IBM Secureway Cryptographic Products
IBM 4758 PCI Cryptographic Coprocessor
CP/Q Operating System
Application Programming Reference
Version 1 Driver 16

Twenty-eighth Edition, January 1998

This edition of the Application Programming Reference manual is for CP/Q Version 1 Driver 16 for Intel Processors.

Comments or queries concerning this document should be addressed to:

D. C. Toll
IBM Research Division, T. J. Watson Research Center
PO Box 704
Yorktown Heights
New York 10598, U.S.A.

E-mail: toll@watson.ibm.com
VNET id: TOLL at YKTVMV
☎ (tie-line) 863 7019
☏ (external, USA) 914 784 7019

FAX:
☎ (tie-line) 863 6225
☏ (external, USA) 914 784 6225

This manual was produced using IBM DCF/SCRIPT release 4.0, and Publishing Systems BookMaster release 4.

Contents

SVC Handler Calls

Figures

SVC Handler Calls

Chapter 1. Section Notes

This part of this manual contains the definitive specification of each SVC Handler Call; The C callable subroutines for the system calls are provided in the library named CPQLIB, and the routine declarations for these, required when compiling the calls to these routines, are contained in the header file CPQLIB.H.

Note: In the Intel implementation of CP/Q, each routine is declared as **pascal**, that is the “Pascal” calling convention is used. For reasons of clarity and simplicity, this is not shown in the SVC Handler Call descriptions in this manual. However, as long as the user includes CPQLIB.H for the declarations, this is taken care of automatically, and the user's code will not have portability problems when moving it between systems.

The Signal facilities are not available via routines in the CPQLIB library; these are accessed via routines in the UNIXLIB library, with declarations in SYS\SIGNAL.H.

Chapter 2. Suspend Operations

CPIntWait (INT_WAIT) - Wait for Interrupt

Function

This SVC enables the caller to await a hardware interrupt. It is restricted to code with input/output (I/O) privilege, namely:

- For Intel, the calling code Current Privilege Level (CPL) must be ≤ the system Input/Output Privilege Level (IOPL).
- For PowerPC, the calling task must be a supervisor mode task.

If this SVC specifies a time-out, and while the task is waiting for the interrupt a signal arrives that causes a signal handler to be entered and this SVC is then resumed after the return from the signal handler, the time-out is restarted from the beginning. Currently, no attempt is made to restart the time-out for only the remaining period.

C Syntax

```
long int CPIntWait(unsigned long int SLIBid,
                   unsigned long int Timeout);
```

Parameters

SLIBid The ID of the Second Level Interrupt Block (SLIB) returned when the CRT_SLIH SVC was issued.

Timeout The time-out period. This may be one of the following values:

0 - SVCNOWAIT
Not valid for this SVC.

0xFFFFFFFF - SVCWAITFOREVER
Wait indefinitely until an interrupt occurs.

other Wait until an interrupt occurs, but only for the specified time (in μsecs). If this time-out occurs, the SVC Handler returns to the caller with return code *QSVCtimedout*.

Return Codes

QSVCgood (0)
Operation completed successfully (an interrupt has occurred).

QSVCbadreq (0x8001001E)
The parameter *Timeout* is 0.

QSVCbadid (0x8001002D)
This can occur for one of the following reasons:

- The specified SLIB ID is not the offset of a SLIB.
- The specified SLIB ID is not that of a SLIH set up by the calling task.
- This SLIB is not for an INT_WAIT type SLIH.

QSVCtimedout (0x8001000B)
: The time-out occurred (no interrupt has been received).

QSVCinterrupt (0x80010024)
: The SVC was interrupted by a signal (no I/O interrupt was received).

SVC Handler Generated Faults

QSVCinvalid (x80)
: The calling code does not have I/O privilege.

CPMuxWait (MUX_WAIT) - Wait on List of SVids

Function

Warning - Optional Facility

There is a compile-time option to omit the MUX_WAIT facilities from the SVC Handler, for example, to save space in memory constrained products.

If MUX_WAIT has been omitted, this SVC results in the calling task being faulted.

This SVC suspends the caller, until one of up to eight events occurs. These events are specified in a parameter list of SVids.

Each SVid in the list may take one of the following values:

- the value zero, which means the message queue of the calling task (the event is a message present on the queue)
- the SVid of a message queue (the event is a message present on the queue)
- the SVid of a synchronization semaphore (the event is the semaphore being clear)
- the SVid of a resource/serialization semaphore (the event is the semaphore being free, whereupon it becomes claimed by the calling task)

A successful return that specifies the SVid of a message queue (this includes the ID zero, which means the calling task's own message queue) indicates at least one message is present on the queue. This does not guarantee that, when a RECV_MESG or SPLIT_RECV SVC is subsequently issued, there is still a message on the queue that can be received. No lock is applied to prevent another task from retrieving the message first.

The use of this SVC might have performance effects on the entire system. Much of the processing of a MUX_WAIT SVC runs with interrupts disabled, which implies that it might not meet the interrupt latency requirements of the remainder of the system.

If this SVC specifies a time-out, and while the task is waiting for an event a signal arrives that causes a signal handler to be entered and this SVC is then resumed after the return from the signal handler, the time-out will be restarted from the beginning. Currently, no attempt is made to restart the time-out for only the remaining period.

C Syntax

```
long int CPMuxWait(unsigned long int *List,
                   unsigned long int *Event,
                   unsigned long int Timeout);
```

Parameters

List	Pointer to a buffer, which is a parameter list of 9 32-bit words. The first word contains the count of the number of SVids that follow (in the range 1-8).
	Note: The full 36-byte parameter list *must* be supplied, even when fewer than eight SVids are specified.

Event A pointer to a location to contain the SVid of the event that occurred.

Timeout The time-out period. This may take one of the following values:

0 - SVCNOWAIT
No wait, immediate return to the caller (with return code *QSVCnoevent*) if none of the message queues contains a message, none of the resource semaphores is free, and none of the synchronization semaphores is clear.

0xFFFFFFFF - SVCWAITFOREVER
Wait indefinitely until one of the specified events occurs.

other Wait until one of the specified events occurs, but only for the specified time (in μsecs). If this time-out occurs, the SVC Handler returns to the caller with return code *QSVCtimedout*.

Return Parameters

******Event* If the return code is *QSVCgood*, this location is set to the SVid of the first item found in the list for which the event has occurred. A value of zero indicates that a message is present on the caller's task message queue.

If the return code is *QSVCdeadSVid*, this location is set to the SVid of the SVT entry that is being or has been removed.

If the return code is other than *QSVCgood* or *QSVCdeadSVid*, this location is indeterminate.

Return Codes

QSVCgood (0)
Operation completed successfully.

QSVCtimedout (0x8001000B)
The time-out has occurred.

QSVCnoevent (0x80010011)
None of the events has occurred (immediate return case only).

QSVCbadcnt (0x8001001B)
The count in bytes 0-3 of the parameter list is zero or > eight.

QSVCduplid (0x8001002C)
One or more of the SVids in the parameter list is duplicated.

QSVCdeadSVid (0x8001000C)
The SVT entry specified by the SVid in EDX is being or has been removed from the system.

QSVCbadSVid (0x80010002)
The SVid in EDX is not zero and does not specify an SVT entry of an acceptable type (for example, it is not a synchronization semaphore, a resource semaphore, or a system message queue).

QSVCinterrupt (0x80010024)
The SVC has been interrupted by a signal (no event has occurred).

SVC Handler Generated Faults

QSVCinvalid (0x80)
The MUX_WAIT facilities were omitted from the SVC Handler.

QSVCinvSVid (0x82)
The requestor is not permitted to access one or more of the specified SVT entries.

QSVCparlist (0x81)
Parameter list is invalid, for example, the parameter list pointer does not point into an allocated page.

CPShortYield (DISP_RET2) - Return to Dispatcher

Function

This SVC voluntarily yields the processor and performs a dispatch operation, without suspending the calling task. If there is another runnable task at the dispatch priority of the caller, the system switches to running that task. In this case, the caller is moved to immediately after the next task in the dispatch chain for its dispatch priority. This is a “voluntary round robin.” If the caller is the only runnable task at its priority, this is effectively a null SVC, and an immediate return to the caller occurs.

This SVC is different from the DISP_RET SVC if there are more than 3 dispatchable tasks at a level. Generally, the DISP_RET2 SVC should be used by only one task at a priority level, and the other tasks should use the DISP_RET SVC. This means one of the tasks can let the other tasks (at a level) run one at a time and then be restarted itself. If two tasks at a level both used the DISP_RET2 SVC, they would alternate use of the processor between them, and the other dispatchable tasks at that level would never run.

C Syntax

```
long int CPShortYield();
```

Parameters

None

Return Codes

QSVCgood (0)
Operation completed successfully.

SVC Handler Generated Faults

None

CPYield (DISP_RET) - Return to Dispatcher

Function

This SVC voluntarily yields the processor and performs a dispatch operation, without suspending the calling task. If another runnable task is at the dispatch priority of the caller, the system switches to run that task. In this case, the caller is moved to the end of the dispatch chain for its dispatch priority. This is a “voluntary round robin.” If the caller is the only runnable task at its priority, this is effectively a null SVC, and an immediate return to the caller occurs.

C Syntax

```
long int CPYield();
```

Parameters

None

Return Codes

QSVCgood (0)
Operation completed successfully.

SVC Handler Generated Faults

None

Chapter 3. Synchronization Semaphore Operations

CPSemClear (SEM_CLEAR) - Clear a Semaphore

Function

This SVC unconditionally clears a synchronization semaphore. This means that the operation is performed even if the semaphore is already clear. Any tasks waiting on this semaphore are restarted.

If the semaphore is already in the cleared state, no indication of this is given to the caller. Furthermore, no indication is given as to the number of tasks (if any) that were made dispatchable as a result of this system call.

C Syntax

```
long int CPSemClear(unsigned long int SVid);
```

Parameters

SVid	The SVid of the synchronization semaphore to be cleared.

Return Codes

QSVCgood (0)
Operation completed successfully.

QSVCbadSVid (0x80010002)
Specified SVT entry is not valid, or it is not a synchronization semaphore. This return code can also occur if the calling task is a *system* task, but it is not permitted to access the specified SVT entry.

QSVCdeadSVid (0x8001000C)
The specified SVT entry is being removed from the system.

SVC Handler Generated Faults

QSVCinvSVid (0x82)
The caller is not a *system* task and is not permitted to access the specified SVT entry.

CPSemSet (SEM_SET) - Set a Semaphore

Function

This SVC sets a synchronization semaphore.

Note: In the case of a semaphore that is being used for the notification of hardware interrupts, if a "pending clear" is present (that is, an interrupt occurred while the semaphore was clear), the semaphore is cleared immediately by this SVC.

C Syntax

```
long int CPSemSet(unsigned long int SVid);
```

Parameters

SVid	The SVid of the synchronization semaphore to be set.

Return Codes

QSVCgood (0)
Operation completed successfully.

QSVCbadSVid (0x80010002)
Specified SVT entry is not valid, or it is not a synchronization semaphore. This return code can also occur if the calling task is a *system* task, but it is not permitted to access the specified SVT entry.

QSVCdeadSVid (0x8001000C)
The specified SVT entry is being removed from the system.

SVC Handler Generated Faults

QSVCinvSVid (0x82)
The caller is not a *system* task and is not permitted to access to the specified SVT entry.

CPSemSetWait (SEM_SETWAIT) - Set a Semaphore and Wait for it to Clear

Function

This SVC sets and then waits on a synchronization semaphore, in a single atomic operation. This guarantees that the wait operation corresponds to this set operation. If **CPSemSet** was used, followed by **CPSemWait**, many things can potentially occur between them, including a **CPSemClear** for this semaphore.

Note: In the case of a semaphore used for the notification of hardware interrupts, if a “pending clear” is present (that is, an interrupt occurred while the semaphore was clear) then the semaphore is immediately cleared by this SVC. This results in an immediate return to the caller.

If this SVC specifies a time-out, and while the task is waiting for the semaphore a signal arrives that causes a signal handler to be entered and this SVC is then resumed after the return from the signal handler, the time-out is restarted from the beginning. Currently, no attempt is made to restart the time-out for only the remaining period.

C Syntax

```
long int CPSemSetWait(unsigned long int SVid,
                      unsigned long int Timeout);
```

Parameters

SVid The SVid of the synchronization semaphore to set and then wait for.

Timeout The wait or no-wait option and time-out period. This accepts the following values:

0 - SVCNOWAIT
No wait, an immediate return to the caller with return code *QSVCsembusy* will occur, since this SVC sets the semaphore.

0xFFFFFFFF - SVCWAITFOREVER
Wait indefinitely until the semaphore next clears.

Other Wait until the semaphore is next cleared, but only for the specified time (in μsecs). If this time-out occurs, the SVC Handler returns to the caller with return code *QSVCtimedout*.

Return Codes

QSVCgood (0)
Operation completed successfully.

QSVCsembusy (0x80010012)
The semaphore is set (immediate return case only).

Note: Because this SVC sets the semaphore, this return code is always given in the case of an immediate return.

QSVCbadSVid (0x80010002)
Specified SVT entry is not valid, or it is not a synchronization semaphore. This return code can also occur if the calling task is a “system” task, but is not permitted to access to the specified SVT entry.

QSVCdeadSVid (0x8001000C)
: The specified SVT entry is being or has been removed from the system.

QSVCtimedout (0x8001000B)
: The time-out has occurred.

QSVCinterrupt (0x80010024)
: This SVC was interrupted by a signal, and the semaphore may or may not have been cleared.

SVC Handler Generated Faults

QSVCinvSVid (0x82)
: The caller is not a “system” task, and it is not permitted to access to the specified SVT entry.

CPSemWait (SEM_WAIT) - Wait for a Semaphore to become Clear

Function

This SVC suspends the calling task until the specified synchronization semaphore is cleared. If the semaphore is already clear, an immediate return occurs.

If this SVC specifies a time-out, and while the task is waiting for the semaphore a signal arrives that causes a signal handler to be entered and this SVC is then resumed after the return from the signal handler, the time-out is restarted from the beginning. Currently, no attempt is made to restart the time-out for only the remaining period.

C Syntax

```
long int CPSemWait(unsigned long int SVid,
                   unsigned long int Timeout);
```

Parameters

SVid The SVid of the synchronization semaphore to wait on.

Timeout The wait/no-wait option and time-out period. This accepts the following values:

0 - SVCNOWAIT
No wait, immediately returns to the caller with return code *QSVCsembusy* if the semaphore is set, or return code *QSVCgood* if the semaphore is clear.

0xFFFFFFFF - SVCWAITFOREVER
Wait indefinitely until the semaphore next clears.

Other Wait until the semaphore is next cleared, but only for the specified time (in μsecs). If this time-out occurs, the SVC Handler returns to the caller with return code *QSVCtimedout*.

Return Codes

QSVCgood (0)
Operation completed successfully.

QSVCsembusy (0x80010012)
The semaphore is set (immediate return case only).

QSVCbadSVid (0x80010002)
Specified SVT entry is not valid, or it is not a synchronization semaphore. This return code can also occur if the calling task is a “system” task, but it is not permitted to access to the specified SVT entry.

QSVCdeadSVid (0x8001000C)
The specified SVT entry is being or has been removed from the system.

QSVCtimedout (0x8001000B)
The time-out has occurred.

QSVCinterrupt (0x80010024)
This SVC was interrupted by a signal, and the semaphore may or may not have been cleared.

SVC Handler Generated Faults

QSVCinvSVid (0x82)
: The caller is not a “system” task and is not permitted to access to the specified SVT entry.

Chapter 4. Inter-Task Messages

CPCountMsg (COUNT_MESG) - Return Count of Messages in a Queue

Function

This call returns the number of messages currently in a queue.

C Syntax

```
long int CPCountMsg(unsigned long int SVid,
                    unsigned long int *Count);
```

Parameters

SVid	The SVid of the message queue to be queried; this may also be the SVid of the caller or zero. If zero is specified, the message queue of the calling task is used.
Count	A pointer to a location to receive the returned message count.

Return Parameters

*****_Count_ If the return code is *QSVCgood*, this location is set to the number of messages in the queue.

Otherwise, this location is indeterminate.

Return Codes

QSVCgood (0)
Operation completed successfully.

QSVCbadSVid (0x80010002)
Specified SVT entry is not valid, or it is not a task or system message queue. This return code can also occur if the calling task is a “system” task, but it is not permitted to access to the specified SVT entry.

QSVCdeadSVid (0x8001000C)
The specified SVT entry is being or has been removed from the system.

SVC Handler Generated Faults

QSVCinvSVid (0x82)
The caller is not a “system” task, and it is not permitted to access to the specified SVT entry.

CPPeekMsg (PEEK_MESG) - “Peek” a Message

Function

This SVC examines an item on a message queue, without removing the item from the queue. This SVC returns only the header of the message buffer. It does *not* return the message data. Specifically, the data count returned in the message header by this SVC is the actual value in the header of the message. Thus this SVC can be used to determine the required buffer length necessary to receive a particular message.

After peeking a message, if that message is to be received in a subsequent SVC, use the “by message id” option in the RECV_MESG or SPLIT_RECV SVC and quote the message ID returned by the PEEK_MESG SVC. This ensures that the correct message is received.

This SVC by default returns data for the first message on the queue. However, by specifying a non-zero *Mode* parameter and setting appropriate values in the supplied message header, a certain specific message can be peeked, or only messages from a specified task, or only messages with the specified type.

A successful peek does not guarantee that a subsequent receive is successful. For example, the system does not lock the peeked message, which means that another task may receive it after it has been peeked.

If this SVC specifies a time-out, and while the task is waiting for a message a signal arrives that causes a signal handler to be entered and this SVC is then resumed after the return from the signal handler, the time-out is restarted from the beginning. Currently, no attempt is made to restart the time-out for only the remaining period.

C Syntax

```
long int CPPeekMsg(void *MsgBuf,
                   unsigned long int *Count,
                   unsigned long int Mode,
                   unsigned long int Timeout);
```

Parameters

MsgBuf A pointer to a buffer to contain a message header (that is, a *QMSGHDR* structure), as shown in Figure 4-1 on page 4-3.

Fields in this structure must be set up as follows:

Byte 0 - *HdrVer*
Must be set to the message header format (currently always 0).

bytes 4-7 - *send_id*
If *Mode* bit 5 is set, this field must be set to the SVid of the required sending task. Otherwise, this field is ignored in the call.

bytes 12-15 - *msgq_id*
Must be set to the SVid of the message queue to be searched; this can also be the SVid of the caller or zero. If

```
typedef struct qmsghdr
{
  unsigned char      HdrVer;      /* Header version                      */
  unsigned char      msg_pri;     /* Message priority                    */
  unsigned short int rsrv1;       /* (reserved)                          */
  unsigned long int  send_id;     /* Sender's task SVid                  */
  unsigned long int  rsrv2;       /* (reserved)                          */
  unsigned long int  msgq_id;     /* Message queue SVid (0 = self)       */
  unsigned long int  rsrv3;       /* (reserved)                          */
  unsigned long int  msg_id;      /* Unique, system assigned Message ID  */
  unsigned long int  dw_count;    /* Data area length (not incl. header) */
  unsigned long int  msg_type;    /* Message type (user defined field)   */
} QMSGHDR;
```

Figure 4-1. Message Header Structure

zero is specified, the message queue of the calling task is used.

bytes 20-23 - *msg_id*
If *Mode* bit 7 is set, this field must be set to the message ID of the message to be peeked. Otherwise, this field is ignored in the call.

bytes 28-31 - *msg_type*
If *Mode* bit 6 is set, this field holds the desired message type. Otherwise, this field is ignored in the call.

Other fields in the buffer are ignored by the SVC Handler.

Count A pointer to a location to receive the count of messages in the queue.

Mode An 8-bit bit significant field, as follows:

bit 7 - QSVCmsg_id (0x01)
If set, this SVC peeks only a message with the message ID contained in the field *msg_id* (bytes 20-23) of the parameter list. If this bit is set, bits 0-6 must be 0, and *timeout* must also be 0.

bit 6 - QSVCmsg_type (0x02)
If set, this SVC peeks only a message with the message type specified in the field *msg_type* (bytes 28-31) of the parameter list.

If this bit is set, bit 5 can also be set. This further restricts the selection to messages of the specified type from a particular sender.

bit 5 - QSVCmsg_sender (0x04)
If set, this SVC peeks only a message sent by the task whose SVid is specified in the field *send_id* (bytes 4-7) of the parameter list.

If this bit is set, bit 6 can also be set. This further restricts the selection to messages with a specific type from this sender.

If this field is zero, the system call peeks the first message on the specified queue.

Timeout The wait/no-wait option and time-out period. This accepts the following values:

Note: If *Mode* bit 7 is set, *Timeout* must be 0.

0 - SVCNOWAIT
No wait, immediate return to the caller (with return code *QSVCQempty*) if a message is not available.

0xFFFFFFFF - SVCWAITFOREVER
Wait indefinitely until a message arrives on the queue.

other Wait until a message becomes available, but only for the specified time (in µsecs). If this time-out occurs, the SVC Handler returns to the caller with return code *QSVCtimedout*.

Return Parameters

***Count** If the return code is set to *QSVCgood* or *QSVCdeadSVid*, this location is set to the count of messages currently in the queue, including the message returned by this SVC.

Otherwise, this location is indeterminate.

message buffer
If the return code is *QSVCgood*, the fields of the message header buffer are set as follows. Otherwise, the buffer is unchanged from the call.

byte 0 - *HdrVer*
Set to 0 (the message buffer format).

byte 1 - *msg_pri*
Set to the message priority.

bytes 2,3 Indeterminate.

bytes 4-7 - *send_id*
Set to the SVid of the message sender.

bytes 8-11
Indeterminate (these bytes actually contain whatever was in bytes 4-7 of the sender's message buffer).

bytes 12-15 - *msgq_id*
The target SVid of this message. In the case of a message from the caller's own message queue, this is the SVid of the calling task.

bytes 16-19
Indeterminate (these bytes actually contain whatever was in bytes 16-19 of the sender's message buffer).

bytes 20-23 - *msg_id*
The message ID assigned by the system when the message was sent.

bytes 24-27 - *dw_count*
The count of double-words specifying the size of the buffer required to receive this message, excluding the message header. This is in the range 0-1024.

bytes 28-31 - *msg_type*
The message type, as specified in bytes 28-31 of the sender's message buffer.

Return Codes

QSVCgood (0)
Operation completed successfully.

QSVCbadSVid (0x80010002)
Specified SVT entry is not valid, or it is not a task or system message queue. This return code can also occur if the calling task is a "system" task, but it is not permitted to access to the specified SVT entry.

QSVCdeadSVid (0x8001000C)
The Specified SVT entry is being or has been removed from the system.

QSVCQempty (0x80010009)
Specified queue is empty, or a message with the specified ID, type, or sender cannot be found on the queue.

QSVCbadreq (0x8001001E)
This can occur for the following reasons:

- *Mode* bit 7 is set, and bits 0-6 are not all unset.
- *Mode* bit 7 is set and *Timeout* ≠ 0.
- *Mode* bit 7 is unset, and bit 5, 6, or both are set, and bits 0-4 are not all unset.
- The buffer header format type (byte 0) is not 0.

QSVCtimedout (0x8001000B)
The Time-out has occurred.

QSVCinterrupt (0x80010024)
This SVC was interrupted by a signal, and no data has been returned.

SVC Handler Generated Faults

QSVCparlist (0x81)
The parameter *MsgBuf* does not point to an allocated writeable page.

QSVCinvSVid (0x82)
The caller is not a "system" task and it is not permitted to access to the specified SVT entry.

If the specified buffer is not accessible or within its segment limits, or if all or part of the buffer cannot have physical pages allocated to it, the following faults can occur for this SVC:

- General protection fault (Intel)
- Stack fault (Intel)
- Data storage interrupt (PowerPC)
- Page fault

CPRecvMsg (RECV_MESG) - Receive a Message

Function

This SVC obtains an item from a request queue. Receiving the message removes it from the message queue.

The task issuing the RECV_MESG SVC must provide a buffer large enough to hold the header plus *n* 32-bit words for the message data. The receiver's message buffer should be aligned on a double-word boundary for maximum efficiency.

Note: If the program uses CP/Q C heap management routines to allocate the buffer, double-word alignment is guaranteed.

If the size of the incoming message is larger than the supplied buffer (as specified by the message header in the buffer), only that portion of the message that fits in the caller's buffer is returned. The remaining data is lost. In this case, the return code will be *QSVCsmallbuff*.

This SVC (by default) returns the first message on the queue. However, by specifying a non-zero *Mode* parameter and setting appropriate values in the supplied message header, the following messages can be received:

- A specific message
- Only messages from a specified task
- Only messages with the specified type
- Only messages from a specified task with a particular type

Each message is associated with a unique message ID, which is assigned by the system when the message is sent. This ID is returned to the sender of a message, and it is included in the message header sent to the receiver of the message. By convention, when a task sends a message that is a reply to some message that task received, the type field in the header of the reply message is set to the message ID of the original (request) message. Thus, after sending a message, a task might wait for the reply to that message by specifying “receive messages with a particular type” and setting the type to the message ID of the sent message.

If this SVC specifies a time-out, and while the task is waiting for a message a signal arrives that causes a signal handler to be entered and this SVC is then resumed after the return from the signal handler, the time-out is restarted from the beginning. Currently, no attempt is made to restart the time-out for only the remaining period.

This call is a potentially blocking function call, which means if the queue is empty, or it does not have a message with a matching message ID, type value, or sender ID, the task can block within the SVC Handler until an acceptable message arrives. There are two ways to avoid an indefinite blocking situation. First, the caller can use the Timeout parameter as described above. The other method uses the CPCountMsg function call. This call (described above) is non-blocking and returns the number of messages in a particular queue.

A task can also examine messages using the CPPeekMsg call.

```
typedef struct qmsghdr
{
  unsigned char      HdrVer;      /* Header version                      */
  unsigned char      msg_pri;     /* Message priority                    */
  unsigned short int rsrv1;       /* (reserved)                          */
  unsigned long int  send_id;     /* Sender's task SVid                  */
  unsigned long int  rsrv2;       /* (reserved)                          */
  unsigned long int  msgq_id;     /* Message queue SVid (0 = self)       */
  unsigned long int  rsrv3;       /* (reserved)                          */
  unsigned long int  msg_id;      /* Unique, system assigned Message ID  */
  unsigned long int  dw_count;    /* Data area length (not incl. header) */
  unsigned long int  msg_type;    /* Message type (user defined field)   */
} QMSGHDR;

typedef struct msg
{
  QMSGHDR            h;
  unsigned long int  msg_data[1024];  /* 0-1024 32_bit words of data     */
} MSG;
```

Figure 4-2. Message Structures

C Syntax

```
long int CPRecvMsg(void *MsgBuf,
                   unsigned long int *Count,
                   unsigned long int Mode,
                   unsigned long int Timeout);
```

Parameters

MsgBuf A pointer to a buffer to contain a message and header (that is, a *MSG* structure), as shown in Figure 4-2.

Fields in this structure must be set up as follows:

byte 0 - *h.HdrVer*
Must be set to the message header format (currently always 0).

bytes 4-7 - *h.send_id*
If *Mode* bit 5 is set, this field must be set to the SVid of the desired sending task. Otherwise, this field is ignored in the call.

bytes 12-15 - *h.msgq_id*
Must be set to the SVid of the message queue to be searched; this may also be the SVid of the caller or zero. If zero is specified, the message queue of the calling task is used.

bytes 20-23 - *h.msg_id*
If *Mode* bit 7 is set, this field must be set to the message ID of the message to be peeked. Otherwise, this field is ignored in the call.

bytes 24-27 - *h.dw_count*
The length in 32-bit words of the message data vector *msg_data* in the supplied buffer. This count (and the

associated vector) should be large enough to receive the maximum anticipated message size. A message is limited to 1024 32-bit words (not including the header).

bytes 28-31 - *h.msg_type*
If *Mode* bit 6 is set, this field holds the desired message type. Otherwise, this field is ignored in the call.

