio.h>
#include <iosbdef.h>
#include <descrip.h>
#include <iledef.h>
#include <lnmdef.h>
#include <string.h>
#include <lib$routines.h>

#define MBX_PROT 0xF000
#define MAX_MSG 1024
#define BUF_QUO 60000
#define LIST_END 0
#define check(S) if(!((S)&1)) sys$exit(S)

int     main(void)
{

        iosb    ios;
        int     status;

        $DESCRIPTOR(ptable,"LNM$PROCESS_DIRECTORY");
        $DESCRIPTOR(lnm,"LNM$TEMPORARY_MAILBOX");
        char    equiv[ ] = "LNM$GROUP";
        ile3    lnm_items[ ] = {{strlen(equiv),LNM$_STRING,equiv},{LIST_END}};
        $DESCRIPTOR(mbx,"DATA_MBX");
        short   chan;
        char    buffer[MAX_MSG + 1];
        int     efn;
        int     i;

/* Create a logical name to allow the next temporary mailbox's name
   we create to be placed in the group logical name table.
*/
        status = sys$crelnm(0,&ptable,&lnm,0,lnm_items);
        check(status);
/* Create/assign a channel to the listener mailbox. */
        status = sys$crembx(0,&chan,MAX_MSG,BUF_QUO,MBX_PROT,0,&mbx,0,0);
        check(status);
/* Get an available event flag number. */
        status = lib$get_ef(&efn);
        check(status);

        i=1;
/* Read and display until EOF. */
        do
        {
                status = sys$qiow(efn,chan,IO$_READVBLK,&ios,0,0,
                        buffer,MAX_MSG,0,0,0,0);
                check(status);
                if(ios.iosb$w_status != SS$_ENDOFFILE)
                {
                        check(ios.iosb$w_status);
                        buffer[ios.iosb$w_bcnt] = '\0';
                        printf("Message %08d: %s\n",i,buffer);
                        i++;
                }
        } while(ios.iosb$w_status != SS$_ENDOFFILE);

        return(SS$_NORMAL);
}

$
$ cc mailbox_reader
$ link mailbox_reader
$ show logical data_mbx
   "DATA_MBX" = "MBA29808:" (LNM$GROUP_000042)
$ show device data_mbx/full

Device MBA29808:, device type local memory mailbox, is online, record-oriented
    device, shareable, mailbox device.

    Error count                    0    Operations completed                  0
    Owner process                 ""    Owner UIC                  [JAVA,ELLIS]
    Owner process ID       00000000    Dev Prot        S:RWPL,O:RWPL,G:RWPL,W
    Reference count                1    Default buffer size                1024

$
$ r mailbox_reader
Message 00000001: Bruce Ellis was here
Message 00000002: Welcome to Mailboxes from BRUDEN-OSSG
Message 00000003: We have lot's of great guys and a great Guy on board.
$
```

Example 6a. Sample DCL Mailbox Writer and Reader

```
$
```

This is from session 1.

```
$
$ type temp_talker.com
$ on error then goto done
$ on control_y then goto done
$ !
$ !Create the logical name in the group logical name table.
$ define/table=lnm$process_directory lnm$temporary_mailbox lnm$group
$
$ ! create the temporary mailbox
$ create/mailbox/log bru_mbx
$
$ !Go back to standard temporary mailbox logical names
$ define/table=lnm$process_directory lnm$temporary_mailbox lnm$group
$
$ !Open the mailbox for write
$ open/write bmbx bru_mbx
$
$ !Read from the keyboard and send to the mailbox until EOF
$ read_loop:
$       read/prompt="Message: "/end=done sys$command record
$       write/now bmbx record
$       goto read_loop
$ done:
$
$ close bmbx
$
$
$ @temp_talker
%CREATE-I-CREATED, MBA33594: created
%DCL-I-SUPERSEDE, previous value of LNM$TEMPORARY_MAILBOX has been superseded
Message: Bruce Ellis was here
Message: We would not have CREATE/MAILBOX
Message: without a wonderful "Guy" at
Message: BRUDEN-OSSG
```

Control-Z entered here.

```
Message: *EXIT*
$
```

The logical name and device name are still there.

```
$ show logical bru_mbx
   "BRU_MBX" = "MBA33594:" (LNM$GROUP_000042)
$ show device mba33594

Device                  Device          Error
 Name                   Status          Count
MBA33594:               Online              0
$
$ deas_mbx=="$ SYS$SYSDEVICE:[ELLIS]DEAS_DCL_MBX_CHAN"
$ deas_mbx bru_mbx  !This feature is not currently available.
```

The mailbox does not go away until ALL channels are deassigned.

```
$ show device mba33594

Device                  Device          Error
 Name                   Status          Count
MBA33594:               Online              0
$
```

This is after the next session did the deassign.

```
$ show logical bru_mbx
%SHOW-S-NOTRAN, no translation for logical name BRU_MBX
```

```
$ show device mba33594
%SYSTEM-W-NOSUCHDEV, no such device available
$
***********************************************************************************************
This is from a separate session.

$ type temp_listener.com
$
$ on error then goto done
$ on control_y then goto done
$ !
$ !Create the logical name in the group logical name table.
$ define/table=lnm$process_directory lnm$temporary_mailbox lnm$group
$
$ ! create the temporary mailbox
$ create/mailbox/log bru_mbx
$
$ !Go back to standard temporary mailbox logical names
$ define/table=lnm$process_directory lnm$temporary_mailbox lnm$group
$
$ i=1
$ !Open the mailbox and read and echo until end of file
$ open/read  bmbx bru_mbx
$ read_loop:
$       read/end=done bmbx   record
$       write sys$output f$fao("Message !8ZL: !AS",i,record)
$       goto read_loop
$ done:
$ close bmbx
$
$ @temp_listener
%CREATE-I-CREATED, MBA33594: created
%DCL-I-SUPERSEDE, previous value of LNM$TEMPORARY_MAILBOX has been superseded
Message 00000001: Bruce Ellis was here
Message 00000001: We would not have CREATE/MAILBOX
Message 00000001: without a wonderful "Guy" at
Message 00000001: BRUDEN-OSSG
$
$
$ deas_mbx==" $ SYS$SYSDEVICE:[ELLIS]DEAS_DCL_MBX_CHAN"
$
$ show logical bru_mbx
   "BRU_MBX" = "MBA33594:" (LNM$GROUP_000042)
$ show device bru_mbx

Device                  Device           Error
 Name                   Status           Count
MBA33594:               Online               0
$
$ deas_mbx bru_mbx  !This feature is not currently available.
$  show logical bru_mbx
%SHOW-S-NOTRAN, no translation for logical name BRU_MBX
$ show device MBA33594:
%SYSTEM-W-NOSUCHDEV, no such device available
$
```

Example 7 provides a variation on the mailbox reader that uses the IO$M_STREAM function modifier on input. The same writer as example 5 was used in the sample run, with the same output provided.

Example 7. Sample Streaming Reads

```
$ type mailbox_streamer.c
/*
        Example of a simple mailbox reader.
        The program reads from a mailbox named DATA_MBX and
        displays the data on sys$output until the writer issues
        an IO$_WRITEOF function.

        Author: Bruce Ellis, BRUDEN-OSSG
*/

#include <starlet.h>
#include <iodef.h>
#include <ssdef.h>
#include <stdio.h>
#include <iosbdef.h>
#include <descrip.h>
#include <iledef.h>
#include <lnmdef.h>
#include <string.h>
#include <lib$routines.h>

#define MBX_PROT 0xF000
#define MAX_MSG 1024
#define BUF_QUO 60000
#define LIST_END 0
#define check(S) if(!((S)&1)) sys$exit(S)
Force 10 byte reads.
#define READ_SIZE 10

int     main(void)
{

        iosb    ios;
        int     status;

        $DESCRIPTOR(ptable,"LNM$PROCESS_DIRECTORY");
        $DESCRIPTOR(lnm,"LNM$TEMPORARY_MAILBOX");
        char    equiv[ ] = "LNM$GROUP";
        ile3    lnm_items[ ] = {{strlen(equiv),LNM$_STRING,equiv},{LIST_END}};
        $DESCRIPTOR(mbx,"DATA_MBX");
        short   chan;
        char    buffer[MAX_MSG + 1];
        int     efn;
        int     i;

/* Create a logical name to allow the next temporary mailbox's name
   we create to be placed in the group logical name table.
*/
        status = sys$crelnm(0,&ptable,&lnm,0,lnm_items);
        check(status);
/* Create/assign a channel to the listener mailbox. */
        status = sys$crembx(0,&chan,MAX_MSG,BUF_QUO,MBX_PROT,0,&mbx,0,0);
        check(status);
/* Get an available event flag number. */
        status = lib$get_ef(&efn);
        check(status);

        i=1;
/* Read and display until EOF. */
        do
        {
Allow the data to be streamed.
                status = sys$qiow(efn,chan,IO$_READVBLK|IO$M_STREAM,&ios,0,0,
                        buffer,READ_SIZE,0,0,0,0);
                check(status);
                if(ios.iosb$w_status != SS$_ENDOFFILE)
                {
                        check(ios.iosb$w_status);
                        buffer[ios.iosb$w_bcnt] = '\0';
                        printf("Message %08d: %s\n",i,buffer);
                        i++;
                }
        } while(ios.iosb$w_status != SS$_ENDOFFILE);

        return(SS$_NORMAL);
}

$
$ cc mailbox_streamer
$ link mailbox_streamer
The MAILBOX_WRITER program was run at the same time as the mailbox
streamer.  The same data was entered when the program ran.
$ r mailbox_streamer
Note: each line is truncated at 10 bytes, but no data is lost.
Message 00000001: Bruce Elli
Message 00000002: s was here
Message 00000003: Welcome to
Message 00000004:  Mailboxes
Message 00000005:  from BRUD
Message 00000006: EN-OSSG
Message 00000007: We have lo
Message 00000008: t's of gre
Message 00000009: at guys an
Message 00000010: d a great
Message 00000011: Guy on boa
Message 00000012: rd.
$
```

Full Mailboxes

When a mailbox becomes full, two different actions can occur. By default, processes attempting to write to a full mailbox will stall in the RWMBX variation of MWAIT state. It should be possible to delete the process in current versions of OpenVMS. You may want, however, to investigate the cause of the mailbox becoming full to prevent this behavior in the future.

The hang is intended to be a good behavior. The hope is that the mailbox will eventually be read and the process will automatically be released from the stalled RWMBX scheduling state. Indeed, a poorly designed mailbox reader that spends too much time processing data before performing the next mailbox read can cause processes to bounce in and out of RWMBX state. In this case, you would like to tune the reader application. If this is not a possible action, you can consider increasing the buffer quota (BUFQUO) setting on the mailbox.

However, the mailbox reader may be stalled in an involuntary wait state, unable to read the mailbox. It may also be the case that the reader has disappeared from the system entirely, due to some internal failure in the application. Example 8 shows processes stalled in RWMBX wait state.

Example 8. Sample RWMBX Wait States

```
$ show sys/sub
OpenVMS V8.3  on node ALPH40  19-NOV-2006 20:17:26.91  Uptime  56 01:15:55
  Pid    Process Name    State  Pri      I/O       CPU       Page flts  Pages
2040043B DTGREET         LEF      4      814   0 00:00:01.44       590    692  S
20400D55 ELLIS_21769     RWAST    6      165   0 00:00:00.13       241    206  S
$ spawn/nowait r mbx_w
%DCL-S-SPAWNED, process ELLIS_35375 spawned
$ spawn/nowait r mbx_w
%DCL-S-SPAWNED, process ELLIS_36751 spawned
$ spawn/nowait r mbx_w
%DCL-S-SPAWNED, process ELLIS_37285 spawned
$ spawn/nowait r mbx_w
%DCL-S-SPAWNED, process ELLIS_27951 spawned
$ spawn/nowait r mbx_w
%DCL-S-SPAWNED, process ELLIS_57898 spawned
$ spawn/nowait r mbx_w
%DCL-S-SPAWNED, process ELLIS_36551 spawned
$ spawn/nowait r mbx_w
%DCL-S-SPAWNED, process ELLIS_19782 spawned
$ sh sys/sub
OpenVMS V8.3  on node ALPH40  19-NOV-2006 20:17:40.95  Uptime  56 01:16:09
  Pid    Process Name    State  Pri      I/O       CPU       Page flts  Pages
2040043B DTGREET         LEF      4      814   0 00:00:01.44       590    692  S
20400D55 ELLIS_21769     RWAST    6      165   0 00:00:00.15       241    206  S
20400D8C ELLIS_35375     LEF      6       17   0 00:00:00.02       241    206  S
20400D8D ELLIS_36751     LEF      6       18   0 00:00:00.02       241    206  S
20400D8E ELLIS_37285     LEF      6       17   0 00:00:00.01       241    206  S
20400D8F ELLIS_27951     LEF      6       17   0 00:00:00.04       241    206  S
20400D90 ELLIS_57898     LEF      6       19   0 00:00:00.02       241    206  S
20400D91 ELLIS_36551     LEF      6       15   0 00:00:00.02       241    206  S
20400D92 ELLIS_19782     LEF      6       14   0 00:00:00.01       241    206  S
$ spawn/nowait r mbx_w
%DCL-S-SPAWNED, process ELLIS_32782 spawned
$ sh sys/sub
OpenVMS V8.3  on node ALPH40  19-NOV-2006 20:21:25.18  Uptime  56 01:19:53
  Pid    Process Name    State  Pri      I/O       CPU       Page flts  Pages
2040043B DTGREET         LEF      4      814   0 00:00:01.44       590    692  S
20400D55 ELLIS_21769     RWAST    6      165   0 00:00:00.15       241    206  S
20400D8C ELLIS_35375     RWMBX    6      144   0 00:00:00.02       241    206  S
20400D8D ELLIS_36751     RWMBX    6      128   0 00:00:00.02       241    206  S
20400D8E ELLIS_37285     RWMBX    6      143   0 00:00:00.01       241    206  S
20400D8F ELLIS_27951     RWMBX    6      131   0 00:00:00.04       241    206  S
20400D90 ELLIS_57898     RWMBX    6      145   0 00:00:00.02       241    206  S
20400D91 ELLIS_36551     RWMBX    6      143   0 00:00:00.02       241    206  S
20400D92 ELLIS_19782     RWMBX    6      142   0 00:00:00.01       241    206  S
20400D93 ELLIS_32782     RWMBX    6       12   0 00:00:00.01       241    206  S
$
```

It may also be the case that a race condition is entered under a heavy load, such that a system that generally does not have processes stalling in RWAST state starts to see this state show up. The ideal solution would be to locate the cause of the race condition and correct it. In some cases, it is cheaper and faster to simply increase the size of the mailbox (BUFQUO).

Troubleshooting Full Mailbox Problems

The two most common problems associated with mailboxes are probably:

1. Processes stalling in RWMBX variation of MWAIT state due to a full mailbox.
2. Processes stalling in RWAST variation of MWAIT state due to exhaustion of buffered I/O limit (BIOLM). This is most commonly caused by improper use of asynchronous sys$qio calls.

In this article we will address RWMBX issues.

When you find a process in RWMBX state, you will likely first want to know which mailbox the process is attempting to write. In the System Dump Analyzer (SDA) you can get into the context of the target process by issuing a SET PROCESS command. If the process has a channel assigned to one mailbox, the process is pretty straightforward. You would just issue a SHOW PROCESS/CHANNEL command.

If there are several channels assigned, you will need to determine which channel is associated with the call to sys$qio. Once in the context of the stalled process, you can view the parameters passed to the sys$qio system service by examining registers. The channel number of the device is passed as the second parameter. On HP AlphaServer systems, you would examine register R17 to determine the second parameter being passed to sys$qio. On an HP Integrity server system examine R33 to determine the second parameter passed to sys$qio. The contents of the appropriate register will give you the channel number for the full mailbox that the process is attempting to write.

You can next issue a SHOW PROCESS/CHANNEL command to determine which channels are assigned by the process. The full mailbox should have a channel number that matches the hexadecimal value that you obtained from the register.

Once you know the device name, you may want to map it to the logical name associated with the mailbox. The UCB for the mailbox contains a pointer to the logical name associated with the mailbox. You can format this address using a type of LNMB (Logical Name Block).

See Example 9 (AlphaServer) or Example 10 (Itanium server) for an illustration of these steps.

Example 9. Locating the Channel Number for a Write to a Full Mailbox (Alpha)

View one of the processes in RWMBX state.

```
SDA> show summary/process=ELLIS_55216


Current process summary
-----------------------
 Extended Indx Process name    Username     State   Pri PCB/KTB    PHD    Wkset
-- PID -- ---- --------------- ----------- ------- --- -------- -------- ------
 20400D9F 019F ELLIS_55216     ELLIS       RWMBX     6 823D08C0 84B74000    206
```

Set context to the target process.

```
SDA> set process ELLIS_55216
```

Identify the channel associated with the QIO.

```
SDA> examine r17
R17:  00000000.000000D0   "Đ......"
```

Map the channel number to the mailbox device.

```
SDA> show process/channel

Process index: 019F   Name: ELLIS_55216       Extended PID: 20400D9F
--------------------------------------------------------------------



                            Process active channels
                            -----------------------

Channel    CCB     Window     Status    Device/file accessed
-------    ---     ------     ------    --------------------
  0010  7FF7C000  00000000              $1$DGA642:
  0020  7FF7C020  8246F8C0              $1$DGA642:[ELLIS]MBX_W.EXE;7
  0030  7FF7C040  81F5D500              $1$DGA642:[VMS$COMMON.SYSEXE]DCL.EXE;1 (section file)
  0040  7FF7C060  00000000              TNA57:
  0050  7FF7C080  00000000              TNA57:
  0060  7FF7C0A0  81F4EA40              $1$DGA642:[VMS$COMMON.SYSLIB]DCLTABLES.EXE;775 (section file)
  0070  7FF7C0C0  81F4ECC0              $1$DGA642:[VMS$COMMON.SYSLIB]LIBOTS.EXE;1 (section file)
  0080  7FF7C0E0  81F53080              $1$DGA642:[VMS$COMMON.SYSLIB]DECC$SHR_EV56.EXE;1 (section file)
  0090  7FF7C100  81F52900              $1$DGA642:[VMS$COMMON.SYSLIB]DPML$SHR.EXE;1 (section file)
  00A0  7FF7C120  81F51140              $1$DGA642:[VMS$COMMON.SYSLIB]CMA$TIS_SHR.EXE;1 (section file)
  00B0  7FF7C140  81F4EC40              $1$DGA642:[VMS$COMMON.SYSLIB]LIBRTL.EXE;1 (section file)
  00C0  7FF7C160  00000000              TNA57:
  00D0  7FF7C180  00000000  Busy        MBA30202:

  Total number of open channels : 13.
SDA>
```

View the mailbox I/O database information.

```
SDA> show device mba30202

I/O data structures
-------------------
MBA30202                                     MBX                UCB: 821362C0

Device status:   88000010 online,exfunc_supp,iopost_local
Characteristics: 0C150001 rec,shr,avl,mbx,idv,odv
                 00000000
SUD Status       00000000

Owner UIC [000042,000042]  Operation count         0   ORB address    823C7300
      PID        00000000  Error count             0   DDB address    81853780
Class/Type          A0/01  Reference count        10   DDT address    818E3740
Def. buf. size        256  BOFF             00000000   SUD address    8246F6C0
DEVDEPEND        0000037C  Byte count       00000000   CRB address    818537F0
```

The logical name block address is in the "LNM address field".