Other fields in the *MsgBuf* structure are ignored by the SVC Handler.

Count A pointer to a location to receive the count of messages in the queue.

Mode An 8-bit bit significant field, as follows:

bit 7 - QSVCmsg_id (0x01)
If set, this SVC receives only a message with the message ID contained in the field *msg_id* (bytes 20-23) of the parameter list. If this bit is set, bits 0-6 must be 0, and *timeout* must also be 0.

bit 6 - QSVCmsg_type (0x02)
If set, this SVC receives only a message with the message type specified in the field *msg_type* (bytes 28-31) of the parameter list.

If this bit is set, bit 5 can also be set. This further restricts the selection to messages of the specified type from a particular sender.

bit 5 - QSVCmsg_sender (0x04)
If set, this SVC receives only a message sent by the task whose SVid is specified in the field *send_id* (bytes 4-7) of the parameter list.

If this bit is set, bit 6 can also be set. This further restricts the selection to messages with a specific type from this sender.

If this field is 0, the system call receives the first message on the specified queue.

Timeout The wait/no-wait option and time-out period. This accepts the following values: if *Mode* bit 7 is set, and *Timeout* must be zero.

0 - SVCNOWAIT
No wait, immediate return to the caller (with return code *QSVCQempty*) if a message is not available.

0xFFFFFFFF - SVCWAITFOREVER
Wait indefinitely until a message arrives on the queue.

other Wait until a message becomes available, but wait only for the specified time (in μsecs). If this time-out occurs, the SVC Handler returns to the caller with return code *QSVCtimedout*.

Return Parameters

*****Count* If the return code is set to *QSVCgood* or *QSVCdeadSVid*, this location is set to the count of messages remaining in the queue after removing this message.

Otherwise, this location is indeterminate.

message buffer

If the return code is *QSVCgood*, the fields of the message header buffer are set as follows.

Note: Otherwise, the buffer is unchanged from the call.

byte 0 - *h.HdrVer*
Set to 0 (the message buffer format).

byte 1 - *h.msg_pri*
Set to the message priority.

bytes 2,3 Indeterminate.

bytes 4-7 - *h.send_id*
Set to the SVid of the sender of the message.

bytes 8-11
Indeterminate (these bytes contain what was in bytes 4-7 of the sender's message buffer).

bytes 12-15 - *h.msgq_id*
The target SVid of this message. In the case of a message from the caller's message queue, this is the SVid of the calling task.

bytes 16-19
Indeterminate (these bytes contain what was in bytes 16-19 of the sender's message buffer).

bytes 20-23 - *h.msg_id*
The ID of this message, as assigned by the system when this message was sent.

bytes 24-27 - *h.dw_count*
The count of 32-bit words returned in bytes 32 onwards of the buffer (that is, the vector *msg_data*). This is in the range 0-1024.

bytes 28-31 - *h.msg_type*
The message type, as specified in bytes 28-31 of the sender's message buffer.

bytes 32 onwards - *msg_data*
Message data, the count is specified in *h.dw_count* (bytes 24-27).

Return Codes

QSVCgood (0)
Operation completed successfully.

QSVCbadSVid (0x80010002)
Specified SVT entry is not valid, or it is not a task or system message queue. This return code can also occur if the calling task is a “system” task, but it is not permitted to access to the specified SVT entry.

QSVCdeadSVid (0x8001000C)
The specified SVT entry is being or has been removed from the system.

QSVCQempty (0x80010009)
The specified queue is empty, or a message with the specified ID, type, or sender cannot be found on the queue

QSVCbadreq (0x8001001E)

This can occur for the following reasons:

- *Mode* bit 7 is set, and bits 0-6 are not all unset.
- *Mode* bit 7 is set and *Timeout* ≠ 0.
- *Mode* bit 7 is unset, and bits 5 or 6 or both are set, and bits 0-4 are not all unset.
- The buffer header format type (byte 0) is not 0.
- The data count is >1024 32-bit words.

QSVCsmallbuff (0x80010010)

A message has been returned, but the caller's buffer is too small to receive all of the message data. The returned data count shows how much data was returned. The remaining data has been lost.

QSVCtimedout (0x8001000B)

The time-out has occurred.

QSVCinterrupt (0x80010024)

This SVC was interrupted by a signal, and no data has been returned.

SVC Handler Generated Faults

QSVCparlist (0x81)

The parameter *MsgBuf* does not point to an allocated writeable page.

QSVCinvSVid (0x82)

The caller is not a “system” task and is not permitted to access to the specified SVT entry.

If the specified buffer is not accessible or within its segment limits, or if all or part of the buffer cannot have physical pages allocated to it, the following faults can occur:

- General protection fault (Intel)
- Stack fault (Intel)
- Data storage interrupt (PowerPC)
- Page fault

CPSendMsg (SEND_MESG) - Send a Message

Function

This SVC sends a message to a message queue or another task.

An item for this message is added to the target queue, in a position that depends on the priority of the new message relative to the priorities of messages already on the queue. The new message is inserted in the queue after all messages on the queue with higher (that is, numerically smaller) priority, and immediately after any messages already on the queue with the same priority as the new message.

Note: The priority should be specified as 255 except in those circumstances when higher priority messages really are required. This gives better performance because the new message is then simply added to the end of the queue. This eliminates searching the queue for the correct place to insert the new item.

In this case, the operation of the queue is FIFO (that is, the first message sent is the first received).

The message header from this call is copied into an RQE (Request Queue Element) on the target message queue. The message data is also copied into this RQE if the data count is not greater than the maximum length of a short message, which defaults to 7 32-bit words, but can be changed at system build time to be any value in the range 6-16. If the data count is too great for a short message, the data is copied to a system buffer within the Message Buffer Area (MBA). It is more efficient to use short messages, because this eliminates the need to allocate an MBA buffer when sending the message and free this buffer when the message is received.

Upon return to the caller from this SVC, the caller's buffer has been copied and is free for immediate re-use.

Each message is associated with a unique message ID, which is assigned by the system when the message is sent. This ID is returned to the sender of a message, and it is included in the message given to the receiver of the message. By convention, when a task sends a message that is a reply to some message that task received, the type field in the header of the reply message is set to the message ID of the original (request) message. Thus, after sending a message, a task might wait for the reply to that specific message by specifying “receive messages with a particular type” and setting the type to the message ID of the sent message.

Although this call is non-blocking, if there is a task of higher dispatch priority currently waiting in a RECV_MESG on the destination message queue, and the new message matches the requirements (type, sending task SVid, etcetera) of the receiving task, then an immediate task change occurs to that task.

It is recommended that the sender's message buffer be aligned on a 32-bit word boundary for maximum efficiency. If the program uses CP/Q C heap management routines to allocate the buffers, this alignment is guaranteed.

```
typedef struct qmsghdr
{
  unsigned char      HdrVer;      /* Header version                        */
  unsigned char      msg_pri;     /* Message priority                      */
  unsigned short int rsrv1;       /* (reserved)                            */
  unsigned long int  send_id;     /* Sender's task SVid                    */
  unsigned long int  rsrv2;       /* (reserved)                            */
  unsigned long int  msgq_id;     /* Message queue SVid (0 = self)         */
  unsigned long int  rsrv3;       /* (reserved)                            */
  unsigned long int  msg_id;      /* Unique, system assigned Message ID    */
  unsigned long int  dw_count;    /* Data area length (not incl. header)   */
  unsigned long int  msg_type;    /* Message type (user defined field)     */
} QMSGHDR;

typedef struct msg
{
  QMSGHDR            h;
  unsigned long int  msg_data[1024];  /* 0-1024 32_bit words of data     */
} MSG;
```

Figure 4-3. Message Structures

C Syntax

`long int CPSendMsg(void *MsgBuf);`

Parameters

MsgBuf A pointer to a buffer to contain a message and header (that is, a MSG structure), as shown in Figure 4-3.

Fields in this structure must be set up as follows:

byte 0 - *h.HdrVer*
Set to the message header format (currently always 0).

byte 1 - *h.msg_pri*
The required message priority.

Priority	Comments
0-31	Sending task must have *TCBsyspriv* privilege bit set.
32-254	Available to all tasks.
255	Available to all tasks. This is the recommended choice for most messages. The system does not perform an insertion scan of the destination queue, but instead simply adds the message to the end of the queue. This results in a performance savings proportional to the current size of the destination queue.

bytes 2,3 Unused.

bytes 4-7 - *h.send_id*
Ignored by the SVC Handler. In the message seen by the receiver, these bytes are over-written with the SVid of the sender.

bytes 8-11
Ignored by the SVC Handler. These bytes are passed unaltered to the receiver of the message.

Note: To avoid future compatibility problems, these bytes should be set to 0.

bytes 12-15 - *h.msgq_id*
In a call from a C program, this must be set to the SVid of the message queue or task to receive this message.

At the assembler interface level, this field is ignored by the SVC Handler; the destination of the message is passed to the SVC Handler in a register (the C-callable stub routine takes care of this).

In the message as seen by the receiver, these bytes are over-written by the SVC Handler with the SVid of the target message queue, as passed to the SVC Handler in the register.

bytes 16-19
Ignored by the SVC Handler. These bytes are passed unaltered to the receiver of the message.

To avoid future compatibility problems, these bytes should be set to 0.

bytes 20-23 - *h.msg_id*
Ignored by the SVC Handler. As seen by the receiver, these message bytes are over-written with the message ID as assigned by the system when the message was sent.

bytes 24-27 - *h.dw_count*
The count in 32-bit words of the data in *msg_data* (bytes 32 onwards). This must be a value in the range 0-1024 (inclusive).

bytes 28-31 - *h.msg_type*
The message type, which is a value assigned by the sender. This is passed unaltered to the receiver. These bytes are not used or checked by the SVC Handler.

Return Parameters

message buffer
If the return code is *QSVCgood*, the field *h.msg_id* (bytes 20-23) of the message header is set to the ID of this message, as assigned by the system when this message was sent. Otherwise, the buffer is unchanged from the call.

Return Codes

QSVCgood (0)
Operation completed successfully.

QSVCbadSVid (0x80010002)
Specified SVT entry is not valid, or it is not a task or system message queue. This return code can also occur if the calling task is a “system” task, but it is not permitted to access the specified SVT entry.

QSVCdeadSVid (0x8001000C)
The specified SVT entry is being or has been removed from the system.

QSVCbadreq (0x8001001E)
This can occur for the following reasons:

- The buffer header format type (byte 0) is not 0.
- The data count is >1024 double-words.

QSVCnobuffer (0x8001000E)
Insufficient free space in the MBA to allocate a buffer for this request.

QSVCtoomanymsgs (0x80010039)
The calling task has sent too many messages that have not yet been received (that is, more than the number in the NDA field *max_task_msg*).

This error cannot occur if the code to perform message count checking is omitted from the SVC Handler.

SVC Handler Generated Faults

QSVCinvalid (0x80)
The request priority is less than QSVCMINPRTY (= 32), and the calling task does not have the *TCBsyspriv* bit set.

QSVCparlist (0x81)
The parameter *MsgBuf* does not point to an allocated writeable page.

QSVCinvSVid (0x82)
The caller is not a “system” task, and it is not permitted to access the specified SVT entry.

QSVCmsgcount (0x85)
The calling task has sent too many messages that have not yet been received (that is, more than the number in the NDA field *max_task_msg*).

This fault cannot occur if the code to perform message count checking is omitted from the SVC Handler.

Note: This fault is not generated in the current versions of CP/Q. Instead, the sending task receives the return code *QSVCtoomanymsgs*.

If the specified buffer is not accessible or within its segment limits, or if all or part of the buffer cannot have physical pages allocated to it, the following faults can occur:

- General protection fault (Intel)
- Stack fault (Intel)
- Data storage interrupt (PowerPC)
- Page fault

CPSendRecvMsg (SEND_RECV) - Send a Message and Await a Reply

Function

This call is used to send a message to a message queue or another task, and then wait for a reply to that message. This SVC is equivalent to a SEND_MESG SVC, using a send message buffer, to a task or message queue (not the calling task, because a task cannot do a SEND_RECV to itself), followed by a RECV_MESG to receive a message from the task's message queue, where the only message to be received is one with a type (in bytes 28-31 of the message header) set to the message ID of the sent message. If no such message is received, the calling task waits indefinitely. This SVC should be used only when the caller is sure that the destination task follows the convention of replying to received messages with a message whose type is the ID of the received message.

Each message is associated with a unique message ID, which is assigned by the system when the message is sent. This ID is returned to the sender of a message, and it is included in the message given to the receiver of the message. By convention, when a task sends a message that is a reply to some message that task received, the type field in the header of the reply message is set to the message ID of the original (request) message. Thus, after sending a message, a task might wait for the reply to that specific message by specifying “receive messages with a particular type” and setting the type to the message ID of the sent message.

The specifications of the SEND_MESG and RECV_MESG SVCs should be also examined; SEND_RECV is implemented as a SEND_MESG, immediately followed by a RECV_MESG. The call for SEND_RECV is the same as SEND_MESG except for the following:

- The SVid cannot be zero and cannot be the sender.
- There is a time-out parameter for the receive.
- There is a second message buffer for the receive.
- The message ID is not inserted into the header of the send message buffer.

The return from SEND_RECV is the same as the return from RECV_MESG, except that extra error codes (from SEND_MESG) are possible. If an error is detected during the SEND_MESG phase, no message is sent. However, if an error is not detected until the RECV_MESG phase (for example, the receive buffer is not checked until the start of the receive phase, after the message has been sent) the message was sent, but the calling task does not wait for the reply.

It is possible for *SendMsgBuf* to have the same value as *RecvMsgBuf* (that is, the send and receive buffers might be the same buffer). In this case, the maximum length of the receive message data is the same as the length of the send message data buffer.

If this SVC specifies a time-out, and while the task is waiting for a message a signal arrives that causes a signal handler to be entered and this SVC is then resumed after the return from the signal handler, the time-out is restarted from the beginning. Currently, no attempt is made to restart the time-out for only the remaining period.

It is recommended that the caller's message buffer be aligned on a 32-bit boundary for maximum efficiency. If the program uses CP/Q C heap management routines to allocate the buffers, this alignment is guaranteed.

C Syntax

```
long int CPSendRecvMsg(void *SendMsgBuf,
                       void *RecvMsgBuf,
                       unsigned long int *Count,
                       unsigned long int Timeout);
```

Parameters

SendMsgBuf
: A pointer to a buffer containing a message and header (that is, a *MSG* structure) to be sent. The format and contents of this are the same as for the buffer for a CPSendMsg (SEND_MESG) SVC, except the field *h.msgq_id* cannot be 0, nor may it hold the SVid of the sender.

Count A pointer to a location to receive the count of messages in the queue.

Timeout The wait/no-wait option and time-out period. This accepts the following values:

0 - SVCNOWAIT
No wait, immediate return to the caller (with return code *QSVCQempty*) if the reply message is not immediately available.

0xFFFFFFFF - SVCWAITFOREVER
Wait indefinitely until the reply message is becomes available.

other Wait until the reply is available, but only for the specified time (in µsecs). If this time-out occurs, the SVC Handler returns to the caller with return code *QSVCtimedout*.

Return Parameters

**Count* If the return code is set to *QSVCgood* or *QSVCdeadSVid*, this location is set to the count of messages remaining in the queue after receiving the reply message.

Otherwise, this location is indeterminate.

SendMsgBuf
Unless this is the same buffer as *RecvMsgBuf*, this buffer is unchanged from the call.

RecvMsgBuf
This buffer is set in the same way as a buffer for a RECV_MESG SVC.

Return Codes

QSVCgood (0)
Operation completed successfully.

QSVCbadSVid (0x80010002)
This return code can occur for the following reasons:

- The specified SVT entry is not valid, or it is not a task or system message queue.
- The specified SVid is 0 or is that of the caller.

- If the calling task is a system task, but it is not permitted to access to the specified SVT entry.

QSVCdeadSVid (0x8001000C)
The specified SVT entry is being or has been removed from the system.

QSVCQempty (0x80010009)
The *Timeout* parameter was 0, and the reply message is not immediately available.

QSVCbadreq (0x8001001E)
This can occur for the following reasons:

- The buffer header format type (byte 0) is not 0.
- The data count is >1024 double-words.

This applies to either buffer.

QSVCnobuffer (0x8001000E)
Insufficient free space in the MBA to allocate a buffer for this request.

QSVCsmallbuff (0x80010010)
A message has been returned, but the caller's buffer is too small to receive all of the message data. The returned data count shows how much data was returned. The remaing data is lost.

QSVCtimedout (0x8001000B)
The time-out has occurred.

QSVCinterrupt (0x80010024)
This SVC was interrupted by a signal, and no data has been returned.

QSVCtoomanymsgs (0x80010039)
The calling task has sent too many messages that have not yet been received (that is, more than the number in the NDA field *max_task_msg*).

This error cannot occur if the code to perform message count checking is omitted from the SVC Handler.

SVC Handler Generated Faults

QSVCinvalid (0x80)
The request priority is < QSVCMINPRTY (= 32), and the calling task does not have the *TCBsyspriv* bit set.

QSVCparlist (0x81)
The parameter *MsgBuf* does not point to an allocated writeable page.

QSVCinvSVid (0x82)
The caller is not a system task, and it is not permitted to access to the specified SVT entry.

QSVCmsgcount (0x85)
The calling task has sent too many messages that have not yet been received (that is, more than the number in the NDA field *max_task_msg*).

This fault cannot occur if the code to perform message count checking is omitted from the SVC Handler.

Note: This fault is not generated in the current versions of CP/Q. Instead, the sending task receives the return code *QSVCtoomanymsgs*.

If the specified buffer is not accessible or within its segment limits, or if some of the buffer cannot have physical pages allocated to it, the following faults may occur:

- General protection fault (Intel)
- Stack fault (Intel)
- Data storage interrupt (PowerPC)
- Page fault

CPSplitMsg (SPLIT_SEND) - Send a Message with a Split Message area

Function

This SVC sends a message to a message queue or another task, similar to the CPSendMsg (SEND_MESG) SVC, except that the message header and message data are not contiguous.

The message buffer supplied to this call consists of the header, plus up to four count/pointer pairs which describe the message data areas to be transmitted. The total data count that can be sent in a single message must be in the range 0-1024 32-bit data words. However, for this SVC it can be in up to four separate pieces.

Note: The full 64-byte parameter list must be supplied, even when the count of used buffer pointers is less than four.

An item for this message is added to the target queue, in a position that depends on the new message priority relative to the priorities of messages already on the queue. The new message is inserted in the queue after all messages on the queue with higher (that is, numerically smaller) priority, and immediately after any messages already on the queue with the same priority as the new message.

Note: The priority should be specified as 255 except in those circumstances when higher priority messages really are required. This gives better performance because the new message is then simply added to the end of the queue. This eliminates searching the queue for the correct place to insert the new item.

In this case, the operation of the queue is FIFO (that is, the first message sent is the first received).

The message header from this call is copied into an RQE (Request Queue Element) on the target message queue. The message data is also copied into this RQE if the data count is not greater than the maximum length of a short message (the maximum length of a short message defaults to 7 32-bit words, but can be changed at system build time to be any value in the range 6-16). If the data count is too great for a short message, the data is copied to a system buffer within the Message Buffer Area (MBA). It is more efficient to use short messages, because this eliminates the need to allocate an MBA buffer when sending the message and free this buffer when the message is received. Upon return to the caller from this SVC, the caller's buffer has been copied, and is free for immediate re-use.

Each message is associated with a unique message ID, which is assigned by the system when the message is sent. This ID is returned to the sender of a message, and it is included in the message given to the receiver of the message. By convention, when a task sends a message that is a reply to some message that task received, the type field in the header of the reply message is set to the message ID of the original (request) message. Thus, after sending some message, a task might wait for the reply to that specific message by specifying “receive messages with a particular type,” and setting the type to the message ID of the sent message.

Although this call is non-blocking, if there is a task of higher dispatch priority currently waiting in a RECV_MESG on the destination message queue, and the new message matches the requirements (type, sending task SVid, and so on) of the receiving task, then an immediate task change occurs to that task.

It is recommended that the sender's message buffers be aligned on a 32-bit boundary for maximum efficiency. If the program uses CP/Q C heap management routines to allocate the buffers, this alignment is guaranteed.

C Syntax

```
long int CPSplitMsg(void *MsgBuf);
```

Parameters

MsgBuf A pointer to a buffer to contain a message and header (that is, a *SPLITMSG* structure) as shown in Figure 4-4 on page 4-21.

Fields in this structure must be set up as follows:

byte 0 - *h.HdrVer*
Set to the message header format (currently always 0).

byte 1 - *h.msg_pri*
The required message priority,

The required message priority.

Priority	**Comments**
0-31	Sending task must have *TCBsyspriv* privilege bit set.
32-254	Available to all tasks.
255	Available to all tasks. This is the recommended choice for most messages. The system does not perform an insertion scan of the destination queue, but instead simply adds the message to the end of the queue. This results in a performance savings proportional to the current size of the destination queue.

bytes 2,3 Unused.

bytes 4-7 - *h.send_id*
Ignored by the SVC Handler. In the message seen by the receiver, these bytes are over-written with the SVid of the sender.

bytes 8-11
Ignored by the SVC Handler. These bytes are passed unaltered to the receiver of the message.

To avoid future compatibility problems, these bytes should be set to 0.

bytes 12-15 - *h.msgq_id*
In a call from a C program, this must be set to the SVid of the message queue or task to receive this message.

At the assembler interface level, this field is ignored by the SVC Handler; the destination of the message is passed to the SVC Handler in a register (the C-callable stub routine takes care of this).

In the message seen by the receiver, these bytes are over-written by the SVC Handler with the SVid of the target

```
typedef struct splitent
{
  unsigned long int dw_count;    /* data count in 32-bit words         */
  void              *pdata;      /* pointer to separate data area      */
} SPLITENT;

typedef struct qmsghdr
{
  unsigned char      HdrVer;      /* Header version                      */
  unsigned char      msg_pri;     /* Message priority                    */
  unsigned short int rsrv1;       /* (reserved)                          */
  unsigned long int  send_id;     /* Sender's task SVid                  */
  unsigned long int  rsrv2;       /* (reserved)                          */
  unsigned long int  msgq_id;     /* Message queue SVid (0 = self)       */
  unsigned long int  rsrv3;       /* (reserved)                          */
  unsigned long int  msg_id;      /* Unique, system assigned Message ID  */
  unsigned long int  dw_count;    /* Data area length (not incl. header) */
  unsigned long int  msg_type;    /* Message type (user defined field)   */
} QMSGHDR;

typedef struct splitmsg
{
  QMSGHDR h;
  SPLITENT e[4];
} SPLITMSG;
```

Figure 4-4. Split Message Structures

message queue, as passed to the SVC Handler in the register.

bytes 16-19
Ignored by the SVC Handler. These bytes are passed unaltered to the receiver of the message.

To avoid future compatibility problems, these bytes should be set to 0.

bytes 20-23 - *h.msg_id*
Ignored by the SVC Handler. In the message seen by the receiver, this field is over-written with the message ID assigned by the system when the message is sent.

bytes 24-27 - *h.dw_count*
The number of used count/pointer pairs in *MsgBuf→e* (bytes 32 onwards). This must be a value in the range 1-4 inclusive.

bytes 28-31 - *h.msg_type*
The message type, which is a value assigned by the sender. This is passed unaltered to the receiver. These bytes are not used or checked by the SVC Handler.

In addition, *h.dw_count* count/pointer pairs are used in the vector *MsgBuf→e*, each of which describes a separate data area to be sent as part of this message. Each of these entries has the following format:

bytes 0-3 - *e.dw_count*
The count in 32-bit words of the data in this data area. This value must be in the range 0-1024 inclusive, but it is also subject to the limit that the total count of all the used data descriptors must not exceed 1024 32-bit words.

bytes 4-7 - *e.pdata*
A pointer to the data area.

Return Parameters

message buffer
If the return code is *QSVCgood*, the field *h.msg_id* (bytes 20-23) of the message header is set to the ID of this message, as assigned by the system when this message was sent. Otherwise, the buffer is unchanged from the call.

Return Codes

QSVCgood (0)
Operation completed successfully.

QSVCbadSVid (0x80010002)
Specified SVT entry is not valid, or it is not a task or system message queue. This return code can also occur if the calling task is a system task, but it is not permitted to access to the specified SVT entry.

QSVCdeadSVid (0x8001000C)
The specified SVT entry is being or has been removed from the system.

QSVCQempty (0x80010009)
Specified queue is empty, or a message with the specified ID, type, or sender cannot be found on the queue

QSVCbadreq (0x8001001E)
This can occur for the following reasons:

- The buffer header format type (byte 0) is not 0.
- The total message data count is >1024 double-words.
- The number of count/pointer pairs is >4.

QSVCnobuffer (0x8001000E)
Insufficient free space in the MBA to allocate a buffer for this request.

QSVCtoomanymsgs (0x80010039)
The calling task has sent too many messages that have not yet been received (that is, more than the number in the NDA field *max_task_msg*).

This error cannot occur if the code to perform message count checking is omitted from the SVC Handler.

SVC Handler Generated Faults

QSVCinvalid (0x80)
The request priority is < QSVCMINPRTY (= 32), and the calling task does not have the *TCBsyspriv* bit set.

QSVCparlist (0x81)
The parameter *MsgBuf* does not point to an allocated writeable page.

QSVCinvSVid (0x82)
: The caller is not a system task, and it is not permitted to access to the specified SVT entry.

QSVCmsgcount (0x85)
: The calling task has sent too many messages that have not yet been received (that is, more than the number in the NDA field *max_task_msg*).

  This fault cannot occur if the code to perform message count checking is omitted from the SVC Handler.

  Note: This fault is not generated in the current versions of CP/Q. Instead, the sending task receives the return code *QSVCtoomanymsgs*.

If the specified buffer is not accessible or within its segment limits, or if all or part of the buffer cannot have physical pages allocated to it, the following faults can occur:

- General protection fault (Intel)
- Stack fault (Intel)
- Data storage interrupt (PowerPC)
- Page fault

CPSplitRecv (SPLIT_RECV) - Receive a Message with a Split Buffer

Function

This call obtains an item from a request queue. Receiving the message removes it from the message queue. This SVC is very similar to CPRecvMsg (RECV_MESG), except that the buffer to receive the message is divided into two parts, namely a buffer for the message header, and a separate buffer to contain the message data.

The task issuing this call must provide a buffer large enough to receive the header and another buffer message data. It is recommended that the receiver's buffers be aligned on a 32-bit boundary for maximum efficiency. If the program uses CP/Q C heap management routines to allocate the buffer, this alignment is guaranteed. If the data count of the incoming message is larger than the supplied data buffer, only that portion of the message that fits in the supplied buffer is returned. The remaining data is lost. In this case, the return code is *QSVCsmallbuff*.

This SVC (by default) returns the first message on the queue. However, by specifying a non-zero *Mode* parameter and setting appropriate values in the supplied message header, a certain specific message can be received, or only messages from a specified task, or only messages with the specified type.

Each message is associated with a unique message ID, which is assigned by the system when the message is sent. This ID is returned to the sender of a message, and it is included in the message given to the receiver of the message. By convention, when a task sends a message that is a reply to some message that task received, the type field in the header of the reply message is set to the message ID of the original (request) message. Thus, after sending some message, a task might wait for the reply to that specific message by specifying “receive messages with a particular type” and setting the type to the message ID of the sent message.