```
DEVDEPND2        00000000  SVAPTE           00000000   LNM address    85322870
DEVDEPND3        00000000  DEVSTS           00000002   I/O wait queue 82136378
FLCK index             0B
DLCK address     824A7980
Charge PID       00030183

        *** I/O request queue is empty ***
SDA> read sysdef
SDA> form 85322870/typ=lnmb
FFFFFFFF.85322870   LNMB$L_FLINK                             850942B0
FFFFFFFF.85322874   LNMB$L_BLINK                    853211D0
FFFFFFFF.85322878   LNMB$W_SIZE                                  0080
FFFFFFFF.8532287A   LNMB$B_TYPE                                  40
FFFFFFFF.8532287B   LNMB$B_PAD                                 00
FFFFFFFF.8532287C   LNMB$L_ACMODE                      00000003
FFFFFFFF.85322880   LNMB$L_TABLE                               85322A08    LNM+00198
FFFFFFFF.85322884   LNMB$L_LNMX                        853228A0            LNM+00030
FFFFFFFF.85322888   LNMB$L_FLAGS                               00000000
                    LNMB$R_BITS
                    LNMB$R_FLAG_BITS
FFFFFFFF.8532288C   LNMB$L_NAMELEN                     00000009
FFFFFFFF.85322890   LNMB$T_NAME                                      42
```
View the mailbox logical name. The length of 9 identifies the characters for the name. Everything beyond the first 9 characters, in this case, is garbage.
```
SDA> examine 85322890;9
30303430 325F4878 626D5F65 63757242  Bruce_mbxH_20400     FFFFFFFF.85322890
SDA>
```

Example 10. Locating the Channel Number for a Write to a Full Mailbox (IA64)

View one of the processes in RWMBX state.
```
SDA> show summary/proc=ELLIS_56220

Current process summary
-----------------------
 Extended Indx Process name    Username     State   Pri PCB/KTB    PHD    Wkset
-- PID -- ---- --------------- ------------ ------- --- -------- -------- ------
 218004C6 00C6 ELLIS_56220     ELLIS        RWMBX     6 8555BF00 8C12C000    265
```
Set context to the target process.
```
SDA> set proc ELLIS_56220
```
Identify the channel associated with the QIO.
```
SDA> examine r33
R33:  00000000.000000D0   "Ð......."
SDA>
```
Map the channel number to the mailbox device.
```
SDA> show process/channel


Process index: 00C6   Name: ELLIS_56220      Extended PID: 218004C6
--------------------------------------------------------------------



                            Process active channels
                            -----------------------

Channel    CCB     Window     Status    Device/file accessed
-------    ---     ------     ------    -------------------
  0010  7FF26000  00000000              $1$DGA242:
  0020  7FF26020  8555CA80              $1$DGA242:[ELLIS]MBX_W.EXE;2
  0030  7FF26040  853BC840              $1$DGA242:[VMS$COMMON.SYSEXE]DCL.EXE;1 (section file)
  0040  7FF26060  00000000              TNA3:
  0050  7FF26080  00000000              TNA3:
  0060  7FF260A0  853AF9C0              $1$DGA242:[VMS$COMMON.SYSLIB]DCLTABLES.EXE;381 (section file)
  0070  7FF260C0  853AFCC0              $1$DGA242:[VMS$COMMON.SYSLIB]LIBOTS.EXE;1 (section file)
  0080  7FF260E0  853B4140              $1$DGA242:[VMS$COMMON.SYSLIB]DECC$SHR.EXE;1 (section file)
  0090  7FF26100  853B37C0              $1$DGA242:[VMS$COMMON.SYSLIB]DPML$SHR.EXE;1 (section file)
  00A0  7FF26120  853B2040              $1$DGA242:[VMS$COMMON.SYSLIB]CMA$TIS_SHR.EXE;1 (section file)
  00B0  7FF26140  853AFB40              $1$DGA242:[VMS$COMMON.SYSLIB]LIBRTL.EXE;1 (section file)
  00C0  7FF26160  00000000              TNA3:
  00D0  7FF26180  00000000  Busy        MBA6706:
```

```
  Total number of open channels : 13.
SDA>
SDA>
View the mailbox I/O database information.
SDA> show device mba6706

I/O data structures
-------------------
MBA6706                                 MBX              UCB: 85417E80

Device status:   88000010 online,exfunc_supp,iopost_local
Characteristics: 0C150001 rec,shr,avl,mbx,idv,odv
                 00000000
SUD Status       00000000

Owner UIC [000042,000042]  Operation count        0  ORB address    8541E780
      PID        00000000  Error count            0  DDB address    841ADB80
Class/Type          A0/01  Reference count       10  DDT address    84248B40
Def. buf. size        256  BOFF            00000000  SUD address    85265280
DEVDEPEND        0000037C  Byte count      00000000  CRB address    841ADBF0
The logical name block address is in the "LNM address field".
DEVDEPND2        00000000  SVAPTE          00000000  LNM address    8D1A65E0
DEVDEPND3        00000000  DEVSTS          00000002  I/O wait queue 85417FB0
FLCK index             0B
DLCK address     85519CC0
Charge PID       000100BC

        *** I/O request queue is empty ***
SDA>
SDA> format  8D1A65E0/type=lnmb
FFFFFFFF.8D1A65E0   LNMB$L_FLINK                              8CDCBCF0
FFFFFFFF.8D1A65E4   LNMB$L_BLINK                     8D1A9EF0
FFFFFFFF.8D1A65E8   LNMB$W_SIZE                                   0070
FFFFFFFF.8D1A65EA   LNMB$B_TYPE                                 40
FFFFFFFF.8D1A65EB   LNMB$B_PAD                                00
FFFFFFFF.8D1A65EC   LNMB$L_ACMODE                    00000003
FFFFFFFF.8D1A65F0   LNMB$L_TABLE                              8D267A98
FFFFFFFF.8D1A65F4   LNMB$L_LNMX                      8D1A6610            LNM+00030
FFFFFFFF.8D1A65F8   LNMB$L_FLAGS                              00000000
                    LNMB$R_BITS
                    LNMB$R_FLAG_BITS
View the mailbox logical name.  The length of 9 identifies the characters for
the name.  Everything beyond the first 9 characters, in this case, is garbage.
FFFFFFFF.8D1A65FC   LNMB$L_NAMELEN                   00000009
FFFFFFFF.8D1A6600   LNMB$T_NAME                                     42
SDA> examine 8D1A6600;9
00000000 00000078 626D5F65 63757242  Bruce_mbx.......    FFFFFFFF.8D1A6600
SDA>
```

The UCB for a mailbox device has fields that are of specific interest when troubleshooting full mailboxes. The first two longwords in a mailbox UCB (UCBL_MB_MSGQFL / UCBL_MB_MSGQBL) contain the message queue forward and backward links. You can walk these links and view the messages queued to the mailbox. The symbol table file SYSDEF.STB contains MBOX symbol definitions that help you interpret these fields. These symbol definitions are not available in older versions of OpenVMS.

Fields that track read (UCB$L_MB_R_AST) and write (UCB$L_MB_W_AST) attention ASTs are after the message queues and size and type fields. If the mailbox driver is currently servicing an I/O request, the field UCB$L_IRP contains a pointer to the IRP.

Immediately after the base UCB, the mailbox driver maintains:

- Counts of read (UCB$L_MB_READERREFC) and write (UCB$L_MB_WRITERREFC) channels that have been assigned to the mailbox.
- A reader queue for outstanding reads that have been queued to the mailbox. (UCB$L_MB_READQFL/UCB$L_MB_READQBL)
- Queues for mailbox waits for write/read channels to be assigned. (UCB$L_MB_WRITERWAITQFL/UCB$L_MB_WRITERWAITQBL and UCB$L_MB_READERWAITQFL/ UCB$L_MB_READERWAITQBL)
- Queues for mailbox waits for all write/read channels to be deassigned. (UCB$L_MB_NOWRITERWAITQFL/ UCB$L_MB_NOWRITERWAITQBL and UCB$L_MB_NOREADERWAITQFL/ UCB$L_MB_NOREADERWAITQBL)
- A list of ACBs for process notification that mailbox room is available. (UCB$L_MB_ROOM_NOTIFY)
- A pointer to the logical name block for the mailbox. (UCB$L_LOGADR)
- The available mailbox size. (UCB$L_MB_BUFQUO)
- The initial mailbox size (Initial BUFQUO). (UCB$L_MB_INIQUO)

After you issue a SHOW DEVICE command on the mailbox a symbol named UCB contains the address of the UCB for the mailbox. To view relative elements on the message queue, you can issue FORMAT @UCB commands. For each "@" character in the command you move forward to that relative message, e.g., FORMAT @@@@UCB formats the fourth message in the message queue. To determine the number of messages queued to the mailbox, issue the command VALIDATE QUEUE UCB. Example 11 illustrates walking the message queue for a given mailbox. The example works the same way on AlphaServer and Integrity server systems.

Example 11. Walking Mailbox Message Queues

```
View the first message on the message queue.
SDA> form @ucb
FFFFFFFF.854CFE80  MBOX_MSG$L_FLINK                    8541FC80
                   MBOX_MSG$PS_ADDR
FFFFFFFF.854CFE84  MBOX_MSG$L_BLINK           85417E80            UCB
                   MBOX_MSG$PS_UVA32
FFFFFFFF.854CFE88  MBOX_MSG$W_MBZ                          0000
FFFFFFFF.854CFE8A  MBOX_MSG$B_TYPE                       79
FFFFFFFF.854CFE8B  MBOX_MSG$B_SUBTYPE                  53
FFFFFFFF.854CFE8C  MBOX_MSG$L_FUNCTION        00000020
FFFFFFFF.854CFE90  MBOX_MSG$PQ_UVA64          00000000.DEAD0001
FFFFFFFF.854CFE98  MBOX_MSG$L_SIZE                     000000C0
FFFFFFFF.854CFE9C  MBOX_MSG$L_IRP             85262E80
FFFFFFFF.854CFEA0  MBOX_MSG$L_NOREADERWAITQFL          00000000
FFFFFFFF.854CFEA4  MBOX_MSG$L_NOREADERWAITQBL 00000000
This is the internal process ID of the process that issued this message.
FFFFFFFF.854CFEA8  MBOX_MSG$L_PID                      000100BE   SYS$K_VERSION_16+0007E
FFFFFFFF.854CFEAC  MBOX_MSG$L_DATASTART       854CFEB8
FFFFFFFF.854CFEB0  MBOX_MSG$W_DATASIZE                     0070
FFFFFFFF.854CFEB2   MBOX_MSG$W_BUFQUOCHARGE                  0001
FFFFFFFF.854CFEB4   MBOX_MSG$L_THREAD_PID           218004BE
                    MBOX_MSG$C_LENGTH
```

Here is the message data.