If this SVC specifies a time-out, and while the task is waiting for a message a signal arrives that causes a signal handler to be entered and this SVC is then resumed after the return from the signal handler, the time-out is restarted from the beginning. Currently, no attempt is made to restart the time-out for only the remaining period.

This call is a potentially blocking function call, which means that if the queue is empty, or it does not have a message with a matching message ID, type value, or sender ID, the task may block within the SVC Handler until a suitable message arrives. There are two ways to avoid an indefinite blocking situation. First, the caller can use the timeout parameter as described above. The other method is to use the CPCountMsg function call. This call (described above) is non-blocking in nature and returns the number of messages in a particular queue.

A task can also examine messages using the CPPeekMsg call.

C Syntax

```
long int CPSplitRecv(void *MsgBuf,
                     unsigned long int *Count,
                     unsigned long int Mode,
                     unsigned long int Timeout);
```

```
typedef struct splitent
{
  unsigned long int dw_count;    /* data count in 32-bit words         */
  void              *pdata;      /* pointer to separate data area      */
} SPLITENT;

typedef struct qmsghdr
{
  unsigned char      HdrVer;      /* Header version                     */
  unsigned char      msg_pri;     /* Message priority                   */
  unsigned short int rsrv1;       /* (reserved)                         */
  unsigned long int  send_id;     /* Sender's task SVid                 */
  unsigned long int  rsrv2;       /* (reserved)                         */
  unsigned long int  msgq_id;     /* Message queue SVid (0 = self)      */
  unsigned long int  rsrv3;       /* (reserved)                         */
  unsigned long int  msg_id;      /* Unique, system assigned Message ID  */
  unsigned long int  dw_count;    /* Data area length (not incl. header) */
  unsigned long int  msg_type;    /* Message type (user defined field)   */
} QMSGHDR;

typedef struct splitmsg
{
  QMSGHDR h;
  SPLITENT e[4];
} SPLITMSG;
```

Figure 4-5. Split Message Structures

Parameters

MsgBuf A pointer to a buffer to contain a message and header (that is, a *SPLITMSG* structure) as shown in Figure 4-5.

Fields in this structure must be set up as follows:

byte 0 - *h.HdrVer*
Must be set to the message header format (currently always 0).

bytes 4-7 - *h.send_id*
If *Mode* bit 5 is set, this field must be set to the SVid of the desired sending task. Otherwise, this field is ignored in the call.

bytes 12-15 - *h.msgq_id*
Must be set to the SVid of the message queue to be searched; this may also be the SVid of the caller or zero. If zero is specified, the message queue of the calling task is used.

bytes 20-23 - *h.msg_id*
If *Mode* bit 7 is set, this field must be set to the message ID of the message to be peeked. Otherwise, this field is ignored in the call.

bytes 24-27 - *h.dw_count*
The number of count/pointer pairs in the vector *MsgBuf→e*. Currently, this must be set to 1.

bytes 28-31 - *h.msg_type*
If *Mode* bit 6 is set, this field holds the desired message type. Otherwise, this field is ignored in the call.

bytes 32-35 - *e[0].dw_count*
The size of the buffer to receive the message data, in 32-bit words.

bytes 36-39 - *e[0].pdata*
A pointer to the buffer to receive the message data.

Other fields in the *MsgBuf* structure are ignored by the SVC Handler.

Count A pointer to a location to receive the count of messages in the queue.

Mode An 8-bit bit significant field, as follows:

bit 7 - QSVCmsg_id (0x01)
If set, this SVC receives only a message with the message ID contained in the field *msg_id* (bytes 20-23) of the parameter list. If this bit is set, bits 0-6 must be 0, and *timeout* must also be 0.

bit 6 - QSVCmsg_type (0x02)
If set, this SVC receives only a message with the message type specified in the field *msg_type* (bytes 28-31) of the parameter list.

If this bit is set, bit 5 can also be set. This further restricts the selection to messages from a specific sender.

bit 5 - QSVCmsg_sender (0x04)
If set, this SVC peeks only a message sent by the task whose SVid is specified in the field *send_id* (bytes 4-7) of the parameter list.

If this bit is set, bit 6 can also be set. This further restricts the selection to messages with a specific type.

If this field is 0, the system call receives the first message on the specified queue.

Timeout The wait/no-wait option and time-out period. This accepts the following values: if *Mode* bit 7 is set, only the value 0 is permitted for *Timeout*.

0 - SVCNOWAIT
No wait, immediate return to the caller (with return code *QSVCQempty*) if there is no message available.

0xFFFFFFFF - SVCWAITFOREVER
Wait indefinitely until a message arrives.

other Wait until a message becomes available, but only for the specified time (in µsecs). If this time-out occurs, the SVC Handler returns to the caller with return code *QSVCtimedout*.

Return Parameters

******Count* If the return code is set to *QSVCgood* or *QSVCdeadSVid*, this location is set to the count of messages remaining in the queue after removing this message.

Otherwise, this location is indeterminate.

message buffer

If the return code is *QSVCgood*, the fields of the message header buffer are set as follows. Otherwise, the buffer is unchanged from the call.

byte 0 - *h.HdrVer*
Set to 0 (the message buffer format).

byte 1 - *h.msg_pri*
Set to the message priority.

bytes 2,3 Indeterminate.

bytes 4-7 - *h.send_id*
Set to the SVid of the sender of the message.

bytes 8-11
Indeterminate.

Note: These bytes contain whatever was in bytes 4-7 of the sender's message buffer.

bytes 12-15 - *h.msgq_id*
The target SVid of this message. In the case of a message from the caller's message queue, this is the SVid of the calling task.

bytes 16-19
Indeterminate.

Note: These bytes actually contain whatever was in bytes 16-19 of the sender's message buffer.

bytes 20-23 - *h.msg_id*
The ID of this message, as assigned by the system when this message was sent.

bytes 24-27 - *h.dw_count*
The number of count/pointer pairs in the vector *MsgBuf→e*. This is set to 1 (it is unchanged from the call).

bytes 28-31 - *h.msg_type*
The type of this message, as specified in bytes 28-31 of the sender's message buffer.

bytes 32-35 - *e[0].dw_count*
The count of 32-bit words received in the data buffer.

bytes 36-39 - *e[0].pdata*
A pointer to the buffer to receive the message data. This is unchanged from the call.

Return Codes

QSVCgood (0)
: Operation completed successfully.

QSVCbadSVid (0x80010002)
: Specified SVT entry is not valid, or it is not a task or system message queue. This return code can also occur if the calling task is a system task, but it is not permitted to access to the specified SVT entry.

QSVCdeadSVid (0x8001000C)
: The specified SVT entry is being or has been removed from the system.

QSVCQempty (0x80010009)
: Specified queue is empty, or a message with the specified ID, type, or sender cannot be found on the queue.

QSVCbadreq (0x8001001E)
: This can occur for the following reasons:

- *Mode* bit 7 is set and AL bits 0-6 are not all unset.
- *Mode* bit 7 is set and *Timeout* ≠ 0.
- *Mode* bit 7 is unset and bit 5, 6, or both are set, and bits 0-4 are not all unset.
- The buffer header format type (byte 0) is not 0.
- The number of count/pointer pairs is not one.
- The data count is >1024 double-words.

QSVCsmallbuff (0x80010010)
: A message has been returned, but the caller's buffer is too small to receive all of the message data. The returned data count shows how much data was returned. The remaining data is lost.

QSVCtimedout (0x8001000B)
: The time-out has occurred.

QSVCinterrupt (0x80010024)
: This SVC was interrupted by a signal, and no data has been returned.

SVC Handler Generated Faults

QSVCparlist (0x81)
: The parameter *MsgBuf* does not point to an allocated writeable page.

QSVCinvSVid (0x82)
: The caller is not a system task, and it is not permitted to access to the specified SVT entry.

If the specified buffer is not accessible or within its segment limits, or if all or part of the buffer cannot have physical pages allocated to it, the following faults can occur:

- General protection fault (Intel)
- Stack fault (Intel)
- Data storage interrupt (PowerPC)
- Page fault

Chapter 5. Resource/Serialization Semaphore Operations

CPSemClaim (CLAIMSEM) - Claim a Semaphore

Function

This call claims a resource serialization semaphore; it is used to guarantee serial access to shared resources.

If the semaphore is free, it is claimed by the caller. Otherwise, the caller can wait until it becomes available or receive an immediate error return.

If this SVC specifies a time-out, and while the task is waiting for the semaphore a signal arrives that causes a signal handler to be entered and this SVC is then resumed after the return from the signal handler, the time-out is restarted from the beginning. Currently, no attempt is made to restart the time-out for only the remaining period.

C Syntax

```
long int CPSemClaim(unsigned long int SVid,
                    unsigned long int Timeout);
```

Parameters

SVid The SVid of the semaphore to be claimed.

Timeout The wait/no-wait option and time-out period. This accepts the following values:

0 - SVCNOWAIT
No wait, immediate return to the caller with return code *QSVCsembusy* if the semaphore is set, or return code *QSVCgood* if the semaphore is clear.

0xFFFFFFFF - SVCWAITFOREVER
Wait indefinitely until the semaphore next clears.

other Wait until the semaphore is next cleared, but only for the specified time (in µsecs). If this time-out occurs, the SVC Handler returns to the caller with return code *QSVCtimedout*.

Return Codes

QSVCgood (0)
Operation completed successfully.

QSVCsembusy (0x80010012)
The semaphore is already claimed, and it is not available (*Timeout*=0, immediate return case, only).

QSVCbadSVid (0x80010002)
Specified SVT entry is not valid, or it is not a resource semaphore. This return code can also occur if the calling task is a "system" task, but it is not permitted to access to the specified SVT entry.

QSVCdeadSVid (0x8001000C)
The specified SVT entry is being or has been removed from the system.

This can occur if the semaphore is marked as “dying” when the call is first issued, or if the semaphore is removed later, while the requesting task is waiting on the task queue of the semaphore.

QSVCtimedout (0x8001000B)
The time-out has occurred.

QSVCinterrupt (0x80010024)
This SVC was interrupted by a signal, and the semaphore has not been claimed by the caller.

SVC Handler Generated Faults

QSVCinvSVid (0x82)
The caller is not a “system” task, and it is not permitted to access to the specified SVT entry.

CPSemQuery (QRY_SEM) - Query the State of a Semaphore

Function

This call obtains the status of a serialization semaphore. The SVid of the holder is returned if the semaphore is claimed, and 0 if it is free.

C Syntax

```
long int CPSemQuery(unsigned long int SVid,
                    unsigned long int *ClaimedBy);
```

Parameters

SVid	The SVid of the semaphore to be queried.
ClaimedBy	A pointer to a location to receive the returned SVid.

Return Parameters

Claimedby	If the return code is *QSVCgood*, this location is set to hold 0 (if the semaphore is currently free) or the SVid of the task currently holding the semaphore if it is claimed. Otherwise, this location is indeterminate.

Return Codes

QSVCgood (0)
Operation completed successfully.

QSVCbadSVid (0x80010002)
The specified SVT entry is not valid, or it is not a resource semaphore. This return code can also occur if the calling task is a system task, but it is not permitted to access to the specified SVT entry.

QSVCdeadSVid (0x8001000C)
The specified SVT entry is being or has been removed from the system.

SVC Handler Generated Faults

QSVCinvSVid (0x82)
The caller is not a system task, and it is not permitted to access to the specified SVT entry.

CPSemRelease (REL_SEM) - Release a Semaphore

Function

This call releases a previously claimed resource serialization semaphore. If there are tasks queued waiting for this semaphore, the semaphore is given to the first task on the queue.

If the queue for the semaphore is not empty, and the task that would be given the semaphore is stopped, or it is “pending stopped” and the task is not in a critical section, then the semaphore is not given to that task. The semaphore is given to the next task on the queue (if any).

Note: No check is made that the semaphore is being released by the task that claimed it.

For example, this means that task A could claim a semaphore, and then issue a request to claim it again. The second claim causes task A to be suspended until the semaphore becomes free. Task B could, on completion of some action, release the semaphore. This causes task A’s second claim request to complete, and task A then resumes execution.

This SVC is allowed, even if the semaphore is marked as “dying”. However, in this situation, the semaphore is not given to the next task on the queue. It is assumed that a task is attempting to clean up and remove the semaphore.

C Syntax

```
long int CPSemRelease(unsigned long int SVid);
```

Parameters

SVid	The SVid of the semaphore to be released.

Return Codes

QSVCgood (0)
Operation completed successfully.

QSVCbadSVid (0x80010002)
The specified SVT entry is not valid, or itis not a resource semaphore. This return code can also occur if the calling task is a system task, but it is not permitted to access to the specified SVT entry.

QSVCnclaim (0x80010005)
The semaphore is already free.

SVC Handler Generated Faults

QSVCinvSVid (0x82)
The caller is not a system task, and it is not permitted to access to the specified SVT entry.

Chapter 6. Id Resolution and Translation

CPCheckID (CHECK_ID) - Query and Check Access to an SVT Entry

Function

This call queries an SVid to obtain the type of system object specified by this SVT entry and checks that the caller can successfully access the specified SVT entry. If the caller can access the SVT entry, the return code is *QSVCgood* and the type of the SVT entry (the SVT entry field *SVTtype*) is returned. If an error occurs while attempting to access to this SVT entry (for example, the SVT entry is not valid, unused, or the caller is not permitted access to the entry), an error return code is given.

The calling task is never faulted by this call.

C Syntax

```
long int CPCheckID(unsigned long int SVid,
                   unsigned long int *Type,
                   unsigned long int *ProcID);
```

Parameters

SVid	The SVid of the SVT entry to be queried.
Type	A pointer to a location to receive the returned SVT entry type.
ProcID	A pointer to a location to receive the returned process ID.

Return Parameters

*****Type
: If the return code is *QSVCgood* or *QSVCnowrite*, this location is set to the type of the SVT entry. Otherwise, this location is unchanged.

The following values can be returned in this field:

0	Task
1	Resource/serialization semaphore
2	Synchronization semaphore
4	User-defined SVT item
6	Message queue

*****ProcID
: If the return code is *QSVCgood* or *QSVCnowrite*, this location is set to the ID of the process owning the specified SVT entry. Otherwise, this location is unchanged.

Return Codes

QSVCgood (0)
: Operation completed successfully.

QSVCnowrite (0x8001000F)
: The SVT entry exists (and its SVid is being returned to the caller), and the caller has read access but no write access to the SVT entry. For example, the caller cannot send messages to this task or queue.

QSVCbadSVid (0x80010002)

This error code indicates one of the following:

- There is no SVT entry with this name.
- The specified SVid is not valid in some manner
- This SVT entry is in some way inaccessible to the requestor
  - The SVT entry is marked as restricted to access from supervisor mode code, and the caller is user mode.
  - The SVT entry is marked as restricted to access from a process other than that of the caller.
  - The SVT entry is marked as restricted to access from a task other than the caller.

QSVCdeadSVid (0x8001000C)

The specified SVT entry is being or has been removed from the system.

SVC Handler Generated Faults

None

CPQueryID (QRY_ID) - Obtain SVid of Caller

Function

This call returns various items of information about the caller's environment.

C Syntax

```
long int CPQueryID(unsigned long int *SVid,
                   unsigned long int *TCBID,
                   unsigned long int *ProcID,
                   unsigned long int *EnvTabAddr);
```

Parameters

SVid	A pointer to a location to receive the caller's SVid
TCBID	A pointer to a location to receive the caller's TCB ID
ProcID	A pointer to a location to receive the caller's process ID
EnvTabAddr	A pointer to a location to receive the address offset of the caller's parameter area (set to 0 if none)

Return Parameters

**SVid*	Set to the caller's SVid
**TCBID*	Set to the caller's TCB ID
**ProcID*	Set to the caller's process ID
**EnvTabAddr*	Set to the address offset of the caller's parameter area (0 if none)

Return Codes

QSVCgood (0)
Operation completed successfully.

SVC Handler Generated Faults

none

CPResolveID (ID_NAME) - Obtain SVT Name Corresponding to an SVid

Function

This call returns the name of an SVT entry. If the SVid is not valid, an error return code is given.

If the SVT entry is marked as being removed from the system, or if the SVT entry is not marked as having a name, the error return code *QSVCdeadSVid* or *QSVCnoname* is given, but the name field from the SVT entry is still returned. This name can be non-null (even if the entry is not “named”). This can be determined by setting the 8-byte buffer to all spaces before making the SVC and testing for spaces afterwards.

C Syntax

```
long int CPResolveID(unsigned long int SVid,
                     char *NameAddr);
```

Parameters

SVid The SVid of the SVT entry whose name is required.

Nameaddr
A pointer to an 8-byte buffer where the name is returned.

Note: This buffer must have write access.

Return Parameters

**NameAddr*
If the return code is *QSVCgood* or *QSVCnoname*, the buffer is over-written with the name field of the SVT entry.

Otherwise, this buffer is unaltered.

Return Codes

QSVCgood (0)
Operation completed successfully.

QSVCbadSVid (0x80010002)
The specified SVid is not valid in some way.

QSVCnoname (0x80010032)
The specified SVid is not marked as “named” (but the name field is still returned in the buffer).

QSVCdeadSVid (0x8001000C)
The specified SVT entry is being removed from the system, but the name field is still returned in the buffer.

SVC Handler Generated Faults

QSVCparlist (0x81)
The parameter list is not valid. For example, the pointer does not refer to an allocated page, or the page or pages specified by the parameter are not writeable by the caller.

CPResolveName (GET_ID) - Obtain SVid Corresponding to an SVT Name

Function

This call resolves an SVT name to an SVid. The SVT is scanned to locate the name specified in the first parameter. Any SVT entries that are currently marked as being removed from the system are ignored in this scan. Furthermore, if the specified name exists in an SVT entry that is not marked as having a name, the GET_ID SVC returns an error.

A complication can arise with the GET_ID SVC, particularly during system start-up, whereby the SVT entry that the caller wants to access does not exist, even though it is known that it will be created at some point during system initialization (this is sometimes called a start-up “race” condition). To eliminate these start-up race problems, the SVC Handler implements a queue associated with the GET_ID SVC, whereby a task seeking the SVid of something has the option of waiting until the required item is created. This “GET_ID” queue is a single system-wide queue. It is scanned (assuming it contains something) every time an SVT entry is created and checks if the new SVT entry satisfies a pending request. If the specified SVT entry does not exist, the caller might receive an immediate error return or might be placed on the system GET_ID queue and suspended until either the desired SVT entry is created or until a specified time-out occurs.

The SVC Handler automatically claims and releases the system data area semaphore during this SVC. This has the effect that the calling task can be suspended by this SVC until this semaphore becomes free.

In general, if the specified SVT entry exists, the SVid, PCB ID, and SVT entry type are returned even if an error occurs. If this information is not returned, the returned SVid is explicitly set to 0.

If this SVC specifies a time-out, and while the task is waiting for the SVT entry to be created a signal arrives that causes a signal handler to be entered and this SVC is then resumed after the return from the signal handler, the time-out is restarted from the beginning. Currently, no attempt is made to restart the time-out for only the remaining period.

C Syntax

```
long int CPResolveName(char *NameAddr,
                       unsigned long int *SVid,
                       unsigned long int *Type,
                       unsigned long int *ProcID,
                       unsigned long int Timeout);
```

Parameters

NameAddr
: A pointer to an 8-byte buffer containing the name to be resolved. If the name is less than eight characters, it should be terminated by a space or NUL character. The buffer can be read-only.

SVid
: A pointer to a location to receive the returned SVid.

Type
: A pointer to a location to receive the returned SVT entry type.

ProcID	A pointer to a location to receive the returned process ID.
Timeout	The wait/no-wait option and time-out period. This accepts the following values:

0 - SVCNOWAIT
: No wait. If the required SVT entry does not exist, an immediate return to the caller occurs (with return code *QSVCbadSVid*).

0xFFFFFFFF - SVCWAITFOREVER
: Wait indefinitely until the SVT entry is created.

other
: Wait until the SVT entry is created, but only for the specified time (in μsecs). If this time-out occurs, the SVC Handler returns to the caller with return code *QSVCtimedout*.

Return Parameters

******SVid* The returned SVid.

******Type* This location is set to the type of the SVT entry.

The following values can be returned in this field:

0	Task
1	Resource/serialization semaphore
2	Synchronization semaphore
4	User-defined SVT item
6	Message queue

******ProcID* This location is set to the ID of the process owning the specified SVT entry.

Note: Generally, when an error occurs, the return information is given even though there is an error return code. If the information is not being returned, the returned SVid is explicitly set to 0.

Return Codes

QSVCgood (0)
Operation completed successfully.

QSVCbadname (0x8001000D)
The specified SVT name is not valid (for example, it is all spaces).

QSVCnowrite (0x8001000F)
The SVT entry exists (and its SVid is being returned to the caller), and the caller has read access but no write access to the SVT entry. For example, the caller cannot send messages to this task or queue.

QSVCbadSVid (0x80010002)
This error code indicates one of the following:

- The supplied SVT name cannot be found in the SVT.
- The SVT entry with this name is not marked as having a name.
- The caller does not have read access to this SVT entry
  - The SVT entry is marked as restricted to supervisor mode, and the caller is user mode.
  - The SVT entry is marked as restricted to access from a process other than that of the caller.
  - the SVT entry is marked as restricted to access from a task other than the caller.

QSVCtimedout (0x8001000B)
The time-out has occurred.

QSVCinterrupt (0x80010024)
This SVC has been interrupted by a signal. No SVid or other information has been returned (EDX is 0).

SVC Handler Generated Faults

QSVCparlist (0x81)
The parameter list is not valid. For example, the pointer does not reference an allocated page.

CPSVid2TCB and CPTCB2SVid (TRAN_ID) - Task Id/SVid Translation

Function

This call translates the task ID (TCB offset) of a task into its SVid, or vice-versa. This is required by tasks such as a command process which needs to print the name of a stopped or faulted task.

The restrictions implied by the SVT entry access control bits being set do not apply to this SVC.

C Syntax

```
long int CPSVid2TCB(unsigned long int SVid,
                    unsigned long int *TCBID,
                    unsigned long int *ProcID);

long int CPTCB2SVid(unsigned long int TCBID,
                    unsigned long int *SVid,
                    unsigned long int *ProcID);
```

Parameters

SVid	The SVid of the task to be translated (CPSVid2TCB), or a pointer to a location to receive the returned SVid (CPTCB2SVid)
TCBID	A pointer to a location to receive the returned Task ID (CPSVid2TCB), or the Task ID to be translated (CPTCB2SVid)
ProcID	A pointer to a location to receive the returned process ID

Return Parameters

*****_SVid_	Set to the SVid of the task
*****_TCBID_	Set to the Task ID of the task
*****_ProcID_	Set to the ID of the process owning the specified task

Return Codes

QSVCgood (0)
: Operation completed successfully.

QSVCbadSVid (0x80010002)
: Specified SVT entry is not valid, or is not a task, or the specified TCB offset is not valid, or the specified task identifier refers to a TCB that is not (or never has been) in use.

QSVCbadreq (0x8001001E)
: The AL register (Intel) or R6 (POWER/PowerPC) is not 1 or 2.

SVC Handler Generated Faults

None

Chapter 7. Signal Operations

There are no "*CPxxxx*" C callable routines for these signal facilities in CPQLIB. The routines for these calls are in UNIXLIB.LIB, with declarations in SYS\SIGNAL.H.

See the SVC Handler manual for more details about signals and their implementation in CP/Q.

CPSigInt (SIG_INTERRUPT) - Signal is/is not to Interrupt SVCs

Function

This call enables and disables the interrupts of other SVCs by signals. Normally, if a signal interrupts an SVC upon return from the signal handler, the suspended SVC is resumed from the point of interruption. This system call (with a flag value of 1) has the result that, on return from a handler for the specified signal, the current SVC is interrupted (that is, terminated), and an immediate return to the calling task occurs with the return code set to *QSVCinterrupt*.

This flag is also set or cleared by the SIG_VEC SVC.

This affects only those system calls that might suspend the task, such as RECV_MESG or CLAIM_SEM. It does not affect non-blocking calls, such as SEND_MESG.

C Syntax

```
#include <sys\signal.h>

int siginterrupt(int sig,
                 int flag);
```

Parameters

sig	The number of the signal to be set. This can be a number in the range 1-31 other than 5, 7, 10, 23, 25, 26 or 27. See the signal name table in the section "CP/Q Signal Facilities" in the SVC Handler manual for a list of the signal names and numbers.
flag	This can be set to 0 to indicate that signals are not to interrupt long-running SVCs (that is, those that might be interrupted by a signal). When the return from the signal handler occurs, the task resumes the SVC from the point of interruption. Resumption of SVCs is the default behavior in CP/Q.
	This parameter can also be set to 1. This means that a signal causes the SVC to be interrupted. Upon return from the signal handler, the SVC Handler returns immediately to the SVC caller with return code *QSVCinterrupt*, and the SVC was not completed.

Return Parameters

The only value returned by this routine is its return code. This is 0 for successful completion, or -1 in case of error. In the case of an error, *errno* is set to indicate the error.

EINVAL The specified signal number is not valid.

Return Codes

QSVCgood (0)
Operation completed successfully.

QSVCbadsignal (0x80010033)
The signal number is not valid.

QSVCbadreq (0x8001001E)
AL is not set to 0 or 1.

QSVCnfSCB (0x80010035)
The specified signal does not have an SCB allocated, and no free SCBs are available.

SVC Handler Generated Faults

none

CPSigMask (SIG_MASK) - Set or Query Signal Mask

Function

This call sets or clears the bits of the signal mask or queries its current state.

This SVC does not set the mask bits for SIGKILL, SIGSTOP or SIGCONT signals. Also, it does not set the bits for those signals not used in CP/Q, namely signals 0, 5, 7, 10, 23, 25, 26 and 27. See the signal name table in the section "CP/Q Signal Facilities" in the SVC Handler manual for a list of the signal names and numbers.

This SVC returns the previous signal mask and the set of pending signals that are currently blocked from delivery.

C Syntax

```
#include <sys\signal.h>

int sigblock(int mask);

int sigpause(int mask);

int sigsetmask(int mask);
```

C Routine Descriptions

`sigblock` This call blocks signals by setting bits in the signal mask of the calling task. The signals specified by *mask* are added to (that is ORed into) the set of signals currently blocked from delivery. A signal is blocked if the corresponding bit in *mask* is set to 1.

It is not possible to block SIGKILL, SIGSTOP or SIGCONT signals or the signals that are unused in CP/Q.

`sigpause` This call atomically releases a set of blocked signals and waits for a signal to arrive. *mask* is temporarily assigned to the current signal mask for the calling task, and the task waits for an unmasked signal to arrive. Upon return from the signal handler, the signal mask is restored to its original value.

`sigsetmask`

This call sets the signal mask of the calling task to the supplied *mask*. If the corresponding bit in *mask* is set to 1, a signal is blocked from delivery.

It is not possible to block SIGKILL, SIGSTOP or SIGCONT signals or the signals that are unused in CP/Q.

Parameters

mask A signal mask. This is a bit significant value, with bits set according to which signals the call applies to. The macro

```
sigmask(signal)
```

can be used to obtain the bit mask for a specific signal. See the signal name table in the section "CP/Q Signal Facilities" in the SVC Handler manual for a list of the signal names and numbers.

Return Parameters

The only values returned by these routines are their return codes. This is 0 for successful completion or -1 in case of error. In the case of an error, *errno* is set to indicate the error.

The routine `sigpause`, in the case of an otherwise successful call, returns the value -1, with *errno* set to **EINTR**.

Return Codes

QSVCgood (0)

Operation completed successfully.

If AL = 3, an unmasked signal was received.

QSVCbadreq (0x8001001E)

This can be for one of the following reasons:

- AL is not set to 0, 1, 2 or 3.
- AL=3 and the task is already in a signal handler.

SVC Handler Generated Faults

none

CPSigReturn (SIG_RETURN) - Return from a Signal Handler

Function

This call returns from a signal handler. This SVC may be used *only* when running in a signal handler and does *not* return to the caller. Instead, the task's signal mask, stack pointers, and instruction pointer are restored from the *sigcontext* structure.

Normally, it is not necessary for a signal handler to include an explicit SIG_RETURN SVC. All that is needed is for the signal handler routine to return, whereupon an implicit SIG_RETURN is implemented. See the description of the signal facilities in the SVC Handler manual for more information about writing a signal handler in C.

C Syntax

```
#include <sys\signal.h>

int sigreturn(struct sigcontext *scp);
```

Parameters

scp A pointer to a *sigstruct* structure. Normally, the SVC handler generates this structure when the signal handler was entered (modified, if necessary).

This structure is defined in SYS\SIGNAL.H, thus:

```
struct sigcontext
{
  unsigned long sc_onstack;    /* sigstack state to restore    */
  unsigned long sc_mask;       /* signal mask to restore       */
  unsigned long sc_ebp;        /* register of interrupted task */
  unsigned long sc_esp;        /* register of interrupted task */
  unsigned long sc_eip;        /* register of interrupted task */
};
```

Return Parameters

None. This call does *not* return.

Return Codes

none

SVC Handler Generated Faults

none

CPSigSend (SIG_SEND) - Send a Signal

Function

This SVC sends a specified signal to a task. If a task other than the caller is specified, then either the task issuing this SVC must have the *TCBstoppriv* privilege bit set, or the target task must be either in the same process as the caller or in a process that is a descendent of the caller's process.

C Syntax

```
#include <sys\signal.h>

int kill(int SVid,
         int sig);
```

Parameters

SVid The SVid of the task to receive the signal.

sig The signal to be sent. This can be a number in the range 1-31, other than 5, 7, 10, 23, 25, 26 or 27. See the signal name table in the section "CP/Q Signal Facilities" in the SVC Handler manual for a list of the signal names and numbers.

This can also be 0. In the case of 0, validity of the operation is checked, but no signal is sent.

Return Parameters

The only value returned by this routine is its return code. This is 0 for successful completion, or -1 in case of error In the case of an error, *errno* is set to indicate the error.

EINVAL The specified signal number is not valid.

ESRCH The specified SVid is not valid, or it is not that of a task.

Return Codes

QSVCgood (0)
Operation completed successfully.

QSVCbadsignal (0x80010033)
The signal number is not valid.

QSVCbadSVid (0x80010002)
This error code signifies one of the following:

- The specified SVid is not valid in some way, for example, it is not a task.
- This SVT entry is inaccessible to the requestor in one of the following ways:
  - The SVT entry is marked as restricted to access from supervisor mode code, and the caller is user mode.
  - The SVT entry is marked as restricted to access from a process other than that of the caller.
  - The SVT entry is marked as restricted to access from a task other than the caller.

QSVCunpriv (0x80010031)
: Caller does not have the *TCBstoppriv* privilege bit set, and the specified SVT entry is not in the same process as the caller nor in a process that is a descendent of the caller's process.

QSVCdeadSVid (0x8001000C)
: The specified SVT entry is being removed from the system.

QSVCnotinit (0x80010015)
: This signal involves stopping or starting the target task, but the task has not been initialized.

QSVCtaskstop (0x80010013)
: This signal involves stopping the target task, but it is already stopped.

QSVCtaskgo (0x80010014)
: This signal involves starting the target task, but it is already going.

SVC Handler Generated Faults

none

CPSigStack (SIG_STACK) - Set or Query Signal Stack

Function

This SVC sets up (that is, informs the SVC Handler of) a signal stack or query the current signal stack. This allows users to specify an alternate stack to be used when operating in a signal handler. When the action of a signal indicates that the handler is to execute on the signal stack, when the signal arrives the system switches the task to the specified stack if it is not already running on that stack.

When this SVC is to set a signal stack, the previous signal stack pointer (if any) is over-written and lost.

There is no way to explicitly remove the signal stack once it has been set up. However, the signal stack setting can be changed to the "normal" task stack, with the specification that the task is currently running on this stack. This is equivalent to removing the signal stack.

C Syntax

```
#include <sys\signal.h>

int sigstack(struct sigstack *ss,
             struct sigstack *oss);
```

Parameters

ss A pointer to a `sigstack` structure, which specifies the new signal stack to be used.

This pointer may be specified as 0, in which case the current signal stack pointer and status (if any) is not changed.

oss A pointer to a `sigstack` structure, into which is placed the previous signal stack pointer and status.

This pointer may be specified as 0, in which case the old stack information is not returned.

The `sigstack` structure is defined in SYS\SIGNAL.H as follows:

```
struct sigstack
{
  caddr_t ss_sp;
  int     ss_onstack;
};
```

where:

ss_sp A pointer to the signal handler stack (that is, the value to be used for the ESP register).

ss_onstack
This can be set to 1 to indicate that the task is currently running on the stack specified by the field *ss_sp*, or 0 otherwise.

Values other than 0 and 1 are not valid.

Return Parameters

return code

This is 0 for successful completion or -1 in case of error. In the case of an error, *errno* is set to indicate the error.

EFAULT The address specified by *ss*, *oss*, or *ss*→*ss_sp* is not valid.

**oss* If *oss* is non-zero, the signal stack state current in at the time of the call is returned in this structure.

Return Codes

QSVCgood (0)

Operation completed successfully.

QSVCbadreq (0x8001001E)

ECX is not set to 0 or 1.

QSVCbadaddr (0x80010034)

The address supplied in the EDI register is not valid. For example, it points to memory location that is not allocated or which is inaccessible to the caller.

SVC Handler Generated Faults

none

CPSigVec (SIG_VEC) - Set or Query Signal Handler

Function

This SVC sets up the action or a signal handler or query the current action for a signal. The previous signal handler or action (if any) is returned to the caller, and then, if a new action is being specified, the old action or signal handler is over-written and lost.

This SVC does not allow the caller to specify a signal that is not used in CP/Q, namely signals 0, 5, 7, 10, 23, 25, 26 and 27. See the signal name table in the section “CP/Q Signal Facilities” in the SVC Handler manual for a list of the signal names and numbers. Furthermore, it is not permitted to specify that the SIGKILL, SIGSTOP or SIGCONT signals be ignored, and it is also not permitted to specify a signal handler for the SIGKILL or SIGSTOP signals.

An SCB is allocated for this signal if one is not already allocated. If the specified action is to enter a signal handler, and the caller is a PL3 or user mode task, a signal handler PL0 or kernel mode stack is allocated (if one is not already allocated).

C Syntax

```
#include <sys\signal.h>

int sigvec(int sig,
           struct sigvec *vec,
           struct sigvec *ovec);
```

Parameters

sig	The number of the signal to be set or queried. This may be a number in the range 1-31, other than 5, 7, 10, 23, 25, 26 or 27.
vec	A pointer to a `sigvec` structure that specifies the new signal handler or action for this signal. This pointer can be specified as 0, in which case the current signal handler or action (if any) is not changed.
ovec	A pointer to a `sigvec` structure, into which is copied the old signal handler or action for this signal.

The `sigvec` structure is defined in SYS\SIGNAL.H as:

```
struct sigvec
{
  int (*sv_handler)();
  int sv_mask;
  int sv_flags;
};
```

where:

sv_handler

This can be set to one of the following:

SIG_DFL (= 0)
The default action is to be taken for the specified signal. In this case, the contents of the fields **sv_mask** and **sv_flags** are ignored.

SIG_IGN (= 1)
The specified signal is ignored. In this case, the contents of the fields **sv_mask** and **sv_flags** are ignored.

an address
This specifies the address of the signal handler to be called when this signal is received.

sv_mask The signal mask to be used when executing in the signal handler (that is, a mask of signals to be blocked while the signal handler is running).

This field is significant only if the field *sv_handler* specifies the address of a signal handler.

sv_flags This is a bit significant field, specifying flags. The meaningful bits are defined by the following constants:

SV_ONSTACK = 1
The signal handler is to run on the signal handler stack. If this bit is unset (or there is no signal stack defined), the signal handler runs on the task's current stack.

SV_INTERRUPT = 8
The signal handler is to interrupt a long running SVC (one that specifies it can be interrupted by a signal). If this bit is unset, the interrupted SVC is re-started on return from the signal handler.

This field is significant only if the field *sv_handler* specifies the address of a signal handler.

Return Parameters

return code

This is 0 for successful completion or -1 in case of error. In the case of 0, *errno* is set to indicate the error.

EFAULT The address specified by *vec*, *ovec*, or *vec→sv_handler* is not valid.

EINVAL This can be be for one of the following reasons:

- The signal number is not valid.
- This call specifies that the SIGKILL, SIGSTOP or SIGCONT signal is to be ignored.
- This call specifies a signal handler for the SIGKILL or SIGSTOP signal.

**ovec* The old signal handler or action for this signal (current at the time of the call) is returned to the user in this structure.

Return Codes

QSVCgood (0)

Operation completed successfully.

QSVCbadaddr (0x80010034)

This can be caused by the following:

- The address supplied in the EDI register is not valid. For example, it points to memory location that is not allocated or which is inaccessible to the caller.
- The signal handler address in the parameter list is not valid.

QSVCbadsignal (0x80010033)

This can be caused by the following:

- The signal number is not valid.
- This SVC specifies that the SIGKILL, SIGSTOP or SIGCONT signal is to be ignored.
- This SVC specifies a signal handler for the SIGKILL or SIGSTOP signal.

QSVCnfSCB (0x80010035)

The specified signal does not have an SCB allocated, and there no free SCBs are available.

QSVCnfPL0 (0x8001001D)

The specified signal is to have a signal handler, and the caller is a PL3 task, and there is no space available to create a PL0 stack for the signal handler.

SVC Handler Generated Faults

none

Chapter 8. Task Control Functions

CPChgPriority (CHG_PRTY) - Change Dispatch priority of a Task

Function

This SVC changes the dispatch priority of a task, which can be the caller or some other specified task. If a task other than the caller is specified, either the task issuing this SVC must have the *TCBstoppriv* privilege bit set, or the target task must be either in the same process as the caller or in a process that is a descendent of the caller's process.

Note: Priority level 0 is reserved for the system error process. Therefore, the new priority of the task must be one of the following:

- In the range 1 to 255 (inclusive)
- 1-31 if the SVC handler has been compiled with the option to restrict the number of dispatch priorities to 32
- 1-7 if the SVC handler has been compiled with the option to restrict the number of dispatch priorities to 8

The new priority is also checked to ensure that it is within the bounds specified for the process containing the task.

If the task affected by this SVC is dispatchable, it is added to the end of the dispatch chain of the new priority level unless it is also the current task (that is, the caller). In that case, the task is placed at the front of the dispatch chain. This is necessary to avoid violating the requirement that the current task always be the first task in its dispatch chain.

The restrictions implied by the SVT entry access control bits being set do not apply to this SVC.

C Syntax

```
long int CPChgPriority(unsigned long int Svid,
                       unsigned long int Priority);
```

Parameters

SVid	The SVid of the task concerned. This field can be set to 0, which implies the calling task.
Priority	The new dispatch priority for the task. The restrictions on this value are detailed above.

Return Codes

QSVCgood (0)
Operation completed successfully.

QSVCbadSVid (0x80010002)
Specified SVT entry is not valid, or it is not a task.

QSVCdeadSVid (0x8001000C)
The specified SVT entry is being removed from the system.

QSVCbadprty (0x80010006)
The specified new priority is not valid for the stated task.

SVC Handler Generated Faults

QSVCinvalid (0x80)
The calling task does not have the privilege required to issue this SVC (possible only if *SVid* ≠ 0).

CPCritEnter (CRIT_ENTER) - Enter Critical Code Section

Function

This call informs the system that the calling task is entering a critical code section. This prevents the system from stopping or removing the task until the corresponding CRIT_LEAVE SVC is processed. This SVC is intended to be used by privileged system extensions to prevent the task from being removed while the task is manipulating resources of some form.

This SVC does not perform any form of serialization (it is probably necessary to use a semaphore to enforce serialization), nor does it do anything to prevent the task from being pre-empted by another task (for example, it does not disable interrupts).

Note: This SVC is restricted to supervisor mode code.

The SVC Handler maintains a count of outstanding CRIT_ENTER SVCs for the calling task in order that these SVCs can be nested. However, if this count overflows, a system abend occurs.

Warning: If a fault occurs in a task that is in a critical section or a task that is in a critical section terminates (that is, issues a TASK_HALT SVC), a system abend results.

C Syntax

```
long int CPCritEnter();
```

Parameters

None

Return Codes

QSVCgood (0)
  Operation completed successfully.

SVC Handler Generated Faults

QSVCinvalid (0x80)
  The calling code is not supervisor mode.

CPCritLeave (CRIT_LEAVE) - Leave Critical Code Section

Function

This call informs the system that the calling task is leaving a critical code section. This is the reverse of a CRIT_ENTER SVC, and causes the critical code section count (maintained by the SVC Handler for the task) to be decremented. If the count is decremented beyond zero (that is, after decrementing, it is 255), the system crashes.

If, as a result of this call, the critical code section count becomes 0 and the task is “pending stopped”, the task is stopped. In this case, no return from this SVC occurs unless the task is subsequently restarted by a GO_TASK SVC from some other task.

Note: This SVC is restricted to supervisor mode code.

C Syntax

```
long int CPCritLeave();
```

Parameters

None

Return Codes

QSVCgood (0)
: Operation completed successfully.

SVC Handler Generated Faults

QSVCinvalid (0x80)
: The calling code is not supervisor mode.

CPFaultTask (FALT_TASK) - Fault a Task

Function

This call places a task in the stopped state, as if a fault has occurred in the task. The following occurs:

- A normal fault report message is sent to the fault handler of the task.
- The specified error code is placed in the TCB field *TCBerrcode*.
- The TCB field *TCBstop* is set to *QSVCsoftflt*.
- The task stopped count is incremented.

The task issuing this SVC must have the *TCBfaltpriv* privilege bit set, unless the specified SVid is 0 (that is, fault the calling task), or the target task is either in the same process as the caller or in a process that is a descendent of the caller's process.

This SVC is not permitted if the target task is in the “stopped, not initialized” state.

If the task is already stopped, the TCB fields *TCBerrcode* and *TCBstop* are overwritten with the new values. The task stopped count is still incremented.

The restrictions implied by the SVT entry access control bits being set do not apply to this SVC.

C Syntax

```
long int CPFaultTask(unsigned long int SVid,
                     unsigned long int Errcode);
```

Parameters

SVid The SVid of the task to be faulted. If this register is set to 0, the calling task is faulted.

The error code for the task, to be placed in the TCB field *TCBerrcode*.

Return Codes

QSVCgood (0)
Operation completed successfully.

QSVCbadSVid (0x80010002)
Specified SVT entry is not valid, or it is not a task.

QSVCdeadSVid (0x8001000C)
The specified SVT entry is being removed from the system.

QSVCtaskstop (0x80010013)
The task was already stopped. The FALT_TASK SVC has completed successfully.

QSVCnotinit (0x80010015)
The target task is in the “stopped, not initialized” state.

SVC Handler Generated Faults

QSVCinvalid (0x80)
: The calling task does not have the privilege required to issue this SVC.

CPGoTask (GO_TASK) - Start a Task

Function

This call places a task in the going state, in order that it can be dispatched. The specified task runs only if it is in the no-wait state. This SVC does not affect the wait status. The stop code and error code for the task (the TCB fields *TCBstop* and *TCBerrcode*) are not cleared by this SVC, but they are left holding their previous contents. The contents of these fields are normally not significant unless the task is stopped (that is, the bit *TCBstpbit* is set).

The task issuing this SVC must have the *TCBstoppriv* privilege bit set, or the target task must be either in the same process as the caller or in a process that is a descendent of the caller's process. This SVC can also be issued by the Resource Manager, both as code called from a task and as the Resource Manager task.

This SVC is not allowed for a task that is in stopped state *QSVCuninit* (that is, when the task has been created but the registers have not yet been set up). After creating a task, it is necessary to set up the registers using a WRITE_REGS SVC before the task can be run.

The task stop count (in the TCB field *TCBstopcount*) is decremented. The bit *TCBstpbit* is unset only if the stop count is decremented to zero.

If the GO_TASK is issued while the task in question is pending stopped, rather than stopped, the pending stop operation is cancelled, subject to the same rules as for un-setting the stopped status bits.

Note: The restrictions implied by the SVT entry access control bits being set do not apply to this SVC.

C Syntax

```
long int CPGoTask(unsigned long int SVid);
```

Parameters

SVid	The SVid of the task to be started

Return Codes

QSVCgood (0)
Operation completed successfully.

QSVCbadSVid (0x80010002)
Specified SVT entry is not valid, or it is not a task.

QSVCdeadSVid (0x8001000C)
The specified SVT entry is being removed from the system.

QSVCtaskgo (0x80010014)
The task is not stopped or pending stopped

QSVCnotinit (0x80010015)
The task cannot be started because it has been created, but the registers have not yet been set up.

SVC Handler Generated Faults

QSVCinvalid (0x80)
: The calling task does not have the privilege required to issue this SVC.

CPHaltTask (TASK_HALT) - Stop the Calling Task

Function

This call is used by a task to indicate to the system that it is halting or terminating. An error code field is provided so that a task may halt with a non-zero "return code". An error code of 0 is conventionally usually means task termination with no error.

The following occurs:

- The error code is placed into the field *TCBerrcode* of the calling task's TCB.
- The task stop reason (the TCB field *TCBstop*) is set to *QSVCtaskhalt* = 5 (that is, task halted).
- An inform message is sent to the task's fault handler.
- The calling task is stopped.
- The SVC Handler dispatches another task. This SVC always increments the task stopped count (in the TCB field *TCBstopcount*).

Warning: If the calling task is within a critical code section when this SVC is issued, the CP/Q system abends because this SVC is a variation on a task fault.

This call does not normally return to the calling task because the only possible SVC Handler return code is *QSVCgood*. However, when the task is stopped, as far as the SVC Handler is concerned, the task can be restarted by another task. If the calling task is restarted, the call completes successfully, and the task continues from that point.

C Syntax

```
long int CPHaltTask(unsigned long int ErrCode);
```

Parameters

ErrCode	The error code for the task, to be placed in the TCB field *TCBerrcode*. By convention, the value 0 means "no error".

Return Codes

QSVCgood (0)
Operation completed successfully.

SVC Handler Generated Faults

None

CPPTrace (P_TRACE) - Debugging Facilities, Intel Only

Function

This call provides various facilities to aid program debugging. There are a number of different options, specified by the *Action* parameter, as described below.

Portability Note

The options and/or facilities provided by this SVC are platform specific; the options available, and the nature and format of parameter areas and/or information required or returned will change from one system architecture to the next, as CP/Q is ported to different processors.

C Syntax

```
long int CPPTrace(unsigned long int Action,
                  unsigned long int SVid,
                  void *Addr);
```

Parameters

Action a value that determines the specific action to be performed by this call, as follows.

QSVC_set_parm = 1
set the program parameter address. *Addr* points to a location holding the address of the parameter area for the task specified by *SVid* (0 implies the calling task), to be set into the TCB field *TCBparmaddr*.

Note: This parameter address points to an area within the task's address space that contains:

- the relocation information
- the environment area pointer
- the full file name of the file that contained the load module for this task
- the command line for this program

See the chapter "Technical Reference" of the C Run-time Library Reference Manual for a description of the memory layout of a loaded program, and a detailed description of this area. There are also some details in the SLEEP/R manual.

The task issuing this SVC must have the *TCBstoppriv* privilege bit set, or the target task must be either in the same process as the caller or in a process that is a descendent of the caller's process.

QSVC_get_parm = 2
obtain the program parameter area address for the task specified by *SVid*, 0 implies the calling task. On return, **Addr* will hold the address of the parameter area for the task, as held in the TCB field *TCBparmaddr*.

See the note for the function **QSVC_set_parm** above.

```
typedef struct drblock
{
   unsigned long int dr0;
   unsigned long int dr1;
   unsigned long int dr2;
   unsigned long int dr3;
   unsigned long int dr6;
   unsigned long int dr7;
} DRBLOCK;
```

Figure 8-1. Intel Debug Register Structures

QSVC_set_debug = 3
set the debug registers for the task specified by *SVid*. This task must be marked as being tested by the caller. The task must be stopped.

The specified debug register values apply *only* to the specified task; the SVC handler sets and/or clears, as necessary, the debug registers during task switches.

There is no provision for system wide debug register settings.

The parameter *Addr* points to a *DRBLOCK* structure, as shown in Figure 8-1, containing the new values for the DR*i* registers of the task. The value for DR6 is always ignored (it is present in the buffer only for compatibility with sub-function *Action*=4). If any of the values for DR0, DR1, DR2 and DR3 are zero, then the corresponding bits in DR7 are forced to 0. If DR0, DR1, DR2 and DR3 are all zero, then DR7 will be forced to 0. Further, the "local enable," "local exact" and "GD" bits of DR7 are also forced to 0 (so as to avoid problems when going from the system to SLEEP/Q and back again).

The addresses for DR0, DR1, DR2 and DR3 (if they are non-zero) must be greater than or equal to the start of private class memory. This is to avoid spurious debug register traps in NAP.

QSVC_get_debug = 4
obtain the debug registers of the task specified by *SVid*. The task must be stopped.

The parameter *Addr* points to a *DRBLOCK* structure, as shown in Figure 8-1, into which are placed the current values of the DR*i* registers of the task.

QSVC_set_FPU = 5
set the floating point processor status of the task specified by *SVid*; this may be 0 which signifies the caller. The task (if other than the caller) must be stopped.

The parameter *Addr* points to a *FPUBLOCK* structure, as shown in Figure 8-2 on page 8-12. This is a 108-byte buffer with the following format; this is the same format as is produced by the Intel 387 or 486 instruction FSAVE.

The contents of this buffer should normally be obtained by using a P_TRACE SVC with *Action*=6, or (for the calling task only) by using a FSAVE instruction.

```
typedef struct fpublock
{
   unsigned short fpu_control;                  /*  the status/control area  */
   unsigned short fpu_rsvd1;
   unsigned short fpu_status;
   unsigned short fpu_rsvd2;
   unsigned short fpu_tagword;
   unsigned short fpu_rsvd3;
   unsigned long  fpu_ip;
   unsigned short fpu_cs;
   unsigned short fpu_opcode;
   unsigned long  fpu_dataoffset;
   unsigned short fpu_dataselector;
   unsigned short fpu_rsvd4;
   unsigned char  fpu_stack0[10];   /*  the stack registers - each is 80 bits  */
   unsigned char  fpu_stack1[10];
   unsigned char  fpu_stack2[10];
   unsigned char  fpu_stack3[10];
   unsigned char  fpu_stack4[10];
   unsigned char  fpu_stack5[10];
   unsigned char  fpu_stack6[10];
   unsigned char  fpu_stack7[10];
} FPUBLOCK;
```

Figure 8-2. Intel Floating Point Registers Structure

bytes 0,1 *fpu_control*
the FPU control register

bytes 2,3 **-** reserved

bytes 4,5 *fpu_status*
the FPU status register

bytes 6,7 **-** reserved

bytes 8,9 *fpu_tagword*
the FPU tag register

bytes 10,11 **-** reserved

bytes 12-15 *fpu_ip*
saved FPU instruction pointer (used when examining a floating point error)

bytes 16,17 *fpu_cs*
saved FPU instruction segment selector (used when examining a floating point error)

bytes 18,19 *fpu_opcode*
saved FPU instruction opcode (used when examining a floating point error)

bytes 20-23 *fpu_dataoffset*
saved FPU operand offset (used when examining a floating point error)

bytes 24,25 *fpu_dataselect*
saved FPU operand segment selector (used when examining a floating point error)

bytes 26,27 **-** reserved

bytes 28-37 *fpu_stack0*
stack register 0

bytes 38-47 *fpu_stack1*
stack register 1

bytes 48-57 *fpu_stack2*
stack register 2

bytes 58-67 *fpu_stack3*
stack register 3

bytes 68-77 *fpu_stack4*
stack register 4

bytes 78-87 *fpu_stack5*
stack register 5

bytes 88-97 *fpu_stack6*
stack register 6

bytes 98-107 *fpu_stack7*
stack register 7

If the target task has no FPU save area allocated, one will be assigned by this call.

QSVC_get_FPU_alloc = 6
obtain the floating point processor status of the task specified by *SVid*; this may be 0 which signifies the caller. The task (if other than the caller) must be stopped.

The parameter *Addr* points to a *FPUBLOCK* structure, as shown in Figure 8-2 on page 8-12, the same as for *Action*=5.

If the target task has no FPU save area allocated, one is assigned by this SVC; in this case, the returned FPU status is as set by the FINIT instruction.

QSVC_get_FPU = 7
obtain the floating point processor status for a task. This is the same as *Action*=6 *except* if the target task has no current FPU context (i.e. no FPU save area allocated), then this call is given return code *QSVCnoFPUregs*, and no valid FPU registers are returned.

An FPU save area is *not* allocated by this form of the P_TRACE SVC.

QSVC_test_task = 8
test a task; *SVid* specifies the task that is to be controlled. The caller is made the fault handler of the task (the caller's fault handler count is incremented), and the task is marked as being debugged by the caller. In particular, this prevents both the caller and the task being debugged from being removed from the system. Further, the lock count is incremented, and the “locked” bit is set, in the PCB for the process containing the target task. This prevents “process delete” operations on that process.

Further, the task to be debugged is stopped. If the task was running, the task “stopped type” is set to 1 and the “stopper” SVid is set to that of the task issuing the P_TRACE SVC. If the task was already stopped, then the stop count is incremented. Note, however, that if the task was in the "stopped, not-initialized" state, the stop count is *not* incremented, since when in this state a GO_TASK SVC cannot be issued to set it back to 1.

The task issuing this SVC must have the *TCBstoppriv* privilege bit set, or the target task must be either in the same process as the caller or in a process that is a descendent of the caller's process.

The parameter *Addr* is not used for this call.

QSVC_end_test = 9
end testing a task; *SVid* specifies the task that is to be released. This task must be marked as being tested by the caller. The dispatch state of the task is not changed by this SVC; in particular, the task is *not* started if it is stopped. The lock count is decremented (and the lock bit unset if the count becomes 0) in the PCB for the process containing the target task.

This form of this call is allowed when the task is marked as “being removed from the system.”

The parameter *Addr* is not used for this call.

```
typedef struct regblock
{
   unsigned long int  esp0;
   unsigned long int  cr3;
   unsigned long int  DebugTrap;
   unsigned long int  edi;
   unsigned long int  esi;
   unsigned long int  ebp;
   unsigned long int  edx;
   unsigned long int  ecx;
   unsigned long int  ebx;
   unsigned long int  eax;
   unsigned long int  Eflags;
   unsigned long int  eip;
   unsigned long int  cs;
   unsigned long int  esp;
} REGBLOCK;
```

Figure 8-3. Intel Registers Structure

QSVC_read_regs = 10
read the registers of a task. The task concerned must be stopped.

The parameter *Addr* points to a *REGBLOCK* structure, as shown in Figure 8-3.

The restrictions implied by the SVT entry access control bits do not apply to this call.

The task issuing this call must have the *TCBstoppriv* privilege bit set, or the target task must either be in the caller's process or in a process that is a descendent of the caller's process.