```
FFFFFFFF.854CFEB8   MBOX_MSG$R_DATA                 00005F53.494C4C45
SDA> examine 854CFEB8;70
65623430 30383132 00005F53 494C4C45  ELLIS_..218004be     FFFFFFFF.854CFEB8
35383338 33323732 39303134 33303242  B203410927238385     FFFFFFFF.854CFEC8
39323930 33323936 37323330 35363936  6965032769230929     FFFFFFFF.854CFED8
33363534 31303532 37383732 39363138  8169278725014563     FFFFFFFF.854CFEE8
31383330 33363734 33323330 35383738  8785032347630381     FFFFFFFF.854CFEF8
35363138 35303730 31323930 33303734  4703092107058165     FFFFFFFF.854CFF08
31303136 33363734 33303134 39383738  8789410347636101     FFFFFFFF.854CFF18
SDA>
```

The next message in the queue.

```
SDA> form @@UCB
FFFFFFFF.8541FC80   MBOX_MSG$L_FLINK                         854CE480
                    MBOX_MSG$PS_ADDR
FFFFFFFF.8541FC84   MBOX_MSG$L_BLINK                854CFE80
                    MBOX_MSG$PS_UVA32
FFFFFFFF.8541FC88   MBOX_MSG$W_MBZ                               0000
FFFFFFFF.8541FC8A   MBOX_MSG$B_TYPE                            79
FFFFFFFF.8541FC8B   MBOX_MSG$B_SUBTYPE                       53
FFFFFFFF.8541FC8C   MBOX_MSG$L_FUNCTION             00000020
FFFFFFFF.8541FC90   MBOX_MSG$PQ_UVA64               00000000.DEAD0001
FFFFFFFF.8541FC98   MBOX_MSG$L_SIZE                          000000C0
FFFFFFFF.8541FC9C   MBOX_MSG$L_IRP                  85261E00
FFFFFFFF.8541FCA0   MBOX_MSG$L_NOREADERWAITQFL               00000000
FFFFFFFF.8541FCA4   MBOX_MSG$L_NOREADERWAITQBL      00000000
FFFFFFFF.8541FCA8   MBOX_MSG$L_PID                           000100BF    SYS$K_V
ERSION_16+0007F
FFFFFFFF.8541FCAC   MBOX_MSG$L_DATASTART            8541FCB8
FFFFFFFF.8541FCB0   MBOX_MSG$W_DATASIZE                          0070
FFFFFFFF.8541FCB2   MBOX_MSG$W_BUFQUOCHARGE                  0001
FFFFFFFF.8541FCB4   MBOX_MSG$L_THREAD_PID           218004BF
                    MBOX_MSG$C_LENGTH

FFFFFFFF.8541FCB8   MBOX_MSG$R_DATA                 00005F53.494C4C45
```

The third message...

```
SDA> form @@@UCB
FFFFFFFF.854CE480   MBOX_MSG$L_FLINK                         854C9140
                    MBOX_MSG$PS_ADDR
FFFFFFFF.854CE484   MBOX_MSG$L_BLINK                8541FC80
                    MBOX_MSG$PS_UVA32
FFFFFFFF.854CE488   MBOX_MSG$W_MBZ                               0000
FFFFFFFF.854CE48A   MBOX_MSG$B_TYPE                            79
FFFFFFFF.854CE48B   MBOX_MSG$B_SUBTYPE                       53
FFFFFFFF.854CE48C   MBOX_MSG$L_FUNCTION             00000020
FFFFFFFF.854CE490   MBOX_MSG$PQ_UVA64               00000000.DEAD0001
FFFFFFFF.854CE498   MBOX_MSG$L_SIZE                          000000C0
FFFFFFFF.854CE49C   MBOX_MSG$L_IRP                  854D1640
FFFFFFFF.854CE4A0   MBOX_MSG$L_NOREADERWAITQFL               00000000
FFFFFFFF.854CE4A4   MBOX_MSG$L_NOREADERWAITQBL      00000000
FFFFFFFF.854CE4A8   MBOX_MSG$L_PID                           000100C0    SYS$K_V
ERSION_16+00080
FFFFFFFF.854CE4AC   MBOX_MSG$L_DATASTART            854CE4B8
FFFFFFFF.854CE4B0   MBOX_MSG$W_DATASIZE                          0070
FFFFFFFF.854CE4B2   MBOX_MSG$W_BUFQUOCHARGE                  0001
FFFFFFFF.854CE4B4   MBOX_MSG$L_THREAD_PID           218004C0
```

```
                    MBOX_MSG$C_LENGTH

FFFFFFFF.854CE4B8   MBOX_MSG$R_DATA                 00005F53.494C4C45
Determine the number of messages queued to the mailbox.
SDA> validate queue ucb
Queue is complete, total of 892 elements in the queue
SDA>
```

Once you have determined how many messages are in the queue and which processes are sending them, you will need to determine what happened to the mailbox reader. Is it hung in a resource wait state? Has it encountered a race condition that caused it to ignore the mailbox? Has the process died for some reason?

To identify where in the code the process has stalled, you can view call frames and walk back to the source of the call. Doing so requires that you have access to link maps and machine code listings for the program that the hung process was running.

On HP AlphaServer systems, the return address of the caller is stored in r26 when the sys$qio code is entered. If a sys$qiow was called, that, in turn, made the call to sys$qio; you will need to view call frames to locate the caller of sys$qiow. Once you know the return PC, you can take it to the map file for the program and find the program section that contains the given PC. You would then subtract the base address of the containing program section to determine the location counter for the machine code that contains the return address from the call. The location counter can be taken to the listing file to locate the machine code instruction for the return from the call.

From the return instruction, you can back up one instruction at a time in the listing file, looking for a source line number. In a 132 column display, the source line number will be all the way to the right and will have a ";" prefix in front of the source line number. This will get you to the source code and you can determine what is happening in the context of the program. Example 12 illustrates mapping the call back to the source in the sys$qio case on an AlphaServer system. Example 13 does the same for the sys$qiow case on an AlphaServer system.

Example 12. Mapping the Return PC to Source for a Process in RWMBX (sys$qio case on AlphaServer)

```
SDA> sh summary


Current process summary
-----------------------
 Extended Indx Process name    Username     State   Pri PCB/KTB    PHD    Wkset
-- PID -- ---- --------------- ------------ ------- --- -------- -------- ------
 20400401 0001 SWAPPER         SYSTEM       HIB      16 818E5DC8 818E5800      0
 20400407 0007 CLUSTER_SERVER  SYSTEM       HIB      13 81DEE600 84B2C000    113
...
 2040043B 003B DTGREET         SYSTEM       LEF       4 81DB9640 84B2A000    692
 204008B8 00B8 TCPIP$BOOTP_1   TCPIP$BOOTP  LEF      10 8223D1C0 84B62000    280
 20400DAD 01AD _TNA58:         ELLIS        CUR 002   6 8222F780 84B54000    659
 20400DC5 01C5 ELLIS_14353     ELLIS        RWMBX     6 822EFC40 84B32000    206
 20400DC6 01C6 ELLIS_29302     ELLIS        RWMBX     6 821D79C0 84B52000    206
 20400DC7 01C7 ELLIS_7240      ELLIS        RWMBX     6 822E8200 84B58000    206
 20400DC8 01C8 ELLIS_61712     ELLIS        RWMBX     6 821B1140 84B60000    206
 20400DC9 01C9 ELLIS_31360     ELLIS        RWMBX     6 821D6CC0 84B64000    206
 20400DCA 01CA ELLIS_60365     ELLIS        RWMBX     6 823D08C0 84B66000    210
SDA> set process/index=1c8
SDA> read/executive
```

View the call frames looking for a call to sys$qiow.

```
SDA> show call/summary

Call Frame Summary
------------------
```

There is no call frame for sys$qiow. Therefore, we will need to look for the return PC in r26.

```
     Frame Type          Frame Address        Return PC         Procedure Entry
--------------------    -----------------    -----------------    -----------------
Stack Frame             00000000.7AE09990    00000000.00020064    00000000.000200A0
SYS$K_VERSION_08+00080
Stack Frame             00000000.7AE09AA0    FFFFFFFF.80385CE4    00000000.00020000
SYS$K_VERSION_06
Stack Frame             00000000.7AE09B30    00000000.7AF6C058    FFFFFFFF.80385B50
SYS$IMGSTA_C
Stack Frame             00000000.7AE09BB0    00000000.7AF6BE88    00000000.7AF6BE9C  DCL+81E9C
Cannot display further call frames (Bottom of stack)
SDA> examine r26
```

This is the return PC.

```
R26:  00000000.000202A4   "¤......."
```

Verify that the instruction preceding the return PC is a jump to subroutine (JSR).

```
SDA> examine/inst 202a4-4
SYS$K_VERSION_08+00280:         JSR            R26,(R26)
SDA>
```

Determine the image that the process is running.