QSVC_write_regs = 11
write the registers of a task. The task concerned must be stopped. This call is to set up the registers of the task as seen by that task after exit from the SVC Handler, not the registers for the task as used when running within the SVC Handler.

The parameter *Addr* points to a *REGBLOCK* structure, as shown in Figure 8-3. The fields in the parameter list for CS, ESP0 and CR3 are always ignored. Further, for a PL0 task that is not being initialized, the ESP value is also ignored (because for such a task, ESP will currently point to the base of an SVC Handler stack frame - changing ESP would imply moving the SVC Handler stack frame to some other location).

If the task is marked as “stopped, not initialized”, that is the task has been created but the registers have not yet been set up, then this SVC performs the following:

- CS and SS are set appropriately according as to whether task is a PL0 or PL3 task (the TCB field *TCBprivbits* has the bit *TCBPL0task* set for a PL0 task, unset for PL3).
- ESP0 is set to match the allocated PL0 stack for a PL3 task, or to ESP if it is a PL0 task.
- CR3 is set to the value for the process containing the task.
- an SVC Handler stack frame is set up within the PL0 stack (which is the task supplied stack for a PL0 task), as if the task had been suspended within the SVC Handler.
- the saved task registers within this stack frame are set to the values supplied in this SVC, apart from the flags register, which is set to have the normal system I/O Privilege level (IOPL) and interrupts enabled.
- the saved registers within the TCB (i.e. ESP and EBP) are set to point to this SVC Handler stack frame.
- the task is marked as “stopped, not started,” so that a subsequent GO_TASK SVC can start execution of the task. Execution will actually start within the dispatcher task change code within the SVC Handler, but the generated SVC Handler stack frame is set up such that control immediately returns to the task at the EIP value specified in this WRITE_REGS SVC.

If the target task entered the SVC Handler by means of a near call SVC (from PL0), and the supplied register values have the “single step” bit set in Eflags, then the stack frame is forced to look like a INT entry, so that the single step can be implemented (so it single steps the task code, rather than the SVC Handler exit code!).

The restrictions implied by the SVT entry access control bits do not apply to this SVC. The task issuing this SVC must have the *TCBstoppriv* privilege bit set, or the target task must be either in the same process as the caller or in a process that is a descendent of the caller's process.

This SVC performs the following checks on the supplied register values:

- the “nested task”, VM and “alignment check” bits of the Eflags register are forced to 0.
- if the TCB field *TCBprivbits* has the bit *TCBPL0task* unset then interrupts are forced to enabled.
- for the first WRITE_REGS for a task (i.e. when "initializing" the task), the CR3 value (from the PCB) is checked to ensure that it is non-zero.
- the pages referenced by EIP and ESP must exist. Further, the page referenced by ESP must be writeable.

- for a supervisor mode task, the page referenced by EBP (held in the TCB, or ESP in the parameter list for task initialization) must exist and be writeable.
- for a user mode task, (the TCB field *TCBprivbits* has the bit *TCBPL0task* unset) then EIP and ESP must point to user mode pages.

Note: See the function CPTaskRegs for an architecture independent method of initializing the registers for a task.

QSVC_qry_state = 14
query the dispatch state of the task *SVid* (0 implies the calling task). This facility was implemented originally as an aid to testing the SVC Handler, but may be of use to certain special systems tasks, such as debugging or monitoring tasks.

The parameter *Addr* points to a *QTBLOCK* structure, as shown in Figure 8-4 on page 8-17. On return from this call, the fields are set as follows:

byte 0 - *wait*
the task wait status (i.e. the types of events awaited by this task). This is a copy of the TCB field *TCBwait*.

byte 1 - *status*
the task dispatch status. This is a copy of the TCB field *TCBstatus*.

byte 2 - *stop*
the task stop reason. This is a copy of the TCB field *TCBstop*.

byte 3 - *priv*
the task's privilege bits. This is a copy of the TCB field *TCBprivbits*.

byte 4 - *priority*
the task's dispatch priority. This is a copy of the TCB field *TCBpriority*.

byte 5 - *critcnt*
the task's critical section count. This is a copy of the TCB field *TCBcritcnt*.

byte 6 - *prempt*
the task's pre-emption status. This is a copy of the TCB field *TCBpreempt*.

byte 7 - *muxcnt*
the task's MUX_WAIT count. This is a copy of the TCB field *TCBmuxcnt*.

bytes 8-11 - *stopcnt*
the task's stop count. This is a copy of the TCB field *TCstopcount*.

bytes 12-15 - *procprio*
the base dispatch priority of the process containing the task. This is a copy of the PCB field *pcb_priority*.

```
typedef struct qtblock
{
   unsigned char     wait;        /*  Wait event types             */
   unsigned char     status;      /*  Dispatch status              */
   unsigned char     stop;        /*  Stop reason                  */
   unsigned char     priv;        /*  Privilege bits               */
   unsigned char     priority;    /*  Task priority                */
   unsigned char     critcnt;     /*  Critical code nesting count  */
   unsigned char     prempt;      /*  pre-emption flag state       */
   unsigned char     muxcnt;      /*  MUX_WAIT count               */
   unsigned long int stopcnt;     /*  Task stopped count           */
   unsigned long int procpri;     /*  Process base priority        */
   unsigned long int faltcnt;     /*  fault handler count          */
   unsigned long int falter;      /*  fault handler TCB pointer    */
   unsigned long int errcode;     /*  Stop or fault error code     */
   unsigned long int errinfo;     /*  Not used for Intel           */
} QTBLOCK;
```

Figure 8-4. Intel Query Task State Structure

bytes 16-19 - *faltcnt*
: the count of tasks for which this task is the fault handler. This is a copy of the TCB field *TCBfaltcnt*.

bytes 20-23 - *falter*
: the (TCB pointer of) the fault handler of this task. This is a copy of the TCB field *TCBfalter*.

bytes 24-27 - *errcode*
: the task's error or return code. This is a copy of the TCB field *TCBerrcode*.

bytes 28-31 - *errinfo*
: supplementary error information for the task. Currently this is unused for Intel systems.

A query task call returns an instantaneous snapshot of the task dispatch status, but it must be remembered that the dispatch status of a task can change very frequently. The values returned for the TCB fields *TCBstop* and *TCBerrcode* will be the values set in those locations when the task was last stopped - non-zero values returned for these locations do *not* imply that the task is currently stopped. A task is stopped if and only if the bit *TCBstpbit* is set in the byte *TCBstatus*.

The restrictions implied by the SVT entry access control bits do not apply to this call.

```
typedef struct cpublock
{
   unsigned long int CPU_high;     /*  m.s. 32 bits of value         */
   unsigned long int CPU_low;      /*  l.s. 16 or 32 bits of value   */
} CPUBLOCK;
```

Figure 8-5. Query Task CPU Usage Structure

QSVC_get_CPU = 15
: get the CPU usage of a task. The parameter *SVid* must specify a task, or be set to 0, which implies the calling task, or may be set to 0xFFFFFFFF, in which case the system idle time is returned.

If CPU time measurement facilities are not available (e.g. the support for it is omitted from the SVC Handler), this call will return the value 0.

The parameter *Addr* points to a *CPUBLOCK* structure, as shown in Figure 8-5 on page 8-17. The format of the returned value is processor specific. On an Intel system, the field *CPU_high* holds the most significant 32 bits of the value, and indicates complete seconds of time; the field *CPU_low* holds the least significant 16 bits of the time, and is a fraction of a second in the range 0-65535 (65536 would be a complete second).

SVid the SVid of the target task for the call, but see under the specifications of each value for *Action* for more information.

Addr the nature of this parameter depends on the action to be performed; see under the specifications of each value for *Action* for more information.

This parameter is not used for *Action*=8 and 9.

Return Parameters

*****_Addr_ this is overwritten for calls with *Action*=2, as described above.

The buffer specified by this parameter is filled with data for calls with *Action*=4, 6, 7 and 15, as described above.

Return Codes

QSVCgood (0)
operation completed successfully.

QSVCbadreq (0x8001001E)
Action is not a valid code as listed above.

This return code may also occur if one of the values for DR0, DR1, DR2 and/or DR3 is not zero and is less than the start of the “private” memory address range (*Action*=3 only).

QSVCbadSVid (0x80010002)
the specified SVT entry is not a task.

QSVCdeadSVid (0x8001000C)
the specified SVT entry is being removed from the system (not *Action*=9).

QSVCdebug2 (0x8001002F)
the specified task is already being debugged (*Action*=8).

QSVCnodebug (0x80010030)
the specified task cannot be debugged (*Action*=8), or is not being debugged by the caller (*Action*=3 or *Action*=9).

QSVCunpriv (0x80010031)
the caller has insufficient privilege to perform this operation (*Action*=1 or *Action*=8).

QSVCtaskstop (0x80010013)
(*Action*=8 only) - the operation was completed successfully, but the task was already stopped. The task stopped count has been incremented; this implies that the task “stopped count” is now > 1.

QSVCtaskgo (0x80010014)
: (*Action*=3, 4, 5 or 6) - the task is not stopped.

QSVCnoFPU (0x80010020)
: (*Action*=5, 6 or 7) - there is no FPU available (this includes no FPU save area in the system and/or no FPU support code in the SVC Handler).

QSVCnoFPUregs (0x80010023)
: (*Action*=7) - the task has no FPU context to return, and has no FPU save area allocated.

QSVCnf387area (0x80010017)
: (*Action*=5, 6) - there is no free FPU save area space to allocate an FPU save area for the target task.

QSVCwrongproc (0x80010039)
: the processor type is incorrect (*Action*=5, 6, 7, 10 or 11).

SVC Handler Generated Faults

QSVCparlist (0x81)
: parameter list is invalid, for example the parameter *Addr* does not point into an allocated page.

CPSetPreempt (SET_PRE) - Set Task Preemption Status

Function

This call sets or clears the “task preemptable” flag in the caller's TCB. The standard SVC Handler takes no other action for this SVC and also makes no use of this flag. See the section “Task Dispatch Organization”in the SVC Handler manual for more information about this subject.

The time slicer that is provided as an option with the Intel version of CP/Q takes note of the preemption flag and alters its behavior accordingly. See the section “Time Slicing”in the SVC Handler manual for more information.

The task issuing this SVC must have the *TCBsyspriv* privilege bit set or be a supervisor mode task.

C Syntax

```
long int CPSetPreempt(unsigned long int Preempt);
```

Parameters

Preempt	The value placed in the task preempt status flag. This can accept the values *QSVCpreempt* or *QSVCpreprio*.

Return Codes

QSVCgood (0)
: Operation completed successfully.

QSVCbadprmpt (0x80010007)
: Requested preemption status is not valid (that is, not *QSVCpreempt* or *QSVCpreprio*).

SVC Handler Generated Faults

QSVCinvalid (0x80)
: The calling task does not have the privilege required to issue this SVC.

CPStopTask (STOP_TASK) - Stop a Task

Function

This call places a task in the stopped state, in order that it is not dispatched. This is achieved by setting the bit *TCBstpbit* in the TCB field *TCBstatus*.

The task issuing this SVC must have the *TCBstoppriv* privilege bit set, or the target task must be either in the same process as the caller or in a process that is a descendent of the caller's process. This SVC can also be issued by the Resource Manager, both as code called from a task and as the Resource Manager task.

This SVC is not permitted if the target task is in the “stopped, not initialized” state.

If the specified task is currently within a critical code section, the task is placed in the “pending stopped” state. This means that the task is not stopped immediately, but instead will be stopped when it exits from the critical code section. In this case, the STOP_TASK SVC receives a *QSVCincrit* error return code. A message with the bytes 12-15 of the data area set to *QSVCinfstop* is sent to the caller when critical code count of the target task becomes zero, and the task is finally stopped.

If the task is already “pending” stopped, this SVC gives return code *QSVCpendstop*, the task stopped count is not incremented, and the caller is not notified when the task is placed into the stopped state. Otherwise, this SVC always increments the task stopped count in the TCB field *TCBstopcount*, except when the task is not previously marked as “stopped” or “pending stopped”, and this SVC marks the task as “pending stopped”. In this case, the stop count is incremented when the task changes from “pending stopped” to “stopped”. This implies that this SVC increases the task stopped count even when an error return code indicates that the task was already stopped.

The TCB field *TCBstop* is set to *QSVCstopped*, and the field *TCBerrcode* is set to the error code passed in the ECX register with the request, unless the task is already stopped or pending stopped.

The restrictions implied by the SVT entry access control bits being set do not apply to this SVC.

C Syntax

```
long int CPStopTask(unsigned long int SVid,
                    unsigned long int ErrCode);
```

Parameters

SVid	The SVid of the task to be stopped
ErrCode	The error code for the task to be placed in the TCB field *TCBerrcode*.

Return Codes

QSVCgood (0)
Operation completed successfully.

QSVCbadSVid (0x80010002)
Specified SVT entry is not valid, or it is not a task.

QSVCdeadSVid (0x8001000C)
The specified SVT entry is being removed from the system.

QSVCtaskstop (0x80010013)
The task is already stopped. However, the task stopped count has still been incremented.

QSVCpendstop (0x80010037)
The task is already “pending” stopped. The task stopped count has not been incremented. The caller is not notified when the task is placed into the stopped state.

QSVCincrit (0x8001001A)
The task has been placed in the “pending stopped” (or “pending process stopped”) state, rather than stopped, because it has a non-zero critical code section count.

QSVCnotinit (0x80010015)
The target task is in the “stopped, not initialized” state.

SVC Handler Generated Faults

QSVCinvalid (0x80)
The calling task does not have the privilege required to issue this SVC.

CPTaskRegs - Initialize Task Registers

Function

This call is intended to provide an architecture-independent method of setting the registers for a task, and is available as an alternative to the *CPWriteRegs* SVC. The task concerned must be stopped. This call sets up the registers of the task as seen by that task after exit from the SVC Handler, not the registers for the task as used when it is dispatched within the SVC Handler.

The restrictions implied by the SVT entry access control bits do not apply to this call. The task issuing this call must have the *TCBstoppriv* privilege bit set, or the target task must either be in the caller's process or in a descendent process.

This call performs the following checks on the supplied register values:

- If the target task is user level, the code and stack must be in user mode pages.

Figure 8-6 on page 8-24 illustrates a C program that makes use of the *CPTaskRegs* call. This program compiles and runs successfully on all architectures for which CP/Q is currently implemented.

C Syntax

```
long int CPTaskRegs(unsigned long int SVid,
                    unsigned long int Entrypoint,
                    unsigned long int StackPointer,
                    unsigned long int rsvd);
```

Parameters

SVid The SVid of the task whose registers are to be set.

EntryPoint
The address of the (code) entrypoint for the task, that is the address of the code or function that is to be executed by the task (see the coding example below).

StackPointer
The address to be used as the stack pointer for the task. This is the byte immediately above the top of an empty stack (since the stack grows downward).

Note: The stack pointer should be double-word aligned, so it is good practice to make the stack an integral number of double-words in length.

rsvd Currently unused (0 is a good value here).

Return Codes

QSVCgood (0)
Operation completed successfully.

QSVCbadSVid (0x80010002)
Specified SVT entry is not valid, or it is not a task.

QSVCdeadSVid (0x8001000C)
The specified SVT entry is being removed from the system.

QSVCtaskgo (0x80010014)
The task is not stopped.

QSVCbadregs (0x80010018)
The supplied registers are in some way unacceptable (see the above list of checks on the register values).

QSVCbadstack (0x80010022)
The page or pages containing the stack of a supervisor mode task do not exist or are in some way inaccessible.

SVC Handler Generated Faults

QSVCinvalid (0x80)
The calling task does not have the privilege required to issue this SVC.

QSVCparlist (0x81)
Parameter list is not valid (for example, it does not point into an allocated page).

```
/**********************************************************************
*                                                                     *
*  Sample program to demonstrate the CPTaskRegs call.  We create a    *
*  second task, write its registers using CPTaskRegs, and then start  *
*  the task.                                                          *
*                                                                     *
*  This program will compile and run under both INTEL and PowerPC     *
*  architectures.                                                     *
*                                                                     *
**********************************************************************/

#include <stdio.h>                      /*  C library include files  */
#include <string.h>
#include <stdlib.h>
#include <stduser.h>
#include <cpqlib.h>                     /*  CP/Q system declarations  */


/*  macros  */

#define MSGSIZ 1024                         /*  1024 32-bit entries  */
#define STKSIZ (8*(1024))                              /*  8K stack  */


/*  global variables  */

MSG               msg;                            /*  message buffer  */

/**********************************************************************
*                                                                     *
*  This function will become a separate task.  The CPTaskRegs call    *
*  will set up the registers for this separate task.                  *
*  All this task does is wait for a message, then terminate.          *
*                                                                     *
**********************************************************************/

void function(void)
{
  unsigned long int  msgcount;                     /*  message count  */
  int                retval;                        /*  return value  */
```

Figure 8-6 (Part 1 of 3). CPTaskRegs Code Example

```
  /*  setup message header.  */

  msg.h.HdrVer = 0;                                /*  header version  */
  msg.h.msgq_id = 0;                               /*  caller's msg q  */
  msg.h.dw_count = MSGSIZ;                    /*  message buffer size  */

  /*  wait for a message from anybody  */

  retval = CPRecvMsg(&msg, &msgcount, 0, SVCWAITFOREVER);

  /*  After we return from CPRecvMsg, we terminate voluntarily and
      our return value is the return value from CPRecvMsg.

      This task shall not return from CPHaltTask.                  */

  CPHaltTask(retval);                                   /*  terminate  */
}

/*********************************************************************
*                                                                    *
*  main function.  We create a new task, allocate memory from the    *
*  Memory Manager to be used as a stack for the created task, and    *
*  then call CPTaskRegs to set up the registers for the task.        *
*                                                                    *
*********************************************************************/

int main(void)
{
  int               retval;                          /*  return value  */
  unsigned char     *stackmem;                       /*  stack memory  */
  unsigned char     *stackptr;       /*  stack ptr for created task  */
  UINT32            svid;                   /*  svid of create task  */

/*  Create a task by calling the Resource Manager  */

  retval = CPCreateTask("FTASK",                        /*  task name  */
                            0,                    /*  use defaults  */
                            0,                    /*  use defaults  */
                            0,                    /*  use defaults  */
                            0,                    /*  use defaults  */
                            0,                    /*  use defaults  */
                            0,                    /*  use defaults  */
                        &svid);             /*  returned task svid  */

  if (retval != QRMgood)                      /*  check return code  */
  {
    printf("CPCreateTask returned 0x%08x\n",retval);
    paws(retval);
    exit(retval);
  }

  /*  next, allocate memory to be used as the stack for the created
      task.  We call the Memory Manager to obtain memory.  Note that
      STKSIZ should double-word aligned.                              */

  retval = CPAllocMem(QMprivate|QMnew_current|QMuser|QMwrite,
                      STKSIZ,
                      &stackmem);
```

Figure 8-6 (Part 2 of 3). CPTaskRegs Code Example

```
    if (retval != QMsuccess)
    {

      /*  the allocate failed - clean up  */

      printf("CPAllocMem returned 0x%08x\n",retval);
      paws(retval);
      CPDelete(svid, 0);                          /*  remove the task  */
      exit(retval);
    }

    /*  Initialize stack frame memory to a value we can recognize when
        using a debugger.  This makes the stack more readily
        identifiable when examining the program's memory                */

    memset(stackmem,0xFF,STKSIZ);

    /*  the stack pointer is set up for an empty stack.  Should be
        double-word aligned.                                        */

    stackptr = stackmem + STKSIZ;

    /*  Set up the registers of the created task  */

    retval = CPTaskRegs((unsigned long int)svid,
                        (unsigned long int)&function,
                        (unsigned long int)stackptr,
                        0);

    if (retval != QSVCgood)
    {

      /*  writing the registers failed - clean up  */

      printf("CPTaskRegs returned 0x%08x\n",retval);
      CPFreeObj(QMold_current,stackmem);              /*  free memory  */
      CPDelete(svid, 0);                          /*  remove the task  */
      paws(retval);
      exit(retval);
    }

    /*  start the task we have created  */

    retval = CPGoTask(svid);                           /*  start task  */

    if (retval != QSVCgood)                     /*  check return code  */
    {
      printf("CPGoTask returned 0x%08x\n",retval);
      paws(retval);
      CPFreeObj(QMold_current,stackmem);              /*  free memory  */
      CPDelete(svid, 0);                          /*  remove the task  */
      exit(retval);
    }

    /*  tell the world we have succeeded  */

    printf("success!\n");

    return(EXIT_SUCCESS);
  }
```

Figure 8-6 (Part 3 of 3). CPTaskRegs Code Example

Chapter 9. Creation or Deletion of System Entities

CPSysCreateSLIH (CRT_SLIH) - Install a Second Level Interrupt Handler

Function

This call installs a second level interrupt handler for a hardware interrupt level. This SVC can be issued only by a suitably privileged task. On a PowerPC system, the caller must be a supervisor mode task. On an Intel system, the code making this request must have I/O privilege, that is the calling code CPL must be ≤ the system IOPL.

The specified system entity is queued to be notified of interrupts from this interrupt level. When an interrupt is received, each SLIH for this interrupt level is notified by sending an IH_ATTN message to a task or tasks or message queues, by the clearing of a synchronization semaphore, or by restarting tasks that are waiting within an INT_WAIT SVC. If other SLIHs are set up for this interrupt level, this SVC creates an additional SLIH for this level. All the SLIHs for this interrupt level are notified whenever an interrupt occurs. Creating a SLIH is independent of the presence or absence of a user exit call from the FLIH. If one or more SLIHs are created for a hardware interrupt for which there is a user-exit routine called from the FLIH, the SLIHs are notified after the user exit routine has been called. Only one of the exit routine or these SLIHs should clear the interrupt and issue the End-of-Interrupt (EOI) command to the interrupt controllers for each occurrence of the interrupt, or unpredictable results can occur.

When installing the SLIH, if no previous SLIHs exist for that level, the SVC Handler un-masks the interrupt level as follows: (if one or more SLIHs already exist for this level, the interrupt is already unmasked, and may be active - in this case, the mask must not be altered).

- The SVC Handler clears the corresponding bit in the IMR and the bit for the cascade if the interrupt level is on the second 8259.

Certain interrupt levels may not be available, and cannot be specified, as follows:

- Interrupt level 2 (the cascade on the first interrupt controller for the second interrupt controller) is always unavailable.
- Either level 8 (the real-time clock) or level 0 (the 8254 timer), depending on the compile time options for the SVC Handler and the system build options used, will be used for the system clock and is hence is unavailable.
- If the real-time clock is used for the system timer and the CPU time measurement facility is enabled, both levels 0 and 8 are unavailable.
- If the processor is a 386 and floating point support is enabled, interrupt level 13 is unavailable.

  This does not apply to a 486 or Pentium processor because internal error reporting (hardware fault 16) is used for floating point errors.

C Syntax

```
long int CPSysCreateSLIH(unsigned long int SVid,
                         unsigned long int IntLevel,
                         unsigned long int *SLIBid);
```

Parameters

SVid	The SVid of the task, message queue, or semaphore by which the interrupts are to be notified. If this specifies a message queue or task, notification is by IH_ATTN message. If this is a synchronization semaphore, this semaphore is cleared each time an interrupt occurs. This parameter can be 0, which implies the notification is by means of a message to the caller. Alternatively, this parameter can be set to 0xFFFFFFFF, which implies the calling task is the SLIH, and it waitsfor interrupt notification using INT_WAIT SVCs. If the user exit **UCslibscan** is defined (see the section “User Code Exits” in the appendices to the SVC Handler manual), then the value in this parameter can also be 0xFFFF*xxxx*, where *xxxx* are any 4 hex digits the caller assigns (except FFFF, because 0xFFFFFFFF implies an INT_WAIT type SLIH). This value has the effect that the normal SVC Handler SLIH notification does not occur, but calls the user exit routine instead.
Intlevel	The interrupt level this SLIH is to serve in the range 0-*n*, where *n* is the minimum of the number of ICBs in the system (specified at system build time) and the number of hardware interrupt handlers (FLIHs) in the SVC Handler (a compile time option). The default value for *n* on both a PS/2 and on an RS/6000 is 15. On a PS/2, interrupt levels 2 and 8 (or level 0, if the system build option NOCMOS was used) might not be used.
SLIBid	A pointer to a 32-bit word into which is placed the ID of the new SLIH.

Return Parameters

**SLIBid*	If the return code is *QSVCgood*, this is set to hold the ID of the second level interrupt block (SLIB) for this task. This is actually the offset of the SLIB within the SDA segment.

Return Codes

QSVCgood (0)
Operation completed successfully.

QSVCSLIH2 (0x8001002E)
The specified SVid is already being notified of interrupts on this level.

QSVCnfSLIB (0x8001001F)
No free second level interrupt blocks (SLIBs).

QSVCbadSVid (0x80010002)
The SVid in EDX does not specify a task, message queue, or synchronization semaphore.

QSVCbadint (0x80010019)
The Specified interrupt level is not valid.

SVC Handler Generated Faults

QSVCinvalid (0x80)
: The calling code does not have the privilege necessary to issue this SVC.

QSVCinvSVid (0x82)
: The caller is not permitted to access the specified SVT entry.

CPSysDeleteSLIH (DEL_SLIH) - Delete a Second Level Interrupt Handler

Function

This call removes a Second Level Interrupt Handler (SLIH) set up by the calling task. This SVC can be issued only by a suitably privileged task; on a PowerPC system, the caller must be a supervisor mode task. On an Intel system, the code making this request must have I/O privilege, that is the calling code CPL must be ≤ the system IOPL.

A SLIH may be deleted only by the task that set it up. This SVC removes references to this SLIH from the chains of second level interrupt blocks (SLIBs) for the appropriate interrupt level. Removing a SLIH has no effect on the presence or absence of a user exit routine in the SVC handler FLIH. If a user exit routine is present, even if the last SLIH is removed, the user exit routine continues to be called on subsequent interrupts.

If the deleted SLIH is the last for that interrupt level, the SVC Handler takes the following action:

- If that interrupt is currently “in service,” that is, the corresponding bit is set in the In Service Register in the 8259 interrupt controller, the appropriate specific EOI command is issued.
- The interrupt level is masked off in the IMR.

If a task is deleted from the system, all SLIH's created by that task are also removed (that is, the appropriate implicit DEL_SLIH SVCs are executed).

C Syntax

```
long int CPSysDeleteSLIH(unsigned long int SLIBid);
```

Parameters

SLIBid	Holds the ID of the SLIB for this SLIH. This ID is the offset of the SLIB within the SDA segment and was returned by the SVC Handler in the reply to the CRT_SLIH SVC that set up the interrupt handler.

Return Codes

QSVCgood (0)
Operation completed successfully.

QSVCbadid (0x8001002D)
The specified SLIB id is not the offset of a SLIB, or this SLIH was not set up by the calling task.

SVC Handler Generated Faults

QSVCinvalid (0x80)
The calling code does not have the privilege necessary to issue this SVC.

Chapter 10. Timer Services

CPBeep (BEEP_IT) - Sound the "Beeper"

Function

This call sounds the system speaker.

Note: A (SLEEP) system build option is available to disable the beeper code, and an SVC Handler compile time option to is available to omit the beeper code. If either of these options is exercised, BEEP_IT SVCs always receive the return code *QSVCnotavail*.

C Syntax

```
long int CPBeep(unsigned long int Period,
                unsigned long int Frequency);
```

Parameters

Period The duration of the beep in micro-seconds in the range 1,000-8,000,000. A value of 150,000 sounds a short beep; 1,000,000 sounds a 1 second beep.

Frequency
The frequency in Hertz in the range 20-20,000.

Return Codes

QSVCgood (0)
Operation completed successfully.