```
SDA> show summary/image/process= ELLIS_61712

Current process summary
-----------------------
 Extended Indx Process name    Username     State   Pri PCB/KTB    PHD    Wkset
-- PID -- ---- --------------- ------------ ------- --- -------- -------- ------
 20400DC8 01C8 ELLIS_61712     ELLIS        RWMBX     6 821B1140 84B60000    210
          $1$DGA642:[ELLIS]MBX_W.EXE;9
SDA> EXIT
$
```

View the map file, looking for the program section containing the PC.

```
$ type mbx_w.map
                                                  19-NOV-2006 22:37       Linker A13-03                  Page
1

                                   +-----------------------+
                                   ! Object Module Synopsis !
                                   +-----------------------+

Module Name     Ident            Bytes      File  Creation Date      Creator
-----------     -----            -----      -----  ------------      -------
MBX_W            V1.0              2377 SYS$SYSDEVICE:[ELLIS]MBX_W.OBJ;7     19-NOV-2006 22:37  Compaq C V6.5-001

                                  +-------------------------+
                                  ! Program Section Synopsis !
                                  +-------------------------+

Psect Name      Module Name       Base     End           Length           Align                 Attributes
----------      -----------       ----     ---           ------           -----                 ----------
$LINK$                          00010000 000101BF 000001C0 (       448.) OCTA4 NOPIC,CON,REL,LCL,NOSHR,NOEXE,NOWRT,NOVEC,  MOD
                MBX_W           00010000 000101BF 000001C0 (       448.) OCTA4
$LITERAL$                       000101C0 000101E4 00000025 (        37.) OCTA4   PIC,CON,REL,LCL,  SHR,NOEXE,NOWRT,NOVEC,  MOD
                MBX_W           000101C0 000101E4 00000025 (        37.) OCTA4
$READONLY$                      000101F0 000101FF 00000010 (        16.) OCTA4   PIC,CON,REL,LCL,  SHR,NOEXE,NOWRT,NOVEC,  MOD
                MBX_W           000101F0 000101FF 00000010 (        16.) OCTA4
CR_VALS                         00010200 0001020F 00000010 (        16.) OCTA4 NOPIC,OVR,REL,GBL,NOSHR,NOEXE,NOWRT,NOVEC,  MOD
```

```
                MBX_W             00010200 0001020F 00000010 (         16.) OCTA4
```

Here is the Program Section containing the return PC. The base of the program section is at 20000, so the offset into module MBX_W for the point of the call is 202a0.

```
$CODE$                            00020000 0002072B 0000072C (       1836.) OCTA4   PIC,CON,REL,LCL,  SHR,  EXE,NOWRT,NOVEC,  MOD
                MBX_W             00020000 0002072B 0000072C (       1836.) OCTA4
$BSS$                             00030000 00030017 00000018 (         24.) OCTA4 NOPIC,CON,REL,LCL,NOSHR,NOEXE,  WRT,NOVEC,NOMOD
                MBX_W             00030000 00030017 00000018 (         24.) OCTA4
...
$ edit mbx_w.lis
```

First search for the location counter 000002a0. We find it below, then back to the source line number.

```
D3400089    0258            BSR     R26, GEN_BUFF                                   ; 021837
```

Source line number 21835 should be the location that the call to sys$qio was made.

```
A7420078    0258            LDQ     R26, 120(R2)                                    ; 021835
47E41410    0260            MOV     32, R16
47E70411    0264            MOV     R7, R17
47E61412    0268            MOV     48, R18
B41E0000    026C            STQ     R0, (SP)
47EE1400    0270            MOV     112, R0
B7FE0010    0274            STQ     R31, 16(SP)
227D0028    0278            LDA     R19, mb_ios     ; R19, 40(FP)
B41E0008    027C            STQ     R0, 8(SP)
B7FE0018    0280            STQ     R31, 24(SP)
47FF0414    0284            CLR     R20
47FF0415    0288            CLR     R21
B7FE0020    028C            STQ     R31, 32(SP)
B7FE0028    0290            STQ     R31, 40(SP)
47E19419    0294            MOV     12, R2
A7620080    0298            LDQ     R27, 128(R2)
2FFE0000    029C            UNOP
6B5A4000    02A0            JSR     R26, SYS$QIO      ; R26, R26
A742FFA8    02A4            LDQ     R26, -88(R2)                                    ; 021838
F0000004    02A8            BLBS    R0, L$21
47E00410    02AC            MOV     R0, status  ; R0, R16                           ; 021835
```

Now we search for the source line.

```
     1   21823 /* Assign a channel to the mailbox. */
     1   21824        status = sys$crembx(0,&mbx_chan,0,0,0,0,&mbx,0,0);
     1   21825        check(status);
     1   21826
     1   21827 /* Write messages to the mailbox. */
     1   21828        for(i=0;i<500;i++)
     2   21829        {
     2   21830
     2   21831               stall = ten_ms * ((rand()%300)+1);
     2   21832               status = sys$setimr(TEFN,&stall,0,0,0);
     2   21833               check(status);
     2   21834               sys$waitfr(TEFN);
```

Here is the point of the call.

```
     2   21835               status = sys$qio(MEFN,mbx_chan,IO$_WRITEVBLK,
     2   21836                      &mb_ios,0,0,
     2   21837                      gen_buff(&mrec,&id,count++),sizeof(mrec),0,0,0,0);
     2   21838               check(status);
     2   21839               //check(mb_ios.iosb$w_status);
     1   21840        }
     1   21841               stall = ten_ms * 300*1000*100;
```

Example 13. Locating the Return PC for a Process in RWMBX (sys$qiow case on AlphaServer)

In this case, the call to sys$qiow does show up in the call frames. Once we have the return PC, the steps are the same as in example 12.