QSVCbadreq (0x8001001E)
The requested time is <1,000 or >8,000,000, or the specified frequency is <20 or >20,000.

QSVCbadCMOS (0x80010008)
The CMOS real-time clock does not appear to be operating.

This return code is possible only on an Intel system, and then only if the system was built such that the CMOS clock is turned on, and the CMOS support code has been included in the SVC Handler.

QSVCnotavail (0x80010021)
This facility is not available (bit 5 is set in the NDA field *SVCbits*, or the beeper support code has been omitted from the SVC Handler by a compile time option).

SVC Handler Generated Faults

None

CPGetTime (GET_TIME) - Get the Time and Date

Function

This call obtains the time and date information held within the CDA.

The CDA is readable by user tasks, and a program can read the time and/or date fields directly. This is acceptable if only one value is being obtained. However, if the program attempts to read the time and date together, and presumably expects to get a consistent set of values, it is possible for the system to receive a timer interrupt while the task is reading the data. This can result in inconsistent or garbled data. A supervisor mode task could protect against this by disabling I/O interrupts; however, a user mode task cannot do this. Thus, any code that can be used within a user mode task should use this SVC to obtain a copy of the timer fields from the CDA.

C Syntax

```
long int CPGetTime(void *DataBuf);
```

Parameters

```
typedef struct time_block
{
  unsigned long int  time_ticksperday;     /*  timer ticks per day            */
  unsigned long int  time_totalticks;      /*  time in ticks since midnight   */
  unsigned char      time_thehundredths;   /*  hundredths of second           */
  unsigned char      time_thesecond;       /*  seconds part of the time       */
  unsigned char      time_theminute;       /*  minutes part of the time       */
  unsigned char      time_thehour;         /*  hours part of the time         */
  unsigned short     time_theyear;         /*  year part of the date          */
  unsigned char      time_themonth;        /*  month part of the date         */
  unsigned char      time_theday;          /*  day part of the date           */
  unsigned short     time_thetick;         /*  ticks within current second    */
  unsigned short     time_tickhertz;       /*  clock frequency in Hertz       */
  unsigned short     time_daynumber;       /*  day number within the year     */
  unsigned char      time_theweekday;      /*  day of the week (Sunday = 0)   */
  unsigned char      time_tickrate;        /*  number defining clock tick rate  */
} TIME_BLOCK;
```

Figure 10-1. Get Time and Date Structures

DataBuf A pointer to a writeable buffer to receive the time data, that is, a *TIME_BLOCK* structure, as shown in Figure 10-1.

Return Parameters

**DataBuf* If the return code is *QSVCgood*, this buffer was overwritten with the contents of the CDA time and date fields.

Return Codes

QSVCgood (0)
Operation completed successfully.

SVC Handler Generated Faults

QSVCparlist (0x81)
: The parameter list or its address is not valid. For example, *DataBuf* does not point to an allocated and writeable page.

A general protection fault, stack fault, or page fault can also occur for this SVC, if the specified buffer is not accessible, or if all or part of the buffer cannot have physical pages allocated to it.

CPSetDate (SET_DATE) - Set Current Date

Function

This call sets the current date, and the day of the week is calculated from the specified date.

On an Intel system, if the hardware contains a CMOS Real Time Clock, and the CMOS support code is included in the SVC Handler, the date is also set in the CMOS real time clock.

On an Intel system, there is a compile-time option within the SVC Handler for this SVC to be restricted to supervisor mode callers.

C Syntax

```
long int CPSetDate(unsigned long int Date,
                   unsigned long int Month,
                   unsigned long int Year);
```

Parameters

Date	The day of the month (in the range 1-31).
Month	The month (in the range 1-12, where January is 1 and December is 12)
Year	The year (in the range 1980-2099)

Return Codes

QSVCgood (0)
Operation completed successfully.

QSVCbadreq (0x8001001E)
One or more of the new date fields is not valid.

QSVCbadCMOS (0x80010008)
The CMOS real-time clock does not seem to be operating.

This return code is possible only on an Intel system, and then only if the system was built such that the CMOS clock is turned on, and the CMOS support code has been included in the SVC Handler.

SVC Handler Generated Faults

QSVCinvalid (x80)
if the SVC Handler was compiled with the option to restrict this SVC to supervisor mode callers, then a user mode caller is faulted with this error code.

CPSetTime (SET_TIME) - Set Current Time

Function

This call sets the current time. The hours, minutes, and seconds are set to the specified values, and the current fraction of a second is set to 0. This prevents inconsistencies between the time as the number of ticks since midnight and the time expressed as hours, minutes, seconds, and ticks.

On an Intel system, if the hardware contains a CMOS Real Time Clock, and the CMOS support code is included in the SVC Handler, the time is also set in the CMOS real time clock.

On an Intel system, there is a compile-time option within the SVC Handler for this SVC to be restricted to supervisor mode callers.

Note: Be careful when using this SVC to change the time when the current time is near midnight. Thus, if at 23:55 this SVC is used to set the time to 00:05, then this is equivalent to putting the clock back by 23 hours 50 minutes. This is not treated as putting the clock forward 10 minutes.

C Syntax

```
long int CPSetTime(unsigned long int Hour,
                   unsigned long int Minute,
                   unsigned long int Second);
```

Parameters

Hour	The hour (in the range 0-23)
Minute	The minutes (in the range 0-59)
Second	The seconds (in the range 0-59)

Return Codes

QSVCgood (0)
: Operation completed successfully.

QSVCbadreq (0x8001001E)
: One or more of the new time fields is not valid.

QSVCbadCMOS (0x80010008)
: The CMOS real-time clock does not seem to be operating.

  This return code is possible only on an Intel system, and then only if the system was built such that the CMOS clock is turned on, and the CMOS support code has been included in the SVC Handler.

SVC Handler Generated Faults

QSVCinvalid (x80)
: if the SVC Handler was compiled with the option to restrict this SVC to supervisor mode callers, then a user mode caller is faulted with this error code.

CPSleep (SLEEP) - Suspend for a Specified Period of Time

Function

This call suspends the caller for the specified period of time.

If while the task is waiting for the specified period, a signal arrives that causes a signal handler to be entered and this SVC is then resumed after the return from the signal handler, the “sleep” is restarted from the point of interruption, that is, it waits for only the remaining period.

C Syntax

```
long int CPSleep(unsigned long int Period,
                 unsigned long int Unit);
```

Parameters

Period The time to wait. If this time is in seconds, it is limited to 86400 seconds (that is, one day).

If this time is 0, this SVC is treated as a DISP_RET SVC. See section “CPYield (DISP_RET) - Return to Dispatcher” on page 2-7 for more information.

Unit If set to 0, the time is in micro-seconds. If set to 1, the time is in seconds.

Return Codes

QSVCgood (0)

Operation completed successfully.

QSVCbadreq (0x8001001E)

This can occur for one of the following reasons:

- *Unit* has a value other than 0 or 1.
- *Unit* is set to 1 and *Period* is greater than 86400 seconds (one day).

QSVCbadCMOS (0x80010008)

The CMOS real-time clock does not seem to be operating.

This return code is possible only on an Intel system, and then only if the system was built such that the CMOS clock is turned on, and the CMOS support code was included in the SVC Handler by the appropriate compile-time options.

QSVCinterrupt (0x80010024)

This SVC was interrupted by a signal, and the task might not have waited for the full period.

SVC Handler Generated Faults

None

CPTimerCancel (TIMER_CNCL) - Cancel a Timer Event

Function

This call cancels an outstanding timer event that was set up by a TIMER_SET SVC.

Note: Only timers allocated by ALLOC_TIMER SVCs can be used for TIMER_CNCL SVCs. Specifically, every task has a timer block allocated by the system, that is used only for time-outs with other SVCs and for SLEEP SVCs. This timer block cannot be specified in a TIMER_CNCL SVC.

C Syntax

```
long int CPTimerCancel(unsigned long int TimerID);
```

Parameters

TimerID The timer ID (handle) for this event, as returned by the ALLOC_TIMER SVC that allocated this timer.

Return Codes

QSVCgood (0)
Operation completed successfully.

QSVCbadid (0x8001002D)
The specified timer ID is not valid, for example it is not a valid timer ID, or it is not the ID of a timer block allocated to the calling task by an ALLOC_TIMER SVC.

QSVCbadtimer (0x8001002B)
The specified timer is not active, for example it might have already occurred.

SVC Handler Generated Faults

none

CPTimerSet (TIMER_SET) - Setup a Timer Event

Function

This call sets up a timer event to occur at some point in the future. It uses a timer block previously allocated to the calling task by means of a CPCreateTimerBlock (ALLOC_TIMER) SVC. If the specified timer is already active when this SVC is issued, the timer is reset without comment as specified by the new SVC. This is as if an implied CPTimerCancel (TIMER_CNCL) SVC were issued first.

Note: Only timers allocated by ALLOC_TIMER SVCs can be used for TIMER_SET SVCs. Specifically, every task has a timer block allocated by the system, that is used only for time-outs on other SVCs and SLEEP SVCs. This timer block cannot be specified in a TIMER_SET SVC.

The timer event can be repetitive, recurring at specified intervals, or can be a once-only event. In the latter case, the requested period should be set to 0. If the event is not repetitive, it is automatically cancelled when the event occurs. A repetitive event continues until explicitly cancelled by the caller.

See the section “Timer Queue Structures” in the SVC Handler manual for more information about the queueing of timer requests.

When the timer event occurs, either a synchronization semaphore is cleared, or a message is sent to a task or message queue. The format of messages sent as a result of timer events is specified in the section “Timer Messages” in the SVC Handler manual. If a message is sent to notify the occurrence of a timer event, a bit is set in the timer control block to indicate that there is a pending but not yet received timer notification. This bit is cleared when the timer message is received. In the case of a repetitive event, no further messages are sent by the SVC Handler for this timer until the pending message has been received. This prevents a timer from generating messages which are not being received, because this could result in a system abend due to running out of RQEs. This has at least one known side effect. If a task that has a repetitive timer every few seconds, and that task is being debugged with a task level debugger, it is highly likely that timer events will be missed as a result of the task being stopped for periods by the debugger. For example, this might occur when the user is examining the registers or data of the program being debugged.

The TIMER_TICK SVC is a simplified (and faster) variant of this SVC. See the section “CPTimerTick (TIMER_TICK) - Setup a Timer Event” on page 10-12 for more information.

Timer Request Modes:

Timer events requested by this call can be in one of the following modes:

Mode 0 - Relative

This mode is used to request a timer event at a time in the future, expressed relative to the current time. That is, the current time is added to the requested time to give the time at which the first event is to occur. If the resultant time is greater than or equal to one day, one day is subtracted from the time, and the event occurs at the resultant time after midnight. The requested time must be less than or equal to one day, or the request is rejected.

The event occurs at the specified time, calculated as described above. If a non-zero period is specified, subsequent events are generated, each occurring at a time interval specified by the period after the previous event.

Mode 1 - Absolute Today
This mode is used to request a timer event to occur at an absolute time of the current day. If the requested time has already passed, that is, the current time is greater than or equal to the requested time, the request is regarded as late, and a timer event is sent immediately (actually on the next timer tick). The requested time must be less than one day, or the SVC returns an error. A request for an event to occur at midnight should specify a time of zero.

Assuming the request is not late, the event occurs at the specified time. If a non-zero period is specified, subsequent events are generated, each occurring at a time interval specified by the period after the previous event. Late repetitive requests are handled specially. For example, if a timer event is to occur at time 1000 and with a period of 500, but is requested at time 2700, then one event is generated immediately, and subsequent events are generated at times 3000, 3500, 4000, and so on until the timer request is cancelled. No timer events are generated for the lost times 1500, 2000, and 2500. This mode can be used to specify timer events of the form “every hour on the hour starting at the stated hour”.

Mode 2 - Absolute Forwards
This mode is used to request a timer event at an absolute time in the future.

For a non-repetitive request (period = 0), if the requested time has already passed, that is the current time of day is greater than or equal to the requested time, the request is regarded as being early for the following day. The requested time must be less than one day, or the SVC returns an error. A request for an event to occur at midnight should specify a time of zero.

When the timer event is to be repetitive (period ≠ 0), if the the requested time is earlier than the current time of day, the event occurs at the specified time. Subsequent events are generated, each occurring at a time interval specified by the period after the previous event. The handling of late repetitive requests is different from mode 1, in that no event is returned immediately. For example, if a timer event is to occur at time 1000 and with a period of 500, but is requested at time 2700, then events are generated at times 3000, 3500, 4000, and so on, until the timer request is cancelled. No timer events are generated for the start time of 1000, or for the lost times 1500, 2000, and 2500. This mode can be used to specify timer events of the form “every hour on the hour starting at the next hour”.

C Syntax

```
long int CPTimerSet(unsigned long int Time,
                    unsigned long int Period,
                    unsigned long int Unit,
                    unsigned long int Mode,
                    unsigned long int SVid,
                    unsigned long int TimerID,
                    unsigned long int Token);
```

Parameters

Time The time of the first event. If the time is specified in seconds, this value is limited to one day (86399 seconds, because midnight is time 0).

Period The period for repetition of the event. This can be zero, in which case the event occurs only once and is automatically cancelled. Alternatively, if this is non-zero, this specifies the time between successive events after the first. If the time is specified in seconds, this value is limited to one day (86400 seconds).

Unit If set to 0, the time and period are in μseconds. If set to 1, the time and period are in seconds.

Mode This is bit significant, thus:

bits 0-28 (0xFFFFFFF8)
Unused, must be 0.

bit 29 (0x00000004)
If set, the timer event message is to be sent at priority 255 instead of *QSVCtimerprty* (= 24).

bits 30,31 (0x00000003)
The mode of the timer event (described above). This may take the value 0, 1 or 2.

SVid The SVid of the task or message queue to receive a message, or synchronization semaphore to be cleared, when this event occurs. This register can be set to 0, which implies that a message is sent to the message queue of the calling task.

If the user exit **UCtimoccur** is defined (see the section “User Code Exits” in the appendices to the SVC Handler manual), then the value of this parameter can also be 0xFFFF*xxxx*, where *xxxx* are any 4 hex digits the caller wants. This value has the effect that the normal SVC Handler timer notification does not occur, but the user exit routine is called instead.

TimerID The ID of the timer block to use. This is the ID (handle) returned by a CreateTimerBlock (ALLOC_TIMER) call.

Token An identifier specified by the user. It can accept any value in the range 0-0xFFFFFFFF. This is returned in the message when the timer event occurs. This identifier has no use if the timer notification is by a semaphore, or by calling the user exit routine (although the user exit routine can use it if needed).

Return Codes

QSVCgood (0)
Operation completed successfully.

QSVCbadreq (0x8001001E)
One of the parameters is invalid, for example:

- The specified time or period is not valid (that is, greater than one day).
- Bits 0-28 of *Mode* are not all 0.
- *SVid* ≠ 0 (the user exit **UCtimoccur** is not defined).

- *SVid* ≠ 0 and *SVid* ≠ 0xFFFF*xxxx*. (the user exit **UCtimoccur** is defined).

QSVCbadCMOS (0x80010008)
: The CMOS real-time clock does not seem to be operating.

  This return code is possible only on an Intel system, and then only if the system was built such that the CMOS clock is turned on, and the CMOS support code has been included in the SVC Handler.

QSVCbadid (0x8001002D)
: The specified timer ID is not valid or is not the ID of a timer block allocated to the calling task by an ALLOC_TIMER SVC.

QSVCbadSVid (0x80010002)
: The specified SVid is not that of a task, message queue, or synchronization semaphore.

SVC Handler Generated Faults

QSVCinvSVid (0x82)
: The requestor cannot access the specified SVT entry.

CPTimerTick (TIMER_TICK) - Setup a Timer Event

Function

This call sets up a timer event to occur at some point in the future. It uses a timer block previously allocated to the calling task by means of an ALLOC_TIMER SVC. If the specified timer is already active when this SVC is issued, the timer is simply reset without comment as specified by the new SVC. This is as if an implied TIMER_CNCL SVC were issued first.

Note: Only timers allocated by CPCreateTimerBlock (ALLOC_TIMER) SVCs can be used for TIMER_TICK SVCs. Specifically, every task has a timer block allocated by the system, which is used only for time-outs on other SVCs and SLEEP SVCs. This timer block cannot be specified in a TIMER_TICK SVC.

See the section “Timer Queue Structures” in the SVC Handler manual for more information about queueing timer requests.

This call is a simplified (and faster) variant of CPTimerSet (TIMER_SET). The differences are:

- The event is a “single shot” only, that is, it cannot be repetitive.
- The only method of event notification is by a message to the calling task. The semaphore or other message queue cannot be used.
- The time period is specified in ticks.
- There is no choice of event mode. The event is always relative to the current time of day, that is, mode 0 of the TIMER_SET SVC.

When the timer event occurs, a message is sent to the calling task. The format of messages sent as a result of timer events is specified in the section “Timer Messages” in the SVC Handler manual.

C Syntax

```
long int CPTimerTick(unsigned long int Time,
                     unsigned long int Notify,
                     unsigned long int TimerID,
                     unsigned long int Token);
```

Parameters

Time	The time of the event, specified in ticks from the current time. This value is limited to one day, which is 11059199 ticks at the default tick rate of 128 Hertz, since midnight is time 0.
Notify	This parameter must normally be set to 0, which implies that the notification of the timer event will be by a message sent to the message queue of the calling task. If the user exit **UCtimoccur** is defined (see the section “User Code Exits” in the appendices to the SVC Handler manual), then the value of this parameter may alternatively be 0xFFFF*xxxx*, where *xxxx* are any 4 hex digits chosen by the caller. This value has the effect that a message is not sent, but the user exit routine is called instead. The value *xxxx* can used by the user exit routine for any desired purpose, since the user exit routine has direct access to the timer control block for this timer event.

TimerID The ID of the timer block to use. This is the ID (handle) returned by a CreateTimerBlock (ALLOC_TIMER) call.

Token An identifier specified by the user. It can accept any value in the range 0-0xFFFFFFFF. This is returned in the message when the timer event occurs. This identifier has no use if the timer notification is by a semaphore or by calling the user exit routine (although the user exit routine can use it).

Return Codes

QSVCgood (0)
Operation completed successfully.

QSVCbadreq (0x8001001E)
This is for one of the following reasons:

- The specified time is not valid (greater than one day).
- *Notify* ≠ 0 (the user exit **UCtimoccur** is not defined).
- *Notify* ≠ 0 and EDX ≠ 0xFFFF*xxxx*. (the user exit **UCtimoccur** is defined).

QSVCbadCMOS (0x80010008)
The CMOS real-time clock does not seem to be operating.

This return code is possible only on an Intel system, and then only if the system was built such that the CMOS clock is turned on, and the CMOS support code has been included in the SVC Handler.

QSVCbadid (0x8001002D)
The specified timer ID is not valid or is not the ID of a timer block allocated to the calling task by an ALLOC_TIMER SVC.

SVC Handler Generated Faults

None

Chapter 11. Miscellaneous SVCs

CPCMOSRead (CMOS_READ) - Read CMOS Location

Function

This call returns the value held in a location of the CMOS read-only memory. Reading CMOS register 12 (0x0C) is not permitted, because doing so can interfere with the operation of the timer or clock.

Note: A SLEEP system build option can disable the CMOS read code. Also, an SVC Handler compile-time option can omit the CMOS support code. If either of these options are exercised, CMOS_READ SVCs always receive the return code *QSVCnotavail*.

C Syntax

```
long int CPCMOSRead(unsigned long int Address,
                    unsigned long int *Contents);
```

Parameters

Address	The CMOS address to be read, in the range 0-63 (the value 12 is not permitted).
Contents	A pointer to a 32-bit location to receive the contents of the CMOS byte.

Return Parameters

**Contents*	The contents of the specified byte of CMOS memory.

Return Codes

QSVCgood (0)
Operation completed successfully.

QSVCbadreq (0x8001001E)
The the specified CMOS address is not valid or out of range.

QSVCbadCMOS (0x80010008)
The CMOS real-time clock does not seem to be operating.

QSVCnotavail (0x80010021)
This facility is not available (bit 6 is set in the NDA field *SVCbits*, or the CMOS support code has been omitted from the SVC Handler by a compile-time option).

SVC Handler Generated Faults

None

CPGetCDA (GET_SDA) - Obtain CDA/SDA Offset

Function

This call returns the virtual address of the user mode read-only alias to the first page of the CDA on Intel or the virtual address of the SDA on PowerPC systems.

C Syntax

```
long int CPGetCDA(unsigned long int *Address);
```

Parameters

Address	A pointer to a 32-bit location to receive the returned address.

Return Parameters

**Address*	Overwritten with the address to access the CDA.

Return Codes

QSVCgood (0)
Operation completed successfully

SVC Handler Generated Faults

None

CPReadUItem (READ_UITEM) - Read Contents of User-Defined SVT Entry

Function

This call obtains the data held in a user-defined SVT entry.

C Syntax

```
long int CPReadUItem(unsigned long int SVid,
                     void *DataBuf);
```

Parameters

SVid SVid of the SVT entry to be read.

DataBuf A pointer to a 24-byte writeable buffer to receive the contents of the SVT entry.

Return Parameters

**DataBuf* If the return code is *QSVCgood*, this buffer is overwritten with the contents of the SVT entry.

Return Codes

QSVCgood (0)
Operation completed successfully.

QSVCbadSVid (0x80010002)
Specified SVT entry is not valid, or it is not a user-defined SVT entry. This return code can also occur if the calling task is a system task, but it is not permitted to access to the specified SVT entry.

QSVCdeadSVid (0x8001000C)
The specified SVT entry is dying or is in a process that is being removed from the system.

SVC Handler Generated Faults

QSVCparlist (0x81)
Parameter list or its address is not valid, for example *DataBuf* does not point to an allocated and writeable page.

QSVCinvSVid (0x82)
The caller is not a system task, and it is not permitted to access to the specified SVT entry.

If the specified buffer is not accessible, or if all or part of the buffer cannot have physical pages allocated to it, the following faults can occur:

- A general protection fault
- Stack fault
- Page fault

CPSVC - Intel

Function

This call provides a general interface to the SVC handler from a C program. Any SVC may be issued by setting up the register structure correctly.

Note: This is not a special SVC.

C Syntax

```
long int CPSVC(void *SVCRegAddr);
```

Parameters

SVCRegAddr
: A pointer to a *SVCREG* structure, as follows. The locations in this structure should be set as required for the desired SVC.

```
typedef struct svcreg
{
   unsigned long int eax;
   unsigned long int ebx;
   unsigned long int ecx;
   unsigned long int edx;
   unsigned long int esi;
   unsigned long int edi;
} SVCREG;
```

Figure 11-1. CPSVC Register Structure - Intel

Return Parameters

**SVCRegAddr*
: The fields of this structure are set to contain the registers as returned by the SVC Handler.

Return Codes

See the specification for the SVC being issued.

SVC Handler Generated Faults

See the specification for the SVC being issued.

CPSVCTrace (SVCTRACE) - SVC Handler Trace Facility - Intel

Function

This call enables, disables or queries the state of SVC Handler tracing, to insert a "user" entry into the trace buffer, and also to obtain the trace information. Various classes of tracing may be enabled separately. An attempt to turn on tracing when there is no SVC Handler trace buffer results in an error return code.

See the section "SVC Handler Trace Facility" in the SVC Handler manual for further information.

Warning by the Programmer General: Enabling SVC Handler tracing can seriously degrade the performance of some or all the features of the SVC Handler. However, if some classes are enabled, there is no overhead (apart from testing the bit) for other trace classes.

C Syntax

```
long int CPSVCTrace(unsigned long int Action,
                    unsigned long int Flags,
                    unsigned long int Incount,
                    void *BufAddr,
                    unsigned long int SVid,
                    unsigned long int *OutCount);
```

Parameters

Action The desired action or option, as follows:

QSVCTtron = 0

Enable tracing. This is done selectively, as specified by the bits of the *Flags* parameter. There is a restriction that only one of bits 23, 29 and 31 may be set concurrently.

Any bits set in *Flags* are ORed into the current trace enable mask (held in the field *svhtrbits* of the NDA); that is, events previously enabled and whose enable bit is not set here are not disabled by this SVC.

If, when this SVC is issued, the tracing enable mask currently has bits 1-31 unset then the trace buffer pointers are reset to the beginning of the trace buffer, that is the buffer is "cleared." Any previous trace buffer contents, including "user" trace entries, are lost.

If *Flags* bit 31 is set (SVCs are to be traced), then *InCount* may be set to 0, in which case all SVC functions are traced; alternatively, *InCount* may be set to a byte count *n*, and *BufAddr* then points to a buffer of length *n* bytes. In this latter case, each of the bytes in the buffer specifies an SVC function code, and only those SVC functions listed in this vector are traced. Thus it is possible, for example, to trace just the send and receive message SVCs. If, when this SVC is issued, the SVC tracing bit was already set, then the SVC function codes specified here are in addition to any previously specified - the others are not cleared. However, if the SVC trace bit was previously unset, on completion of this SVC only the SVC functions specified in this call are to be traced - the others are not traced.

If *Flags* bit 29 is set (hardware interrupts are to be traced), then *InCount* may be set to 0, in which case all interrupt levels are traced; alternatively, *InCount* may be set to a byte count *n*, and *BufAddr* then points to a buffer of length *n* bytes. In this latter case, each of the bytes in the buffer specifies an interrupt level (in the range 0-31), and only those interrupt levels specified in this vector are traced. If, when this SVC is issued, the interrupt tracing bit was already set, then the interrupt levels specified here are in addition to any previously specified - the others are not cleared. However, if the interrupt trace bit was previously unset, on completion of this SVC only the interrupt levels specified in this call are to be traced - the others are not traced.

If *Flags* bit 23 is set (extension SVCs are to be traced), then *InCount* may be set to 0, in which case all external SVC functions are traced; alternatively, *InCount* may be set to a byte count *n*, and *BufAddr* then points to a buffer of length *n* bytes. In this latter case, each of the bytes in the buffer specifies an extension SVC function code, and only those functions specified in this vector are traced. If, when this SVC is issued, the extension SVC tracing bit was already set, then the function codes specified here are in addition to any previously specified - the others are not cleared. However, if the extension SVC trace bit was previously unset, on completion of this SVC only the extension SVC functions specified in this call are to be traced - the others are not traced.

Unless *Flags* bit 23, 29 or 31 is set and *InCount* is non-zero, there is no buffer for this form of the SVCTRACE SVC - the *BufAddr* parameter is not used.

QSVCTtroff = 1
Disable tracing. This is done selectively, as specified by the bits of the *Flags* parameter. The state (enabled/disabled) remains unchanged for those types of trace event for which the bit is not set in *Flags*.

There is no buffer for this form of the SVCTRACE SVC - the *BufAddr* parameter is not used.

QSVCTutrace = 2
User trace. *InCount* is a double word count, in the range 1-*QSVCmaxusertrace* (=16) of trace data pointed to by *BufAddr*. A user trace item is created, containing the specified trace data.

This receives the return code *QSVCtraceoff* and the data is not saved if user tracing is turned off, that is *svhtrbits* bit 24 is unset.

This receives the return code *QSVCtracefull* and the data is not saved if *svhtrbits* bit 0 is set and the new trace item does not fit in the trace buffer.