```
SDA> set proc/index=1ca
SDA> show call/summary

Call Frame Summary
------------------

     Frame Type          Frame Address        Return PC         Procedure Entry
--------------------    -----------------    -----------------    -----------------
Stack Frame            00000000.7AE09930    00000000.000202A4    FFFFFFFF.80114BD0  SYS$QIOW_C
Stack Frame            00000000.7AE09990    00000000.00020064    00000000.000200A0  SYS$K_VERSION_08+00080
Stack Frame            00000000.7AE09AA0    FFFFFFFF.80385CE4    00000000.00020000  SYS$K_VERSION_06
Stack Frame            00000000.7AE09B30    00000000.7AF6C058    FFFFFFFF.80385B50  SYS$IMGSTA_C
Stack Frame            00000000.7AE09BB0    00000000.7AF6BE88    00000000.7AF6BE9C  DCL+81E9C
Cannot display further call frames (Bottom of stack)
SDA>
```

Full mailboxes should be rare in a well-designed application. Hopefully, the steps above will help you out in the rare case that you need to troubleshoot an RWMBX hang.

Designing Applications that Operate Asynchronously Using Mailboxes

For some, a picture is worth a thousand words. For others, seeing code in a complete application helps clarify the concept. The following example illustrates most of the concepts described in this article.

It is rare that an application uses mailboxes for the sole purpose of doing mailbox communication. To illustrate operating asynchronously in an OpenVMS environment, we designed a series of functions and programs to sample the Program Counter, the Buffered and Direct I/O counts for any given application. This data, along with a time stamp, will be logged to a file.

The data will be captured by an Asynchronous System Trap (AST) routine that will be called based on timer expiration. The data we are capturing could be used to profile the performance characteristics of any application. This specific data is not as relevant as the design considerations. Note that the same approach could be used to sample other forms of real-time data.

This example illustrates:

1. Requesting timers
2. AST routines
3. Local Event Flags
4. The "I/O" status block
5. Getting Job/Process information
6. Obtaining a time stamp
7. Using item list entries (ile3 structures)
8. Processing Mailboxes
9. Process creation
10. Creating Mailboxes
11. QIO Interface to the mailbox

From a design perspective, we attempt to maintain data encapsulation through the use of structures that describe the context of operations, such as samples, files, I/O, etc. The structures are passed as parameters to procedures and external/common storage is avoided. This method improves the ability to debug, maintain, and extend the application.

We attempt to minimize "noise" and "drift" in the main sampler process by creating a background process that will:

- Create a temporary mailbox for communication with the parent process.
- Create the log file, whose name is passed from the parent process.
- Accept samples from the mailbox and write them to the log file, until an EOF is sent from the parent.

All mailbox I/O is processed asynchronously.

The file is synched in the background by the child process and the drift is reduced on the samples.

You may want to be able to view the data in real-time. The PC_LOGGER is designed to write the samples to a "listener" mailbox. This mailbox can be read by a listener process that may be logged in on another terminal session. The listener can then display the data in real-time.

The listener process will be logged in separately from the process in which the sampler is being run. Therefore, the mailbox logical name presents a problem. Normally, temporary mailbox names are entered into the job logical name table.

In the listener and the logger the logical name LNM$TEMPORARY_MAILBOX is equated to LNM$GROUP to make the mailbox logical name visible to other processes in the UIC group. The logical name is placed in the logical name table LNM$PROCESS_DIRECTORY.
Since the logical name is created in user mode, it goes away when the images run down, so as to not impact other images run by this process.

It is important to note that the logical name is created AFTER the logger mailbox is created/channel assigned. So, the logger mailbox logical name still goes into the job logical name table.

The listener mailbox is implemented as write-only by the logger and read-only by the listener. This method allows us to simply send the message to the mailbox from the logger. If there is no reader (listener), the mailbox write completes immediately with a status of SS$_NOREADER. When this status is received, we "shrug our shoulders" and try again next time. Similarly, the listener will abort if there is no writer.

When we are done the application design looks like figure 5.

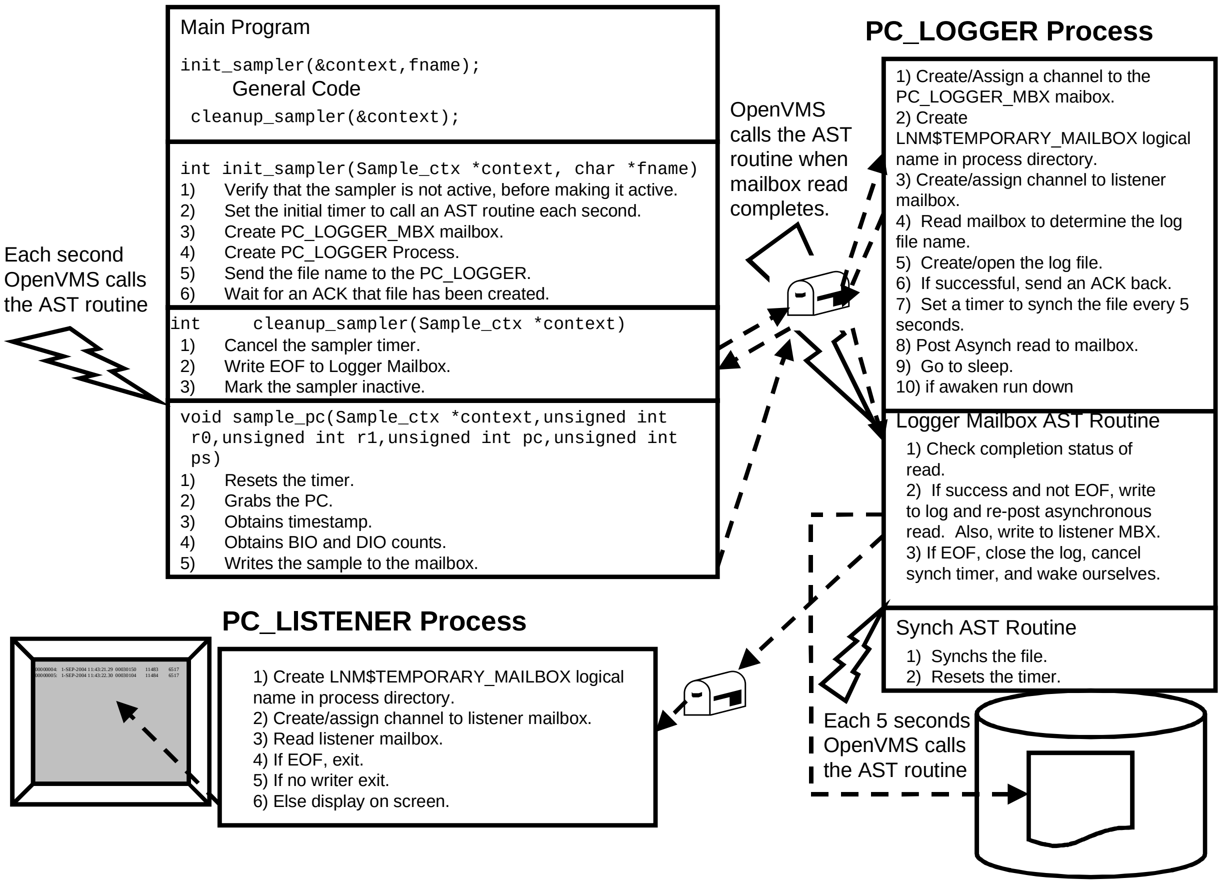
Main Program
init_sampler(&context,fname);
General Code
cleanup_sampler(&context);
int init_sampler(Sample_ctx *context, char *fname)
1) Verify that the sampler is not active, before making it active.
2) Set the initial timer to call an AST routine each second.
3) Create PC_LOGGER_MBX mailbox.
4) Create PC_LOGGER Process.
5) Send the file name to the PC_LOGGER.
6) Wait for an ACK that file has been created.
int cleanup_sampler(Sample_ctx *context)
1) Cancel the sampler timer.
2) Write EOF to Logger Mailbox.
3) Mark the sampler inactive.
void sample_pc(Sample_ctx *context,unsigned int r0,unsigned int r1,unsigned int pc,unsigned int ps)
1) Resets the timer.
2) Grabs the PC.
3) Obtains timestamp.
4) Obtains BIO and DIO counts.
5) Writes the sample to the mailbox.
Each second OpenVMS calls the AST routine
OpenVMS calls the AST routine when mailbox read completes.
PC_LOGGER Process
1) Create/Assign a channel to the PC_LOGGER_MBX maibox.
2) Create LNM$TEMPORARY_MAILBOX logical name in process directory.
3) Create/assign channel to listener mailbox.
4) Read mailbox to determine the log file name.
5) Create/open the log file.
6) If successful, send an ACK back.
7) Set a timer to synch the file every 5 seconds.
8) Post Asynch read to mailbox.
9) Go to sleep.
10) if awaken run down
Logger Mailbox AST Routine
1) Check completion status of read.
2) If success and not EOF, write to log and re-post asynchronous read. Also, write to listener MBX.
3) If EOF, close the log, cancel synch timer, and wake ourselves.
Synch AST Routine
1) Synchs the file.
2) Resets the timer.
Each 5 seconds OpenVMS calls the AST routine
PC_LISTENER Process
1) Create LNM$TEMPORARY_MAILBOX logical name in process directory.
2) Create/assign channel to listener mailbox.
3) Read listener mailbox.
4) If EOF, exit.
5) If no writer exit.
6) Else display on screen.

Figure 5. PC Sampler "System Design"

Example 14. The PC_SAMPLER Header File

```
$ type pc_sampler2.h
#include <stdio.h>
/*
        Context for our PC sampler.
        File pointer for where to write the data.
        Delta time for our sampling interval.
*/

typedef struct pc_sample_ctx
{
        __int64 delta;
        int     c_pid;
        int     sample_no;
        short   mbx_chan;
} Sample_ctx;

/* Profile data */
typedef struct sample_data
{
        __int64 time_stamp;
        int     pc;
        int     bio;
        int     dio;
        int     sample_no;
} Samp;
#define CPU_TIMER 1
int     init_sampler(Sample_ctx *,char *);
void    sample_pc(Sample_ctx *, unsigned int, unsigned int,
                        unsigned int, unsigned int);
int     cleanup_sampler(Sample_ctx *);

$
$
```

Example 15. PC Sampler Test and Stub Programs

```
$ type pc_tester2.c
#include <stdio.h>
#include <stdlib.h>
#include "pc_sampler2.h"
void    stub(void);
int     main(void)
{
        int     i;
        Sample_ctx ctx;

        init_sampler(&ctx,"sample.data");


        for(i=0;i<1000000000;i++)
        {
                stub();
        }

        cleanup_sampler(&ctx);
}
$
$ type stub.c
void    stub(void) {;}
$
```

Example 16. PC Sampler Routines

```
$ type pc_sampler3.c
/* Set of routines to sample program counters at
        1 second (CPU time) intervals and send the PC and time of
        sample to a logger process created by this routine.
*/
#include "pc_sampler2.h"
#include <starlet.h>
#include <stdio.h>
#include <ssdef.h>
#include <iodef.h>
#include <iosbdef.h>
#include <string.h>
#include <descrip.h>
```

```
#include <rmsdef.h>
#include <iledef.h>
#include <jpidef.h>
#define JPI_LISTEND 0
#define check(S) if(!((S)&1)) sys$exit(S)

/* Flag to indicate that the sample is active. */
static  int     active = 0;
/* Sample interval is a constant 1 second. */
static const __int64 sample_interval = -10000000;
#define BASE_PRI 4
#define MBX_EFN 32
#define ACK_SIZE 4
#define TERM_MAX 255

/*
        Initialize the PC Sampler:
        1)      Verify that the sampler is not active, before making it active
        2)      Set the initial timer.
        3)      Create a mailbox.
        4)      Create a child process to log messages.
*/
int     init_sampler(Sample_ctx *context, char *fname)
{
        int     status;
        $DESCRIPTOR(mbx_name,"PC_LOGGER_MBX");
        $DESCRIPTOR(image,"PC_LOGGER2");
        char    ack_buffer[ACK_SIZE];
        iosb    ios;
        char    terminal[TERM_MAX+1];
        $DESCRIPTOR(term,terminal);
        ile3    term_list[] = {{TERM_MAX,JPI$_TERMINAL,terminal,
                        &term.dsc$w_length},{0,0}};
/* If the sampler is already active, return a failure status,
        else make it active.
*/
        if(!active)
        {
                active = 1;
        }
        else
        {
                fprintf(stderr,"Sampler is already active.\n");
        }
/*
        Create the mailbox for communications with the child.
*/
        status = sys$crembx(0,&context->mbx_chan,0,0,0,0,&mbx_name,0,0);
        check(status);

/* Call SYS$GETJPI to obtain our terminal name. */
        status = sys$getjpiw(0,0,0,term_list,&ios,0,0);
        check(status);
        check(ios.iosb$w_status);

/* Create the child logger process. */
        status = sys$creprc(&context->c_pid,&image,0,&term,&term,
                        0,0,&image,BASE_PRI,0,0,0,0,0,0);
        check(status);
/* Send the file name to the child. */
```

```
        status = sys$qiow(MBX_EFN,context->mbx_chan,IO$_WRITEVBLK,&ios,
                        0,0,fname,strlen(fname),0,0,0,0);
        check(status);
        check(ios.iosb$w_status);
/* Read the same mailbox for an acknowledgment that the file was created
   properly.
*/
        status = sys$qiow(MBX_EFN,context->mbx_chan,IO$_READVBLK,&ios,
                        0,0,ack_buffer,ACK_SIZE,0,0,0,0);
        check(status);
        check(ios.iosb$w_status);
        ack_buffer[ios.iosb$w_bcnt] = '\0';

/* Make sure the child created the log file properly.  If not, return
   error.
*/
        if(strcmp(ack_buffer,"ACK") != 0)
        {
                return(RMS$_FNF);
        }
/* Save collection interval in context block. */
        context->delta = sample_interval;
        context->sample_no = 0;
/* Set a timer for collections. */
        status = sys$setimr(0,&context->delta,sample_pc,context,CPU_TIMER);
        check(status);
}