QSVCTqstate = 3
Query trace state. *BufAddr* points to a *SVHTRACE_QRY* structure, into which is placed the following information.

```
typedef struct svhtrace_qry
{
   unsigned long  trqry_bits;                  /*  trace enable bits       */
   unsigned long  trqry_size;                  /*  trace buffer size       */
   unsigned long  trqry_start;                 /*  trace buffer start      */
   unsigned long  trqry_free;             /*  trace buffer free area start  */
   unsigned long  trqry_funcbits[8];           /*  SVC trace function bits  */
} SVHTRACE_QRY;
```

Figure 11-2. SVC Trace Query Structure - Intel

bytes 0-3 *trqry_bits*
Current trace enable bit mask (the NDA field *svhtrbits*).

bytes 4-7 *trqry_size*
Size (in bytes) of the trace buffer.

bytes 8-11 *trqry_start*
Offset from the start of the trace buffer of the start of the data in the buffer.

bytes 12-15 *trqry_free*
Offset from the start of the trace buffer of the start of the free area in the trace buffer (i.e. a pointer to the byte immediately after the last data byte in the trace buffer).

bytes 16-47 *trqry_funcbits*
SVC function trace bits. This is a vector of 256 bits, one bit per possible SVC function. Bit *i* is set if SVCs with function code *i* are to be traced. The contents of this vector are meaningful if and only if bit 31 is set in the SVC Handler trace enable bit mask returned in bytes 0-3 of the buffer. This bit vector is in the order required by the Intel “bit test” instructions; that is, when viewed as a sequence of bits, the bits are in the order 7-0, 15-8, 23-16, ..., 255-248.

QSVCTqdata = 4
Query trace state, and obtain the trace data. *BufAddr* points to a buffer which consists of a *SVHTRACE_QRY* structure (see above), followed by sufficient space to receive the contents of the trace buffer. If the parameter list buffer is not large enough, the calling task's data may be overwritten, or the calling task may fail with a general protection fault. An SVCTRACE SVC with *Action*=3 may be used to determine the necessary buffer size; in the current version of CP/Q, the maximum buffer size required is 384K+48 bytes.

The data returned is as for *Action*=3, plus the entire current trace buffer contents, regardless of whether the buffer contents are meaningful or useful or not.

QSVCTqcount = 5
Get SVC counts. *BufAddr* points to a buffer of length *InCount* double words to receive a copy of the SVC count vector. The first *InCount* entries of the count vector are placed in the buffer. On return, *OutCount* holds the number of entries returned - this is smaller than the value in the call if the supplied *InCount* value is larger than the size of the SVC count vector in the NDA (currently 256 double words).

The returned count vector is simply a sequence of double words. Element *i* in this vector is the count of SVCs with function code *i*. Thus element 0 is the count of DISP_RET SVCs. Certain SVC function codes are not used, for example function 4; for such unused SVC codes, the corresponding entry in the count vector is always returned as 0.

QSVCTptsvon = 6
Per-task tracing of SVCs (both “built-in” SVCs and installable extension SVCs) is to apply to the task whose SVid is in *SVid*.

Tracing occurs only if either or both bits 23 and 31 of the NDA field *SVCtrbits* have been set.

QSVCTptsvoff = 7
Per-task tracing of SVCs (both “built-in” SVCs and installable extension SVCs) is not to apply to the task whose SVid is in *SVid*.

If *SVid* is set to 0, per-task tracing is turned off for all tasks.

QSVCTptexiton = 8
Per-task tracing of SVC Handler exits (from both “built-in” SVCs and installable extension SVCs) is to apply to the task whose SVid is in *SVid*.

Tracing occurs only if either or both bits 22 and 30 of the NDA field *SVCtrbits* have been set.

QSVCTptexitoff = 9
Per-task tracing of SVC Handler exits (from both “built-in” SVCs and installable extension SVCs) is not to apply to the task whose SVid is in *SVid*.

If *SVid* is set to 0, per-task tracing is turned off for all tasks.

Flags A bit significant field, used for *Action* = 0 and 1, to specify which trace actions are to occur or are to be stopped, as follows. The corresponding bits are set (*Action*=0) or unset (*Action*=1) in the NDA field *SVCtrbits*, as appropriate.

bit 0 - QSVCTnowrap (0x80000000)
If set, SVC Handler tracing will cease (bits 22-31 will be set to 0) when the trace buffer becomes full (i.e. it will not wrap around and over-write the old data in the buffer) for *Action* = 0.

For *Action* = 1, the “stop trace on buffer full” facility is turned off.

bit 1 - QSVCTsvc_pt (0x40000000)
If set, the tracing of SVC Handler exits is to be on a per-task basis (*Action* = 0) or for every task (*Action* = 1).

bit 2 - QSVCTexit_pt (0x20000000)
If set, the tracing of SVCs is to be on a per-task basis (*Action* = 0) or for every task (*Action* = 1).

bits 3-21 (0x1FFFFC00)
Unused, should be 0.

bit 22 - QSVCTextexit (0x00000200)
If set, returns from installable extension SVCs are to be traced (*Action* = 0) or not traced (*Action* = 1).

bit 23 - QSVCTextsvc (0x00000100)
If set, installable extension SVCs are to be traced (*Action* = 0) or not traced (*Action* = 1). For *Action* = 0, the tracing of each external SVC function can be controlled independently, according to the value supplied in *InCount* and the buffer pointed to by *BufAddr* (see above).

bit 24 - QSVCTdouser (0x00000080)
If set, "user trace" SVCs (SVCTRACE with AL = 2) is to be implemented (*Action* = 0) or not implemented (*Action* = 1).

bit 25 - QSVCTtimestmp (0x00000040)
If set, time stamps (the start of each new second of time) are to be traced (*Action* = 0) or not traced (*Action* = 1).

bit 26 - QSVCTtskchg (0x00000020)
If set, task changes are to be traced (*Action* = 0) or not traced (*Action* = 1).

bit 27 - QSVCTtskflt (0x00000010)
If set, task faults are to be traced (*Action* = 0) or not traced (*Action* = 1).

bit 28 - QSVCTuiint (0x00000008)
INT instructions specifying an un-initialized IDT entry are to be traced (*Action* = 0) or not traced (*Action* = 1).

bit 29 - QSVCThwint (0x00000004)
If set, hardware interrupts (apart from those for the timer) are to be traced (*Action* = 0) or not traced (*Action* = 1). For *Action* = 0, the tracing of each interrupt level can be controlled independently, according to the value supplied in *InCount* and the buffer pointed to by *BufAddr* (see above).

bit 30 - QSVCTsvcexit (0x00000002)
If set, SVC Handler exits from "built-in" SVCs (i.e. those within the SVC Handler, Memory Manager and Resource Manager) to the calling task are to be traced (*Action* = 0) or not traced (*Action* = 1).

bit 31 - QSVCTsvc (0x00000001)
If set, "built-in" SVCs (i.e. those within the SVC Handler, Memory Manager and Resource Manager) are to be traced (*Action* = 0) or not traced (*Action* = 1). For *Action* = 0, the tracing of each SVC function can be controlled independently, according to the value supplied in *InCount* and the buffer pointed to by *BufAddr* (see above).

InCount	Byte count or buffer size for *Action* = 0, 2, 3, 4 and 5.
BufAddr	Pointer to a data buffer for *Action* = 0, 2, 3, 4 and 5.
SVid	The SVid for per-task tracing for *Action* = 6, 7, 8 and 9.
OutCount	A pointer to a 32-bit location to receive the returned count for *Action* = 5.

Return Parameters

**BufAddr* The buffer is overwritten for calls with *Action* = 3, 4 and 5, as described above. The buffer is preserved for calls with *Action* = 0 and 2. This parameter is unused for calls with *Action* =1 and *Action* =6-9.

**OutCount*
The returned count for calls with *Action* =5. This parameter is otherwise unchanged.

Return Codes

QSVCgood (0)
Operation completed successfully.

QSVCnotrace (0x80010016)
There is no SVC Handler trace area in the system.

QSVCtracefull (0x80010025)
The trace area is full - a user trace request (AL=2) has not been implemented.

QSVCtraceoff (0x80010026)
User tracing is turned off - the user trace request (AL=2) has not been implemented.

QSVCbadreq (0x8001001E)
This can be for one of the following reasons:

- *Action* not in the range 0-9.
- *Action*=0, and more than one of bits 23, 29 and 31 is set concurrently in the parameter *Flags*.
- *Action*=0, and bits 3-21 of *Flags* are not all zero.

QSVCbadSVid (0x80010002)
The specified SVid is invalid in some manner (*Action* =6-9 only).

QSVCbadint (0x80010019)
When *Action*=0 and *Flags* bit 29 is set, one or more of the interrupt levels specified in the buffer is invalid. In this case, the trace bit is not set for any hardware interrupt level.

QSVCdeadSVid (0x8001000C)
The specified SVid is being deleted from the system (*Action* =6-9 only).

SVC Handler Generated Faults

QSVCparlist (0x81)
Parameter list or its address is invalid, for example the parameter *BufAddr* does not point to an allocated page.

CPWriteUItem (WRITE_UITEM) - Write Contents of User-Defined SVT Entry

Function

This call changes the data held in a user-defined SVT entry.

C Syntax

```
long int CPWriteUItem(unsigned long int SVid,
                      void *DataBuf);
```

Parameters

SVid SVid of the SVT entry to be read.

DataBuf A pointer to a 24-byte readable buffer containing the new contents for the SVT entry.

Return Codes

QSVCgood (0)
Operation completed successfully.

QSVCbadSVid (0x80010002)
Specified SVT entry is not valid, or is not a user-defined SVT entry. This return code can also occur if the calling task is a system task, but it is not permitted to access to the specified SVT entry.

QSVCdeadSVid (0x8001000C)
The specified SVT entry is dying or is in a process that is being removed from the system.

SVC Handler Generated Faults

None

QSVCparlist (0x81)
Parameter list or its address is not valid. For example *DataBuf* does not point to an allocated page.

QSVCinvSVid (0x82)
The caller is not a system task, and it is not permitted to access to the specified SVT entry.

If the specified buffer is not accessible, or if all or part of the buffer cannot have physical pages allocated to it, the following faults can occur:

- A general protection
- Stack
- Page fault

Memory Manager Calls

Chapter 12. Section Notes

All routines in this section require that *cpqlib.h* be included in the calling program.

All structures and typedefs which appear in this section are defined in *cpqlib.h* and in files thereby automatically included. In particular, definitions for the bits in the Type parameter can be found in the file *memconst.h*.

Note: The Memory Manager may enable interrupts while processing a call. Applications which disable interrupts should take this into consideration.

Chapter 13. Allocation Functions

CPAllocBase

```
Syntax -

     long int CPAllocBase(Type,Size,&Offset,Real_Address)

     unsigned long int Type;
     unsigned long int Size;
     void *Offset;
     unsigned long int Real_Address;
```

Type	This value contains information about the attributes of the memory object being created.
Size	This is the number of bytes required. The number of bytes requested is rounded up to a multiple of the page size. The maximum size object that can be allocated is 2GB-1.
Offset	Variable in which the offset of the object created is returned. Also, if **QMthis_off** is set in the Type field, then this is the requested offset where the memory object should be created.
Real_Address	This is the required 32-bit physical address, and must be aligned at a page boundary.

Usage Notes

This function is used to obtain storage which begins at a particular real address and is intended to support user-managed real memory or system memory mapped devices. The function is reserved to callers with I/O privilege. The new object is always designated nonswappable and long-term fixed. The requested real storage must begin at an address that is on a page boundary, and the size is rounded up to a multiple of the page size. Allocation is made from free adapter memory.

Note: This function and CPAllocRange are the only functions that access adapter memory.

CP/Q uses the concept of regions of real storage. There are two types of regions:

- General purpose storage is memory that is used to satisfy general memory allocations. It is not available for allocation through this call.
- Adapter storage is memory that is defined as some special type of storage such as adapter memory. It is not available for general allocation, but it is reserved for allocation through this call.

Because the object is obtained from free adapter memory, there is the implication that it is some special type of storage. The object is given a fix count of 1. The fixing of the underlying pages is considered to be a long-term fix.

It is possible to request that a specific offset be assigned to an object, and it is intended only for those cases where this capability is truly required. An example might be special purpose memory which is accessed and viewed by other hardware components which do not implement virtual memory and therefore, some performance advantage might be gained by a specified offset. Indiscriminate use of this function can cause unnecessary page tables/system control blocks to be created, thereby wasting real memory.

The Type parameter defines the following for a memory object:

- Object class of global, common, or private
- Specific offset required
- Address space (caller's effective address space or current address space)
- Privilege level, either user or supervisor
- Access, either read/write or read only

The Class of the memory object is defined as follows:

QMglobal
This indicates a Global class object.

QMprivate
This indicates a Private class object.

QMcommon
This indicates a Common class object.

The following apply only to Common objects:

- Privilege Level Limit

  QMshare_supvr
  This is used to indicate the limit of access to be allowed to other processes. Specifically, it allows Supervisor access.

  QMshare_user
  This is used to indicate the limit of access to be allowed to other processes. Specifically, it allows User access.

- Read/Write Access Limit

  QMshare_read
  It is used to indicate the limit of access to be allowed to other processes. Specifically, it allows read-only access.

  QMshare_write
  It is used to indicate the limit of access to be allowed to other processes. Specifically, it allows read/write access.

- Give/Get Access

  QMgive This indicates the access mechanism for Common class objects from the CPGiveMem call.

  QMget This indicates the access mechanism for Common class objects from the CPGetMem call.

- Copy Mode

  QMcopy It is used to indicate copy mode for the object.

  QMcopy_on_write
  This is valid only on creation of a copy mode common object. This indicates that the copy is to be performed by

copy on write. Immediate copy is performed if this is not explicitly requested.

Specific Offset is defined as follows:

QMthis_off
This is used if a specific offset is being requested for the object being created.

Address space is defined as follows:

QMnew_eff
This is used to indicate that the new object is to be created in the address space of the caller's effective address space.

QMnew_current
This is used to indicate that the new object is to be created in the current address space.

Privilege level is defined as follows:

QMuser This is used to indicate user privilege level is requested.

QMsupervisor
This is used to indicate supervisor privilege level is requested.

Access is defined as follows:

QMwrite This is used to indicate read/write access is requested,

QMread This is used to indicate read-only access is requested.

Implementation Notes

By the very nature of this function, the real storage pages assigned to map the requested object are contiguous pages in real storage. This function and CPAllocRange are the only functions that provides contiguous real pages.

This call creates a memory object and returns an Offset. The object is accessed or referenced by this Offset, not by the Real_Address. The only exception is for adapter memory in ranges created with the mapped option and being accessed by supervisor level code in systems whose DRMM (Direct Real Memory Map) starts at offset 0.

DMA or bus master I/O cannot be performed directly into Common Class, Copy-on-Write objects. A page alias must be made first.

Return Codes

The Memory Manager provides a return code on all calls. Except for a success return code (QMsuccess), which has a value of 0, all Memory Manager return codes have the form **0x8002xxxx**. The following are possible return codes for this call along with their low order 16 bits:

QMsuccess	Request was successful, offset returned.
QMbad_offset-(0x0001)	The caller requested a specific offset, but that offset was not available, not large enough, or not of the class requested.
QMbad_type-(0x0002)	Type field error - caller's privilege insufficient.

QMbad_size-(0x0004)	A size of zero or greater than 2GB-1 was specified.
QMbad_addr-(0x0005)	Address error - the address is not aligned on a page boundary.
QMbad_range-(0x0008)	The real address requested is not within a valid range of adapter memory.
QMno_block-(0x0010)	No free linear address block exists of sufficient size to map the requested real addresses.
QMno_page-(0x0012)	The real storage at the requested base address required new page tables to map it into linear address space, and no page frames were available to create the page tables.
QMreal_alloc-(0x0016)	The real address requested has been previously allocated and is unavailable.
QMproc_limit-(0x0018)	The allocation size would exceed the process's limits.
QMprocess_dead-(0x0022)	The process, in whose address space the new object was to be created, is in the final stages of the Memory Manager's process removal. No new requests to create memory objects can be honored. This should not normally occur, particularly if the Resource Manager resource provider facilities have been used.
QMrestricted_function-(0x0030)	Caller does not have the required privilege level.
QMbusy_fork-(0x0035)	The process, in whose address space the new object was to be created, is performing a fork operation. The request cannot be accommodated at this time.

Memory Manager Generated Faults

It is possible that the memory manager will detect internal or system errors during the processing of this call. If this occurs, an internal error code is either returned to the caller, or is transmitted to the caller's fault handler as a result of the Memory Manager issuing a CPFaultTask call. In addition, certain fault conditions relating to memory objects may be detected asynchronously by the page fault handler and reported to the task's fault handler. See *SPL Volume 4: Memory Manager* for a description of the error codes that can occur.

CPAllocMem

```
Syntax -

     long int CPAllocMem(Type,Size,&Offset)

     unsigned long int Type;
     unsigned long int Size;
     void *Offset;
```

`Type`	This value contains information about the attributes of the memory object being created.
`Size`	This is the number of bytes required. The number of bytes requested is rounded up to a multiple of the page size. The maximum size object that can be allocated is 2GB-1.
`Offset`	A variable in which the offset of the object created is returned. Also, if **QMthis_off** is set in the Type field, then this is the requested offset where the memory object should be created.

Usage Notes

This function is used to create a memory object. This function obtains storage from “free storage”, that is, from free linear address space in the class of memory specified by the caller. As required, the memory manager maps this storage to real memory. If assignment at allocation was requested, page frames are immediately assigned to the entire object. Otherwise, page frames are assigned by the page fault handler when and if the page is actually used. In either case the page frames are taken from the list of free general purpose page frames.

In the case of sparse objects, no page frames are assigned at allocation, nor are frames assigned on page faults. Not present page faults are treated as errors and the task is faulted. Page frames can only be assigned to sparse objects by explicit requests to commit pages of real memory to the object. Sparse objects are normally intended for use by the system or system extensions.

When allocating Common class objects, the object is created only in the address space of the caller (as indicated in the Type field). Access to this object in another address space requires additional Memory Manager calls. The conditions under which access may be obtained are specified in the allocate call. The caller can specify a limit for access attributes for common objects. If specified, this limit must not be less restrictive than that requested by the caller. If not specified, it defaults to user/writeable. The caller must also specify the mode of a common object (shared or copy) and how access is to be obtained (give or get).

The caller must be at supervisor level to create a supervisor object.

It is possible to request that a specific offset be assigned to an object. This requires IOPL and is intended only for those cases where this capability is truly required. Indiscriminate use of this function can cause unnecessary page tables to be created, thereby wasting real memory.

The object obtained belongs to the caller's effective process.

The Type parameter defines the following for a memory object:

- Object class (global, common, or private)
- Type of physical memory (default, type 1, type 2, or type 3)
- Specific offset required
- Page frame assignment (at allocation or on page faults when pages are accessed)
- Address space (caller's effective address space or current address space)
- Sparse object, either yes or no
- Fix state, either create object with initial fix or not
- Privilege level, either user or supervisor
- Access, either read/write or read only

The Class of the memory object is defined as follows:

QMglobal
This indicates a Global class object.

QMprivate
This indicates a Private class object.

QMcommon
This indicates a Common class object.

The following apply only to Common objects:

- Privilege Level Limit

  QMshare_supvr
  This is used to indicate the limit of access to be allowed to other processes. Specifically, it allows Supervisor access.

  QMshare_user
  This is used to indicate the limit of access to be allowed to other processes. Specifically, it allows User access.

- Read/Write Access Limit

  QMshare_read
  This is used to indicate the limit of access to be allowed to other processes. Specifically, it allows read only access.

  QMshare_write
  This is used to indicate the limit of access to be allowed to other processes. Specifically, it allows read/write access.

- Give/Get Access

  QMgive This indicates the access mechanism for Common class objects using the CPGiveMem call.

  QMget This indicates the access mechanism for Common class objects using the CPGetMem call.

- Copy Mode

  QMcopy This is used to indicate copy mode for the object.

  QMcopy_on_write
  This is valid only on creation of a copy mode common object. This indicates that the copy is to be performed by copy on write. Immediate copy is performed if this is not explicitly requested.

These masks indicate the type of physical memory to use when making storage page assignments to this object as follows:

QMtype_default
A request for the default memory type. That is, use memory type 1 if available. Otherwise memory type 2 or 3 is used (in that order).

QMtype1, QMtype2 or QMtype3
A request for a specific memory type. That is, use only the memory type that is specified.

Specific Offset is defined as follows:

QMthis_off
This is used if a specific offset is being requested for the object being created. Use of this requires that the caller have I/O privilege.

Page frames are assigned as follows:

QMassign_alloc
This is used to indicate that real storage frames are to be assigned at allocation time. If this mask is not used, page frames are assigned only when and if the storage is touched and a page fault occurs.

Address space is defined as follows:

QMnew_eff
This is used to indicate that the new object is to be created in the address space of the caller's effective address space.

QMnew_current
This is used to indicate that the new object is to be created in the current address space.

Sparse object is defined as follows:

QMsparse
This is used if the object is sparse. Use of this requires that the caller have I/O privilege.

Fix status is defined as follows:

QMfixed This is used request that the object be fixed at allocation. Use of this requires that the caller have I/O privilege.

Privilege level is defined as follows:

QMuser This is used to indicate user privilege level is requested.

QMsupervisor
This is used to indicate supervisor privilege level is requested.

Access is defined as follows:

QMwrite This is used to indicate read/write access is requested.

QMread This is used to indicate read-only access is requested.

Implementation Notes

None

Return Codes

The Memory Manager provides a return code on all calls. Except for a success return code (QMsuccess), which has a value of 0, all Memory Manager return codes have the form **0x8002xxxx**. The possible return codes for this call, along with their low order 16 bits, are listed below.

QMsuccess	Request was successful, offset returned.
QMbad_offset-(0x0001)	The caller requested a specific offset, but that offset was not available, not large enough, or not of the class requested.
QMbad_type-(0x0002)	Type field error - the caller's privilege was insufficient for the attributes or function requested.
QMbad_size-(0x0004)	A size of zero or greater than 2GB-1 was specified.
QMno_block-(0x0010)	Sufficient free virtual storage does not exist to satisfy the request.
QMno_page-(0x0012)	Assignment of page frames was requested (**QMassign_alloc**), or additional page tables were required to map the memory allocated, and insufficient free page frames were available to satisfy the request.
QMproc_limit-(0x0018)	Allocation size would exceed process's limits.
QMprocess_dead-(0x0022)	The process, in whose address space the new object was to be created, is in the final stages of the Memory Manager's process removal. No new requests to create memory objects can be honored. This should not normally occur, particularly if the Resource Manager resource provider facilities have been used.
QMbusy_fork-(0x0035)	The process, in whose address space the new object was to be created, is performing a fork operation. The request cannot be accommodated at this time.

Memory Manager Generated Faults

It is possible that the memory manager will detect internal or system errors during the processing of this call. If this occurs, an internal error code is either returned to the caller, or is transmitted to the caller's fault handler as a result of the Memory Manager issuing a CPFaultTask call. In addition, certain fault conditions relating to memory objects may be detected asynchronously by the page fault handler and reported to the task's fault handler. See *SPL Volume 4: Memory Manager* for a description of the error codes that can occur.

CPAllocRange

```
Syntax -

     long int CPAllocRange(Type,Size,&Offset,&Real_Address)

     unsigned long int Type;
     unsigned long int Size;
     void *Offset;
     unsigned long int Real_Address;


Type      This value contains information about the attributes of the memory object being
          created.

Size      This is the number of bytes required.  The number of bytes requested is rounded up
          to a multiple of the page size.  The maximum size object that can be allocated is
          2GB-1.

Offset    Variable in which the offset of the object created is returned.  Also, if
          QMthis_off is set in the Type field, then this is the requested offset where the
          memory object should be created.

Real_Address
          This is the real address where the call starts a search for the requested
          contiguous block of storage.  This real address must be aligned at a page
          boundary.  The address can be the start of a range of adapter memory, or somewhere
          within a range of adapter memory.  In either case the search looks at ascending
          real addresses for a contiguous block of Size bytes, until one is found or the end
          of the range is reached.  If the function is successful, upon return this variable
          contains the real address of the start of the storage block used to create the
          memory object.
```

Usage Notes

This function is used to obtain storage within a particular range of adapter memory, and thus within a particular range of real addresses. It is intended to provide support for I/O devices requiring contiguous real storage areas larger than 4K bytes for DMA. The function is reserved to callers with I/O privilege. The new object is always designated non-swappable and fixed. The Real_Address provided as an input to the call must be an address within an existing range of adapter memory and must be on a page boundary. The Size is rounded up to a multiple of the page size. The memory range is examined starting at the specified Real_Address to determine if a set of contiguous pages exists within the range of the requested size. The object created must be within a single range. The first available block of storage found, if any, is used, and the Real_Address of the start of the block is returned along with the Offset (virtual address) of the created object. Allocation is made from free adapter memory. The resulting object must be accessed using it's Offset(virtual address).

Note: This is one of only two functions that accesses adapter memory.

CP/Q uses the concept of regions of real storage. There are two types of regions:

- General purpose storage is memory that is used to satisfy general memory allocations. It is not available for allocation through this call.

- Adapter storage is memory that is defined as some special type of storage. It is not available for general allocation, but it is reserved for allocation through this call and through the call CPAllocBase.

Because the object is obtained from free adapter memory, there is the implication that it is some special type of storage. The object is given a fix count of 1. The fixing of the underlying pages is considered to be a long-term fix. Furthermore, as adapter memory often has special characteristics and is generally used by device drivers or similar code, a specific real address range can only be directly allocated to one memory at a time. This means once a real address area has been allocated by this call, subsequent calls for this range or an overlapping range fail. This enables the device driver to retain full control over this memory. The device driver can allow access to the underlying memory through standard CP/Q Memory Manager calls such as CPAlias, CPGiveMem, and CPGetMem, if the object created by this call was in the common class.

It is possible to request that a specific Offset be assigned to an object, and it is intended only for those cases where this capability is truly required. An example might be special purpose memory which is accessed and viewed by other hardware components which do not implement virtual memory and, therefore, some performance advantage might be gained by a specified Offset. Indiscriminate use of this function can cause unnecessary page tables/control blocks to be created, thereby wasting real memory.

Special Case:
When **QMany_real** is set, this call behaves in a somewhat different manner than has just been described. In this case, the real address specified is not restricted to adapter memory. It can also refer to general purpose memory. It is allocated as long as the block of memory is entirely free and is within a single defined range of memory. Only the specific block beginning with the specified real address is examined. This case does not search blocks at ascending real addresses as in the normal case.

The Type parameter defines the following for a memory object:

- Object class of global, common, or private
- Specific offset required
- Address space (caller's effective address space or current address space)
- Privilege level, either user or supervisor
- Access, either read/write or read only

The Class of the memory object is defined as follows:

QMglobal
This indicates a Global class object.

QMprivate
This indicates a Private class object.

QMcommon
This indicates a Common class object.

The following apply only to Common objects:

- Privilege Level Limit

QMshare_supvr
This is used to indicate the limit of access to be allowed to other processes. Specifically, it allows Supervisor access.

QMshare_user
This is used to indicate the limit of access to be allowed to other processes. Specifically, it allows User access.

- Read/Write Access Limit

  QMshare_read
  This is used to indicate the limit of access to be allowed to other processes. Specifically, it allows read only access.

  QMshare_write
  This is used to indicate the limit of access to be allowed to other processes. Specifically, it allows read/write access.

- Give/Get Access

  QMgive This indicates the access mechanism for Common class objects through the CPGiveMem call.

  QMget This indicates the access mechanism for Common class objects through the CPGetMem call.

- Copy Mode

  QMcopy It is used to indicate copy mode for the object.

  QMcopy_on_write
  This is valid only on creation of a copy mode common object. This indicates that the copy is to be performed by copy on write. Immediate copy is performed if this is not explicitly requested.

Specific Offset is defined as follows:

QMthis_off
This is used if a specific offset is being requested for the object being created. Use of this requires that the caller have I/O privilege.

Address space is defined as follows:

QMnew_eff
This is used to indicate that the new object is to be created in the address space of the caller's effective address space.