/***********************************
        Sampler AST Routine.
        1)      Grabs the PC.
        2)      Writes the PC and a timestamp to collection log process
                through mailbox.
        3)      Resets the timer.
***********************************/
void    sample_pc(Sample_ctx *context,unsigned int r0,unsigned int r1,
                        unsigned int pc,unsigned int ps)
{
        int     wrt_cnt;
        static Samp     sample;
        int     status;
        ile3    jpi_items[] = {
                                {sizeof(sample.bio),JPI$_BUFIO,
                                 &sample.bio},
                                {sizeof(sample.dio),JPI$_DIRIO,
                                 &sample.dio},
                                {0,JPI_LISTEND}
                              };
        iosb    ios;

/* Reset the timer. */
        status = sys$setimr(0,&context->delta,sample_pc,context,CPU_TIMER);
        check(status);
/* Save the sample PC. */
        sample.pc = pc;
/* Obtain a timestamp. */
        status = sys$gettim(&sample.time_stamp);
        check(status);
/* Call SYS$GETJPI to obtain our buffered and direct I/O counts. */
        status = sys$getjpiw(0,0,0,jpi_items,&ios,0,0);
        check(status);
        check(ios.iosb$w_status);

/* Update and copy the sample number. */
        sample.sample_no = ++(context->sample_no);
/* Write our collection data to the log file. */
        status = sys$qiow(MBX_EFN,context->mbx_chan,IO$_WRITEVBLK|IO$M_NOW,
                        &ios,0,0,
                        &sample,sizeof(sample),0,0,0,0);
        check(status);
        check(ios.iosb$w_status);
}
/***********************************
        Cleanup the PC sampler by:
        1)      Cancelling the sampler timer.
        2)      Marking the sampler inactive.
        3)      Sending EOF status to the child.
***********************************/
int     cleanup_sampler(Sample_ctx *context)
{
        int     status;
        iosb    ios;
/* Cancel the timer. */
        status = sys$cantim(context,0);
        check(status);
/* Notify the logger to close the file. */
        status = sys$qiow(MBX_EFN,context->mbx_chan,IO$_WRITEOF,&ios,
                        0,0,0,0,0,0,0,0);
        check(status);
        check(ios.iosb$w_status);

/* Mark the sampler inactive. */
        active = 0;
        return(status);
}
$
```

Example 17. PC_LOGGER Header File

```
Header file for the PC Sample logger.
$ type pc_logger.h
#include <stdio.h>
#include "pc_sampler2.h"
typedef struct mbx_context
{
        FILE *fp;
        Samp    *samp_buffer;
        iosb    ios;
        short chan;
} Mbx_ctx;
$
```

Example 18. PC_LOGGER Code

```
$ type pc_logger2.c
/*
        Program to accept PC_Sample data and write it
        to a log file.
        The logger will send the data to a listener.
*/
#include <starlet.h>
#include <iodef.h>
#include <stdio.h>
#include <iosbdef.h>
#include <descrip.h>
#include <iledef.h>
#include <lnmdef.h>
#include <cmbdef.h>
#include <string.h>
#include "pc_logger2.h"
#include <ssdef.h>
#define check(S) if(!((S)&1)) sys$exit(S)
#define LIST_END 0
#define MAX_FNAME 255
void    synch_ast(Mbx_ctx *);
void    mbx_ast(Mbx_ctx *);

int     main(void)
{
        int     status;
        Mbx_ctx ctx;
        Samp    sample;
        char    fname[MAX_FNAME+1];
        char    nak[] = "NAK";
        char    ack[] = "ACK";
        char    *msg;
        __int64 synch_time = (__int64) -50000000;
        $DESCRIPTOR(mbx_name,"PC_LOGGER_MBX");
/* Descriptors to allow temporary mailbox names to be placed in the
   group logical name table.
*/
        $DESCRIPTOR(ptable,"LNM$PROCESS_DIRECTORY");
        $DESCRIPTOR(lnm,"LNM$TEMPORARY_MAILBOX");
        char    equiv[ ] = "LNM$GROUP";
        ile3    lnm_items[ ] = {{strlen(equiv),LNM$_STRING,equiv},{LIST_END}};
        $DESCRIPTOR(l_mbx,"PC_LISTENER_MBX");

/* Create /assign a channel to a temporary mailbox for log data. */
        status = sys$crembx(0,&ctx.chan,0,0,0,0,&mbx_name,0,0);
        check(status);

/* Read the mailbox to determine the target (log) file name. */
        status = sys$qiow(0,ctx.chan,IO$_READVBLK,&ctx.ios,0,0,
                        fname,MAX_FNAME,0,0,0,0);
        check(status);
        check(ctx.ios.iosb$w_status);
        fname[ctx.ios.iosb$w_bcnt] = '\0';
/* Open/create the file. */
        ctx.fp = fopen(fname,"w");


/* Send an ACK/NAK dependent on the creation status. */
```

```
        if(!ctx.fp)
        {
                msg = nak;
        }
        else
        {
                msg = ack;
        }
        status = sys$qiow(0,ctx.chan,IO$_WRITEVBLK,&ctx.ios,0,0,
                        msg,strlen(msg),0,0,0,0);
        check(status);
        check(ctx.ios.iosb$w_status);

/* Create a logical name to allow the next temporary mailbox's name
   we create to be placed in the group logical name table.
*/
        status = sys$crelnm(0,&ptable,&lnm,0,lnm_items);
        check(status);
/* Create/assign a channel to the listener mailbox. */
        status = sys$crembx(0,&ctx.l_chan,0,0,0,0,&l_mbx,CMB$M_WRITEONLY,0);
        check(status);

/* Set a timer for synching the file. */
        status = sys$setimr(0,&synch_time,synch_ast,&ctx,0);
        check(status);
/* set up shared context for the sample buffer. */
        ctx.samp_buffer = &sample;

/* Post an asynchronous read to the mailbox. */
        status = sys$qio(0,ctx.chan,IO$_READVBLK,&ctx.ios,mbx_ast,&ctx,
                        ctx.samp_buffer,sizeof(*(ctx.samp_buffer)),0,0,0,0);
        check(status);

/* Go to sleep. */
        sys$hiber();

/* If we are waken, run down. */
        return(SS$_NORMAL);
}

/* AST routine to read the mailbox. */
#define EXP_OBJECTS_WRITTEN 1
void    mbx_ast(Mbx_ctx *ctx)
{
        int     write_cnt;
        int     status;
/* Check to see if the qio completed properly. */

        switch(ctx->ios.iosb$w_status)
        {
                case SS$_ENDOFFILE:
                        fclose(ctx->fp);
                        status = sys$wake(0,0);
                        check(status);
                        status = sys$cantim(0,0);
                        check(status);
                /* Send EOF to listener. */
                        status = sys$qiow(0,ctx->l_chan,
                                        IO$_WRITEOF|IO$M_READERCHECK,
                                        &ctx->l_ios,0,0,
                                        0,0,0,0,0,0);
                        check(status);
                        if(ctx->l_ios.iosb$w_status == SS$_NOREADER)
                        {
                /* Ignore if no reader. */
                                ;
                        }
                        else
                        {
                                check(ctx->l_ios.iosb$w_status);
                        }

                        break;

                default:
                        check(ctx->ios.iosb$w_status);
                        write_cnt = fwrite(ctx->samp_buffer,
                                        sizeof(*(ctx->samp_buffer)),
                                        1,ctx->fp);
                        if(write_cnt != EXP_OBJECTS_WRITTEN)
                        {
                                fprintf(stderr,"Write error!\n");
                        }
                /* Post another read to the mailbox. */
                        status = sys$qio(0,ctx->chan,IO$_READVBLK,
                                        &ctx->ios,mbx_ast,ctx,
                                        ctx->samp_buffer,
                                        sizeof(*(ctx->samp_buffer)),0,0,0,0);
                        check(status);
                /* Send buffer to listener. */
                        status = sys$qiow(0,ctx->l_chan,
                                        IO$_WRITEVBLK|IO$M_READERCHECK,
                                        &ctx->l_ios,0,0,
                                        ctx->samp_buffer,
                                        sizeof(*(ctx->samp_buffer)),0,0,0,0);
                        check(status);
                        if(ctx->l_ios.iosb$w_status == SS$_NOREADER)
                        {
                /* Ignore if no reader. */
                                ;
                        }
                        else
                        {
                                check(ctx->l_ios.iosb$w_status);
                        }
                        break;
        }
}

#include <unistd.h>
/* AST routine to synch the file every 5 seconds. */
void    synch_ast(Mbx_ctx *ctx)
{
        __int64 synch_time = -50000000;
        int     status;

/* Synch the file. */
        fsync(fileno(ctx->fp));
/* Reset a timer for synching the file. */
        status = sys$setimr(0,&synch_time,synch_ast,ctx,0);
        check(status);
}
$
```

Example 19. Code to Dump Samples

This code is implemented as a foreign command.

```
$
$ type dump_samples.c
/*****************************************
        Program to dump output from PC Sampler log file.
        Foreign comand setup by using the DCL command:
        $ PC_DUMP == "$dev[dir]DUMP_SAMPLES.EXE"
*****************************************/
#include <stdio.h>
#include <stdlib.h>
#include <descrip.h>   // This is from SYS$LIBRARY:DECCRTLDEF.TLB
#include <starlet.h>
#include <ssdef.h>
#include "pc_sampler1.h"
#define EXPECTED_ARGS 2
#define NO_ARGS 1
#define FILE_ARG 1
#define TIME_STR_LEN 23
#define LINES_PER_PAGE 24
#define CMD_ARG 1
#define check(S) if(!((S)&1)) sys$exit(S)
/* Get parameter(s) from the command line. */
int     main(int argc, char **args)
{

/* File pointer for the data file. */
        FILE *fp;
        int     i;
/* Structure for the sample data. */
        Samp    sample;
        int     items_read;
        int     status;
/* String to hold the text representation of the time stamp. */
        char    time_str[TIME_STR_LEN+1];
        $DESCRIPTOR(time_dsc,time_str);
/* Process the command line argument(s). */
        switch(argc)
        {
                default:
                        fprintf(stderr,"Bad command format!"
                                "\nUse: PC_DUMP file-name\n");
                        exit(SS$_INSFARG);
                        break;
                case    EXPECTED_ARGS:
        /* Open the data file. */
                        fp = fopen(args[FILE_ARG],"r");
                        if(!fp)
                        {
                                fprintf(stderr,"Bad input file name.\n");
                                perror(args[CMD_ARG]);
                                exit(EXIT_FAILURE);
                        }
        }
        i = 0;
/* Read until EOF or error and display the samples. */
        while((items_read = fread(&sample,sizeof(sample),1,fp)) == 1)
        {
                status = sys$asctim(&time_dsc.dsc$w_length,&time_dsc,
                                &sample.time_stamp,0);
                check(status);
/* Convert the string returned to a C-style string. */
                time_str[time_dsc.dsc$w_length] = '\0';
/* Print a header after each 24 lines. */
                if(i%LINES_PER_PAGE == 0)
                {
                        printf("%-8s  %-23s  %-8s  %-10s  %-10s\n",
                                "Sample","Time of Sample","PC","BIO","DIO");
                        printf("%-8s  %-23s  %-8s  %-10s  %-10s\n",
                                "------","--------------","--","---","---");
                }
                printf("%08d:  %23s  %08x  %10d  %-10d\n",
                        ++i,time_str,sample.pc,sample.bio,sample.dio);
        }
/* Make sure we hit the end of file. */
        if(feof(fp))
        {
                puts("***** No more data *****");
        }
        else
        {
                fprintf(stderr,"Error reading %s\n",args[FILE_ARG]);
                exit(EXIT_FAILURE);
        }
        return(EXIT_SUCCESS);
}

$
$
$ cc dump_samples
$ link dump_samples
$
```

Define a foreign command for the sample dumper.

```
$ pc_dump == "$SYS$SYSDEVICE:[ELLIS.NASA]dump_samples"
$
```

Validate that the code generates an error when no sample file name is provided.

```
$ pc_dump
Bad command format!
Use: PC_DUMP file-name
%SYSTEM-F-INSFARG, insufficient call arguments
$
```

Example 20. PC_LISTENER Code

```
$ type pc_listener.c
/**************************************
        PC_LISTENER
        listens to PC_LISTENER_MBX mailbox and displays samples.
        If no PC_LOGGER is active, aborts
*************************************/
#include <descrip.h>
#include <cmbdef.h>
#include <stdio.h>
#include <starlet.h>
#include <iosbdef.h>
#include <iodef.h>
#include <ssdef.h>
#include <lnmdef.h>
#include <iledef.h>
#include <string.h>
#define check(S) if(!((S)&1)) sys$exit(S)
#include "pc_sampler3.h"
#define TIME_STR_LEN 23
#define LIST_END 0
int     main(void)
{
/* Mailbox name. */
        $DESCRIPTOR(mbx,"PC_LISTENER_MBX");
        int     status;
        iosb    ios;
        short chan;
        __int64 stall = (__int64) -20000000;
        Samp    buffer;
/* String yo hold the text representation of the time stamp. */
        char    time_str[TIME_STR_LEN+1];
        $DESCRIPTOR(time_dsc,time_str);
        $DESCRIPTOR(ptable,"LNM$PROCESS_DIRECTORY");
        $DESCRIPTOR(lnm,"LNM$TEMPORARY_MAILBOX");
        char    equiv[] = "LNM$GROUP";
        ile3    lnm_items[] = {{strlen(equiv),LNM$_STRING,equiv},{LIST_END}};
/*
        Create a logical name to cause the mailbox name to be placed
        in the group logical name table.
*/
        status = sys$crelnm(0,&ptable,&lnm,0,lnm_items);
        check(status);
/* Create/assign a table to the mailbox. */
        status = sys$crembx(0,&chan,0,0,0,0,&mbx,CMB$M_READONLY,0);
        check(status);

/* Read the mailbox using SYS$QIO until EOF. */
        do
        {
                status = sys$qiow(0,chan,IO$_READVBLK|IO$M_WRITERCHECK,&ios,
                                0,0,&buffer,sizeof(buffer),0,0,0,0);
                check(status);
                switch(ios.iosb$w_status)
                {
                        case SS$_ENDOFFILE:
                                puts("***  No more Data *** ");
                                break;
                        case SS$_NOWRITER:
                                fprintf(stderr,"Logger is not active.  "
                                        "Try again later.\n");
                                sys$exit(SS$_NOLISTENER);
                                break;
                        default:
                                check(ios.iosb$w_status);
/* Send the message to the screen. */
                                status = sys$asctim(&time_dsc.dsc$w_length,
                                        &time_dsc,
                                        &buffer.time_stamp,0);
                                check(status);
/* Convert the time string returned to a C-style string. */
                                time_str[time_dsc.dsc$w_length] = '\0';
                                printf("%08d:  %23s  %08x  %10d  %10d\n",
                                        buffer.sample_no,time_str,
                                        buffer.pc,buffer.bio,
                                        buffer.dio);
                }
        } while(ios.iosb$w_status != SS$_ENDOFFILE);
        return(SS$_NORMAL);
}
```

Example 21. Build Process

```
$
$ cc pc_tester
$ cc stub
$ cc pc_sampler3
$ link pc_tester,stub,pc_sampler3
$
$ cc pc_logger2
$ link pc_logger2
$
$ cc pc_listener
$ link pc_listener
$
```

Example 22. Sample Runs

```
$ r pc_tester
$ pc_dump sample.data
Sample    Time of Sample           PC        BIO        DIO
------    --------------           --        ---        ---
00000001:   1-SEP-2004 23:39:00.33  000300f0        3013  4625
00000002:   1-SEP-2004 23:39:01.33  00030110        3014  4625
00000003:   1-SEP-2004 23:39:02.33  00030150        3015  4625
00000004:   1-SEP-2004 23:39:03.34  00030150        3016  4625
00000005:   1-SEP-2004 23:39:04.34  00030104        3017  4625
00000006:   1-SEP-2004 23:39:05.34  00030110        3018  4625
```

```
00000007:   1-SEP-2004 23:39:06.34  00030150       3019  4625
00000008:   1-SEP-2004 23:39:07.34  000300f0       3020  4625
00000009:   1-SEP-2004 23:39:08.36  00030150       3021  4625
00000010:   1-SEP-2004 23:39:09.36  00030100       3022  4625
00000011:   1-SEP-2004 23:39:10.36  00030110       3023  4625
00000012:   1-SEP-2004 23:39:11.36  00030150       3024  4625
00000013:   1-SEP-2004 23:39:12.36  00030110       3025  4625
00000014:   1-SEP-2004 23:39:13.36  00030104       3026  4625
***** No more data *****
$
```

The Listener is run independently from another terminal session and picks up the data as it comes in.

Example 23. Sample PC_LISTENER Runs

```
$ r pc_listener
00000010:   1-SEP-2004 23:39:09.36  00030100       3022        4625
00000011:   1-SEP-2004 23:39:10.36  00030110       3023        4625
00000012:   1-SEP-2004 23:39:11.36  00030150       3024        4625
00000013:   1-SEP-2004 23:39:12.36  00030110       3025        4625
00000014:   1-SEP-2004 23:39:13.36  00030104       3026        4625
***  No more Data ***
$
Sample run with the sampler inactive.
$ r pc_listener
Logger is not active.  Try again later.
%SYSTEM-F-NOLISTENER, specified remote system process not listening
$
$ r pc_tester
$ pc_dump sample.data
Sample    Time of Sample           PC        BIO        DIO
------    --------------           --        ---        ---
00000001:   1-SEP-2004 23:40:05.92  00030104       3113  4631
00000002:   1-SEP-2004 23:40:06.92  00030100       3114  4631
00000003:   1-SEP-2004 23:40:07.93  00030104       3115  4631
00000004:   1-SEP-2004 23:40:08.93  00030150       3116  4631
00000005:   1-SEP-2004 23:40:09.93  000300f0       3117  4631
00000006:   1-SEP-2004 23:40:10.93  00030150       3118  4631
00000007:   1-SEP-2004 23:40:11.94  000300f0       3119  4631
00000008:   1-SEP-2004 23:40:12.94  000300f0       3120  4631
00000009:   1-SEP-2004 23:40:13.94  00030150       3121  4631
00000010:   1-SEP-2004 23:40:14.94  00030100       3122  4631
00000011:   1-SEP-2004 23:40:15.94  000300f0       3123  4631
00000012:   1-SEP-2004 23:40:16.94  00030110       3124  4631
00000013:   1-SEP-2004 23:40:17.94  000300f0       3125  4631
00000014:   1-SEP-2004 23:40:18.95  00030150       3126  4631
***** No more data *****
$
```

Again, this is run from another session.

```
$ r pc_listener
00000007:   1-SEP-2004 23:40:11.94  000300f0       3119        4631
00000008:   1-SEP-2004 23:40:12.94  000300f0       3120        4631
00000009:   1-SEP-2004 23:40:13.94  00030150       3121        4631
00000010:   1-SEP-2004 23:40:14.94  00030100       3122        4631
```

```
00000011:   1-SEP-2004 23:40:15.94  000300f0       3123       4631
00000012:   1-SEP-2004 23:40:16.94  00030110       3124       4631
00000013:   1-SEP-2004 23:40:17.94  000300f0       3125       4631
00000014:   1-SEP-2004 23:40:18.95  00030150       3126       4631
***  No more Data ***
$
```

For more information

On Mailboxes go to: http://h71000.www7.hp.com/doc/os83_index.html
Consult the following Manuals:
HP OpenVMS I/O User's Reference Manual
HP OpenVMS Programming Concepts Manual
HP OpenVMS System Services Reference Manual

To contact the author send email to: Bruce.Ellis@BRUDEN.com