QMnew_current
This is used to indicate that the new object is to be created in the current address space.

Privilege level is defined as follows:

QMuser This is used to indicate user privilege level is requested.

QMsupervisor
This is used to indicate supervisor privilege level is requested.

Access is defined as follows:

QMwrite This is used to indicate read/write access is requested,

QMread This is used to indicate read-only access is requested.

Implementation Notes

By the very nature of this function, the real storage pages assigned to map the requested object are contiguous pages in real storage. This function and the CPAllocBase function are the only ones that provides contiguous real pages.

This call creates a memory object and returns an Offset. The object is accessed or referenced by this Offset, not by the Real_Address. The only exception is for adapter memory in ranges created with the mapped option and being accessed by supervisor level code in systems whose DRMM (Direct Real Memory Map) starts at offset 0.

DMA or bus master I/O cannot be performed directly into Common Class, Copy-on-Write objects. A page alias must be made first.

Return Codes

The Memory Manager provides a return code on all calls. Except for a success return code (QMsuccess), which has a value of 0, all Memory Manager return codes have the form **0x8002xxxx**. The possible return codes for this call, along with their low order 16 bits, are listed below.

QMsuccess	Request was successful, Offset and Real_Address returned.
QMbad_offset-(0x0001)	The caller requested a specific offset, but that offset was not available or not of the class requested.
QMbad_type-(0x0002)	Type field error - the caller's privilege is insufficient for the attributes or function requested.
QMbad_size-(0x0004)	A size of zero or greater than 2GB-1 was specified.
QMbad_addr-(0x0005)	Address error. The address is not aligned on a page boundary.
QMbad_range-(0x0008)	The real address requested is not within a valid range of adapter memory.
QMno_block-(0x0010)	No free linear address block exists of sufficient size to map the requested real addresses, or a specific offset was requested and, while requested offset was available, there was not a block large enough for the size of the request.
QMno_page-(0x0012)	The real storage at the requested base address required new page tables to map it into linear address space, and no page frames were available to create the page tables.
QMreal_alloc-(0x0016)	All the possible areas within the specified area of the range have previously been allocated and are unavailable.
QMproc_limit-(0x0018)	The allocation size would exceed process's limits.
QMprocess_dead-(0x0022)	The process, in whose address space the new object was to be created, is in the final stages of the Memory Manager's process removal. No new requests to create memory objects can be honored. This should not normally occur, particularly if the Resource Manager resource provider facilities have been used.

QMrestricted_function-(0x0030)	Caller does not have I/O privilege.
QMbusy_fork-(0x0035)	The process, in whose address space the new object was to be created, is performing a fork operation. The request cannot be accommodated at this time.

Memory Manager Generated Faults

It is possible that the memory manager will detect internal or system errors during the processing of this call. If this occurs, an internal error code is either returned to the caller, or is transmitted to the caller's fault handler as a result of the Memory Manager issuing a CPFaultTask call. In addition, certain fault conditions relating to memory objects may be detected asynchronously by the page fault handler and reported to the task's fault handler. See *SPL Volume 4: Memory Manager* for a description of the error codes that can occur.

Chapter 14. Deallocation Functions

CPFreeObj

```
Syntax -

     long int CPFreeObj(Type,Offset)

     unsigned long int Type;
     void *Offset;

Type      This value determines the address space of the incoming object.

Offset    Offset of the object to be deleted.
```

Usage Notes

This function removes a memory object (or “pseudo object” in the case of an alias). The object must be owned in the caller's effective address space. If common shared mode, underlying storage is released if this is last user. If copy mode, storage is released.

Objects allocated from adapter memory have an implied fix count of 1 when allocated. These can be deleted with a fix count of 1. There is no need to unfix the implied fix.

The Type parameter is used to define the incoming object's address space, as follows:

QMold_eff
: This is used to indicate that the incoming object is defined in the address space of the caller's effective address space.

QMold_current
: This is used to indicate that the incoming object is defined in the caller's address space.

Implementation Notes

None

Return Codes

The Memory Manager provides a return code on all calls. Except for a success return code (QMsuccess), which has a value of 0, all Memory Manager return codes have the form **0x8002xxxx**. The possible return codes for this call, along with their low order 16 bits, are listed below.

QMsuccess	Request was successful, object deleted.
QMbad_offset-(0x0001)	Specified offset is not that of a valid memory object.
QMbad_type-(0x0002)	Type field error - the caller is trying to delete supervisor level object without supervisor privilege.

QMnot_owned-(0x0013)	Specified object is not owned by caller's effective process.
QMobj_fixed-(0x001A)	Specified object had non-zero fix count.
QMobj_accessed-(0x001B)	Specified object had non-zero verify count or outstanding aliases.
QMwrap_count-(0x0021)	The call could not be processed because the memory lock is at its maximum, that is, there are 255 tasks currently executing within Memory Manager code referring to this object.
QMbusy_fork-(0x0035)	The process in which the object to be removed exists is performing a fork operation. The request cannot be accommodated at this time.

Memory Manager Generated Faults

It is possible that the memory manager will detect internal or system errors during the processing of this call. If this occurs, an internal error code is either returned to the caller, or is transmitted to the caller's fault handler as a result of the Memory Manager issuing a CPFaultTask call. In addition, certain fault conditions relating to memory objects may be detected asynchronously by the page fault handler and reported to the task's fault handler. See *SPL Volume 4: Memory Manager* for a description of the error codes that can occur.

CPFreeRange

```
Syntax -

     long int CPFreeRange(Type,Real_Address)

     unsigned long int Type;
     unsigned long int Real_Address;


Type      This value defines the type of memory and its attributes.  It should currently be
          set to zero.

Real_Address
          The address in real memory of the first byte of the range to be freed.
```

Usage Notes

This function is used to free an existing range of adapter memory. The entire range of adapter memory must be unused at the time of the call, that is, there can be no allocated memory objects within the range. The real address must be that of the beginning of an existing range of adapter memory.

This function is only available to code having I/O privilege.

Implementation Notes

None

Return Codes

The Memory Manager provides a return code on all calls. Except for a success return code (QMsuccess), which has a value of 0, all Memory Manager return codes have the form **0x8002xxxx**. The possible return codes for this call, along with their low order 16 bits, are listed below.

QMsuccess	Request was successful, and the range has been freed.
QMbad_addr-(0x0005)	The real memory address was not page aligned.
QMbad_range-(0x0008)	The real memory address is not that of an existing range of adapter memory, or the range was not adapter memory.
QMreal_alloc-(0x0016)	The range was not entirely free (allocated objects exist in range).
QMrestricted_function-(0x0030)	Caller does not have the required privilege level.

Memory Manager Generated Faults

It is possible that the memory manager will detect internal or system errors during the processing of this call. If this occurs, an internal error code is either returned to the caller, or is transmitted to the caller's fault handler as a result of the Memory Manager issuing a CPFaultTask call. In addition, certain fault conditions relating to memory objects may be detected asynchronously by the page fault handler and reported to the task's fault handler. See *SPL Volume 4: Memory Manager* for a description of the error codes that can occur.

Chapter 15. Shared Object Functions

CPAlias

```
Syntax -

     long int CPAlias(Type,Old_Offset,&New_Offset,Size)

     unsigned long int Type;
     void *Old_Offset;
     void *New_Offset;
     unsigned long int Size;


Type      This value contains information defining the object being aliased and about the
          attributes the memory object being created.

Old_Offset
          If QMalias_all of the Type parameter is used, this is the offset of the object to
          which an alias is to be made.  If QMalias_part is used, this is the area within an
          object to which an alias is to be made.

New_Offset
          Variable in which to return the offset of the alias created.

Size      Used only if QMalias_part of the Type parameter is specified.  Indicates the size
          of the area within an object to which an alias is to be made.  The span of
          Old_Offset to (Old_Offset + (Size - 1)) must be entirely within a single memory
          object.
```

Usage Notes

This function can be used to make a temporary alias to a memory object (or part of a memory object) in another address space. The alias created is at a different offset from the original object.

The alias is owned by the caller's effective process.

The caller must be at supervisor level to make an alias to a supervisor object. If the access through the alias is less restrictive than the underlying object, then the underlying object must be owned by the caller's effective process. If the underlying object is also an alias then the original underlying object is checked to establish ownership.

If an alias is made to a common class copy_on_write object, then a physical copy of the underlying pages is made during this call.

Restriction

Aliases cannot be made to sparse memory objects that contain uncommitted pages within the area being aliased. If you make an alias to an object containing a force-not-present page, that alias is not updated to reflect any changes in the force-not-present state of any pages, changes such as the removal of any existing force-not-present pages, or the creation of new ones. Creation of new pages is effectively prevented, because aliasing forces page assignment, and thus precludes any new force-not-present pages.

The Type parameter defines the following:

QMalias_all
This is used if an alias is to be made to the entire memory object.

QMalias_part
This is used if an alias applies only to a part of the object.

QMglobal
This indicates the class of the alias (new object) is Global.

QMprivate
This indicates the class of the alias (new object) is Private.

QMold_eff
This indicates that the incoming object is to be defined in the address space of the caller's effective address space.

QMold_current
This indicates that the incoming object is to be defined in the caller's current address space.

QMnew_eff
This indicates that the new object is to be created in the caller's effective address space.

QMnew_current
This indicates that the new object is to be created in the current address space.

QMuser This indicates user privilege level is requested.

QMsupervisor
This indicates supervisor privilege level is requested.

QMwrite This indicates read/write access is requested.

QMread This is used to indicate read-only access is requested.

Implementation Notes

The unit of storage allocation is the page. Aliases can only be created in units of whole pages. This has no side effect if an entire object is aliased, because the object is page aligned and is an integral number of pages. However, if only part of an object is aliased, then more than the requested memory area is physically aliased. This means the entire page containing the starting offset of the target area and the entire page containing the last byte are physically aliased by page table manipulation. The offset returned for the alias is the offset of the first byte of the desired area. As long as the program limits its access to the memory defined by offset and size, there are no undesired side effects. Conversely, the program has access to other memory adjacent to the desired area, and if ill-behaved, this can cause unexpected side effects.

Return Codes

The Memory Manager provides a return code on all calls. Except for a success return code (QMsuccess), which has a value of 0, all Memory Manager return codes have the form **0x8002xxxx**. The possible return codes for this call, along with their low order 16 bits, are listed below.

QMsuccess	Request was successful, and the alias was created.
QMbad_offset-(0x0001)	Specified offset is not that of a valid memory object or the offset is not within a valid object.
QMbad_type-(0x0002)	Type field error - the new object has less restrictive access than original object, or the caller does not have appropriate privilege for the access level requested.
QMbad_size-(0x0004)	The size specified for an alias of part of an object extends beyond the end of the object, or it has a value of zero.
QMno_block-(0x0010)	Sufficient free virtual storage does not exist to satisfy the request.
QMno_page-(0x0012)	Real storage had to be assigned while making the alias, or additional page tables were required to map the alias, and there were insufficient free page frames available to satisfy the request.
QMnot_owned-(0x0013)	The call attempts to make a less restrictive alias, and the underlying object is not owned by the caller's effective process. If the underlying object is also an alias, the original object is subjected to an ownership test.
QMproc_limit-(0x0018)	The allocation size would exceed the process's limits.
QMwrap_count-(0x0021)	One or more physical pages underlying the area being aliased has more than the system limit (255) of aliases and (if common) shared accesses to it, or if the memory lock is at its maximum, that is, there are 255 tasks currently executing within Memory Manager code referring to this object.
QMprocess_dead-(0x0022)	The process, in whose address space the new object was to be created, is in the final stages of the Memory Manager's process removal. No new requests to create memory objects can be honored. This should not normally occur, particularly if the Resource Manager resource provider facilities have been used.
QMalias_sparse-(0x0032)	Attempting to make an alias to an area within a sparse memory object that contains uncommitted pages.
QMalias_FNP-(0x0034)	Attempting to make an alias to an area containing a Forced Not Present page.

QMbusy_fork-(0x0035)	The process containing the incoming object or the process in which the alias is to be created is performing a fork operation. The request cannot be accommodated at this time.

Memory Manager Generated Faults

It is possible that the memory manager will detect internal or system errors during the processing of this call. If this occurs, an internal error code is either returned to the caller, or is transmitted to the caller's fault handler as a result of the Memory Manager issuing a CPFaultTask call. In addition, certain fault conditions relating to memory objects may be detected asynchronously by the page fault handler and reported to the task's fault handler. See *SPL Volume 4: Memory Manager* for a description of the error codes that can occur.

CPGetMem

```
Syntax -

     long int CPGetMem(Type,Offset_in)

     unsigned long int Type;
     void *Offset_in;


Type       This value contains information about the attributes of the memory object being
           created.

Offset_in Offset of the object being retrieved.
```

Usage Notes

This function causes access to a common object to be created in a process. The object must have been previously created as common with get option. Object is created in the address space of the caller's effective process. The object appears in the caller's address space at the same offset as the creating process, that is, at Offset_in.

If the object was created in shared mode, then the caller receives shared access to the original memory object. If the object was created in copy mode, the caller receives access to a unique copy of the original object. The contents of the copy are as of the time of the call, not at the time it was created. Access attributes of the new object can be different from those of the original object, but they cannot be less restrictive than the limits defined for the original object. If the access requested is less restrictive than the limits set when the object was created, the access attributes are coerced to the limit specified at creation. In this case, the call completes successfully, but the access is the more restrictive one.

The caller must be at supervisor level to get access to a supervisor object.

If the object has already been obtained by a previous CPGetMem call, the subsequent call is processed. The object is already accessible in the address space, but the usage count is incremented. This means that the object is not be removed from the address space until as many CPFreeObj calls have been made as CPGetMem calls. If tasks use this capability, it is possible to write programs, with respect to sharing memory, that can run in the same or separate processes. In this case, the Memory Manager manages the usage and removal of the object. If there are more than one task in a process and they do not use this approach and only obtain the object once, they must communicate among themselves and ensure that the object is not removed until all the tasks using the object are finished. If multiple requests are made within a single address space, checks are made to ensure that subsequent calls do not cause more restrictive access than that which already exists.

The Type parameter defines the following:

QMtype_default
A request for the default memory type, that is, use memory type 1 if available. Otherwise, memory types 2 or 3 are used (in that order). This is only meaningful on copy mode objects.

QMtype1, QMtype2 or QMtype3
A request for a specific memory type, that is, use only the memory type that is specified. This is only meaningful on copy mode objects.

QMnew_eff
This is used to indicate that the new object is to be created in the caller's effective address space.

QMnew_current
This is used to indicate that the new object is to be created in the current address space.

QMuser This is used to indicate user privilege level is requested.

QMsupervisor
This is used to indicate supervisor privilege level is requested.

QMwrite This is used to indicate read/write access is requested.

QMread This is used to indicate read-only access is requested.

Implementation Notes

None

Return Codes

The Memory Manager provides a return code on all calls. Except for a success return code (QMsuccess), which has a value of 0, all Memory Manager return codes have the form **0x8002xxxx**. The possible return codes for this call, along with their low order 16 bits, are listed below.

QMsuccess	Request was successful, object access created.
QMbad_offset-(0x0001)	Specified offset is not that of a valid memory object. This could occur with a shared mode object if all previous users had deleted the object; therefore, it no longer exists. It could also occur with a copy mode object if the object's creator has been removed or has deleted the original object.
QMbad_type-(0x0002)	Type field error. The new object has less restrictive access than original object, or the caller does not have proper privilege for access level requested.
QMbad_handle-(0x000A)	The handle provided was not a valid handle.
QMno_page-(0x0012)	A physical copy was either requested or required, and there were insufficient free page frames available to satisfy the request, or additional page tables were required to map the object into the process.
QMproc_limit-(0x0018)	The allocation size would exceed the process's limits.
QMwrap_count-(0x0021)	The call could not be processed because the memory lock is at its maximum; that is, there are 255 tasks currently executing within Memory Manager code referring to this object.

QMprocess_dead-(0x0022)	The process, in whose address space the new object was to be created, is in the final stages of the Memory Manager's process removal. No new requests to create memory objects can be honored. This should not normally occur, particularly if the Resource Manager resource provider facilities have been used.
QMbusy_fork-(0x0035)	The receiving process is performing a fork operation. The request cannot be accommodated at this time.

Memory Manager Generated Faults

It is possible that the memory manager will detect internal or system errors during the processing of this call. If this occurs, an internal error code is either returned to the caller, or is transmitted to the caller's fault handler as a result of the Memory Manager issuing a CPFaultTask call. In addition, certain fault conditions relating to memory objects may be detected asynchronously by the page fault handler and reported to the task's fault handler. See *SPL Volume 4: Memory Manager* for a description of the error codes that can occur.

CPGiveMem

```
Syntax -

     long int CPGiveMem(Type,Offset,ProcID)

     unsigned long int Type;
     void *Offset_in;
     unsigned int ProcID;


Type       This value contains information about the attributes of the memory object being
           created.

Offset_in  Offset of the retrieved object.

ProcID     Process ID of the process to which common access is being given.
```

Usage Notes

This function causes common access to an object in the address space of another process to be established. Object must have been previously created as common with give option. Caller's effective process must be that of the object's owner. Caller does not need effective process of target process, nor does it need set effective process privilege. The target process is specified explicitly. The object appears in the caller's address space at the same offset as the creating process (that is, at Offset_in).

If the object was created in shared mode, then the target process receives shared access to the original memory object. If the object was created in copy mode, the target process receives access to a unique copy of the original object. The contents of the copy are as of the time of the call, not the time the original object was created. Access attributes of the new object can be different from those of the original object, but cannot be less restrictive than the limits defined for the original object. If the access requested is less restrictive than the limits set when the object was created, the access attributes are coerced to the limit specified at creation. In this case, the call completes successfully, but the access is the more restrictive one.

The caller must be at supervisor level to give access to a supervisor object.

If the object has already given to a process by a previous CPGiveMem call, the subsequent call is processed. The object is already accessible in the address space, but the usage count is incremented. This means that the object is not removed from the address space until as many CPFreeObj calls have been made as CPGiveMem calls (within the same address space). If tasks use this capability, it is possible to write programs, with respect to sharing memory that can run in the same or separate processes. In this case, the Memory Manager manages the usage and removal of the object. If there are more than one task in a process and they do not use this approach and only obtain the object once, then they must communicate among themselves and ensure that the object is not removed until all the tasks using the object are finished. If multiple requests are made within a single address space, checks are made to ensure that subsequent calls do not cause more restrictive access than that which already exists.

The Type parameter defines the following:

QMtype_default
A request for the default memory type, that is, use memory type 1 if available. Otherwise memory types 2 or 3 are used (in that order). This is only meaningful on copy mode objects.

QMtype1, QMtype2 or QMtype3
A request for a specific memory type (that is, use only the memory type that is specified). This is only meaningful on copy mode objects.

QMuser This is used to indicate user privilege level is requested.

QMsupervisor
This is used to indicate supervisor privilege level is requested.

QMwrite This is used to indicate read/write access is requested.

QMread This is used to indicate read-only access is requested.

Implementation Notes

None

Return Codes

The Memory Manager provides a return code on all calls. Except for a success return code (QMsuccess), which has a value of 0, all Memory Manager return codes have the form **0x8002xxxx**. The possible return codes for this call, along with their low order 16 bits, are listed below.

QMsuccess	Request was successful, object access created in requested address space.
QMbad_offset-(0x0001)	Specified offset is not that of a valid memory object. This can occur if the original object has been removed, even if others are still sharing it or using a copy of it.
QMbad_type-(0x0002)	Type field error. The new object has less restrictive access than limits allowed on original object when created, or the caller does not have proper privilege for access level requested.
QMbad_PID-(0x0003)	Process ID is not that of a valid process.
QMbad_handle-(0x000A)	The handle provided was not a valid handle.
QMno_page-(0x0012)	A physical copy was either requested or required, and there were insufficient free page frames available to satisfy the request, or additional page tables were required to map the object into the process.
QMnot_owned-(0x0013)	Caller's effective process not the same process owning original object.
QMproc_limit-(0x0018)	The allocation size would exceed the process's limits.
QMwrap_count-(0x0021)	The call could not be processed because the memory lock is at its maximum; that is, there are 255 tasks currently executing within Memory Manager code referring to this object.

QMprocess_dead-(0x0022)	The process, in whose address space the new object was to be created, is in the final stages of the Memory Manager's process removal. No new requests to create memory objects can be honored. This should not normally occur, particularly if the Resource Manager resource provider facilities have been used.
QMbusy_fork-(0x0035)	The target process is performing a fork operation. The request cannot be accommodated at this time.

Memory Manager Generated Faults

It is possible that the memory manager will detect internal or system errors during the processing of this call. If this occurs, an internal error code is either returned to the caller, or is transmitted to the caller's fault handler as a result of the Memory Manager issuing a CPFaultTask call. In addition, certain fault conditions relating to memory objects may be detected asynchronously by the page fault handler and reported to the task's fault handler. See *SPL Volume 4: Memory Manager* for a description of the error codes that can occur.

Chapter 16. Change Object Functions

CPChangeAttr

```
Syntax -

     long int CPChangeAttr(Type,Offset)

     unsigned long int Type;
     void *Offset;

Type      This value contains information about the attributes of the memory object being
          changed.

Offset    Offset of the object whose attributes are to be changed.
```

Usage Notes

This function alters the attributes of a memory object. Changes which increase access, that is which are less restrictive (such as changing from supervisor to user), can only be made if the object is owned by the caller's effective process, or (in the case of global object or shared mode common object) if the object is one created by the caller's effective process. If the object is an alias, then the original underlying object is checked for ownership. Furthermore, changes which reduce access (such as changing from writeable to read-only) can only be made if the fix count, verify counts, and memory lock are all zero. The implied fix for adapter memory obtained from a CPAllocBase or CPAllocRange call does not prevent the changing of attributes.

The caller must have supervisor privilege to access a supervisor object.

The Type parameter defines the following:

QMold_eff
This is used to indicate that the incoming object is to be defined in the address space of the caller's effective address space.

QMold_current
This is used to indicate that the incoming object is to be defined in the caller's current address space.

QMuser This is used to indicate user privilege level is requested.

QMsupervisor
This is used to indicate supervisor privilege level is requested.

QMwrite This is used to indicate read/write access is requested.

QMread This is used to indicate read-only access is requested.

Implementation Notes

For shared mode objects, the type parameters **QMuser** and **QMsupervisor** are only used for access validation purposes. It is not possible to have a shared common object with user level access in one address space and supervisor level access in another due to the &PPCPWRA.. If after validation access is granted, the access matches that of the original object.

Return Codes

The Memory Manager provides a return code on all calls. Except for a success return code (QMsuccess), which has a value of 0, all Memory Manager return codes have the form **0x8002xxxx**. The possible return codes for this call, along with their low order 16 bits, are listed below.

QMsuccess	Request was successful, object's attributes have been altered.
QMbad_offset-(0x0001)	Specified offset is not that of a valid memory object.
QMbad_type-(0x0002)	Type field error. The new object has less restrictive access than original object and object not owned by caller, or the caller has insufficient privilege to access object.
QMnot_owned-(0x0013)	Object is not owned by the caller's effective process.
QMobj_fixed-(0x001A)	Specified object had nonzero fix count.
QMobj_accessed-(0x001B)	Object has outstanding verifies or memory locks (attributes not changed).
QMwrap_count-(0x0021)	The call could not be processed because the memory lock is at its maximum; that is, there are 255 tasks currently executing within Memory Manager code referring to this object.
QMbusy_fork-(0x0035)	The process, in which the referenced object exists, is performing a fork operation. The request cannot be accommodated at this time.

Memory Manager Generated Faults

It is possible that the memory manager will detect internal or system errors during the processing of this call. If this occurs, an internal error code is either returned to the caller, or is transmitted to the caller's fault handler as a result of the Memory Manager issuing a CPFaultTask call. In addition, certain fault conditions relating to memory objects may be detected asynchronously by the page fault handler and reported to the task's fault handler. See *SPL Volume 4: Memory Manager* for a description of the error codes that can occur.

CPShrinkSize

```
Syntax -

     long int CPShrinkSize(Type,Offset,Size)

     unsigned long int Type;
     void *Offset;
     unsigned long int Size;


Type      This value contains information about the memory object being referenced.

Offset    The offset of the memory object whose size is to be reduced.

Size      The object's new size (in bytes).  This is rounded up to a multiple of the page
          size.
```

Usage Notes

This function can be used to shrink the size of a memory object.

The caller must be at supervisor level to access a supervisor object. The object must be owned in the caller's effective process.

The Type parameter defines the following:

QMold_eff
This is used to indicate that the incoming object is to be defined in the address space of the caller's effective address space.

QMold_current
This is used to indicate that the incoming object is to be defined in the caller's current address space.

Implementation Notes

None

Return Codes

The Memory Manager provides a return code on all calls. Except for a success return code (QMsuccess), which has a value of 0, all Memory Manager return codes have the form **0x8002xxxx**. The possible return codes for this call, along with their low order 16 bits, are listed below.

QMsuccess	Request was successful, and the size changed.
QMbad_offset-(0x0001)	Specified offset is not that of a valid memory object.
QMbad_type-(0x0002)	Type field error. The caller does not have proper privilege to access the object.
QMbad_size-(0x0004)	The size specified was greater than the current object size, or a size of zero was specified.
QMnot_owned-(0x0013)	Object is not owned by the caller's effective process.
QMobj_fixed-(0x001A)	Specified object had nonzero fix count.

QMobj_accessed-(0x001B)	Specified object had nonzero verify count or outstanding aliases, and the size was not changed.
QMbusy_fork-(0x0035)	The process, in which the referenced object exists, is doing a fork operation. The request cannot be accommodated at this time.

Memory Manager Generated Faults

It is possible that the memory manager will detect internal or system errors during the processing of this call. If this occurs, an internal error code is either returned to the caller, or is transmitted to the caller's fault handler as a result of the Memory Manager issuing a CPFaultTask call. In addition, certain fault conditions relating to memory objects may be detected asynchronously by the page fault handler and reported to the task's fault handler. See *SPL Volume 4: Memory Manager* for a description of the error codes that can occur.

Chapter 17. Sparse Object Functions

CPCommit

```
Syntax -

     long int CPCommit(Type,Offset,Size,OffsetPage)

     unsigned long int Type;
     void *Offset;
     unsigned long int Size;
     void *OffsetPage;


Type      This value contains information about the attributes of the memory object being
          referenced.

Offset    Offset of the sparse object in which pages are to be committed.

Size      Number of bytes to be committed at offset specified by OffsetPage.  This must be a
          page size (4K) multiple.

OffsetPage
          Offset of the page (must be within object defined by Offset) where the pages are
          to be committed.
```

Usage Notes

This function commits one or more real storage pages within a sparse object. Offset is the address of the beginning of the sparse object and identifies the object. OffsetPage is the address of a page that is part of the object, and it identifies the first page to be committed. If Size is 4096, one page is committed. If Size is 12288, three pages are committed, providing that all three pages are within the sparse object.

The Type parameter defines the following:

QMold_eff
: This is used to indicate that the incoming object is to be defined in the address space of the caller's effective address space.

QMold_current
: This is used to indicate that the incoming object is to be defined in the caller's current address space.

Implementation Notes

None

Return Codes

The Memory Manager provides a return code on all calls. Except for a success return code (QMsuccess), which has a value of 0, all Memory Manager return codes have the form **0x8002xxxx**. The possible return codes for this call, along with their low order 16 bits, are listed below.

QMsuccess	Request was successful, a page or pages have been committed.
QMbad_offset-(0x0001)	Specified offset is not that of a valid memory object.
QMbad_type-(0x0002)	Type field error. The object is not sparse, or the caller does not have privilege level required to access object.
QMbad_size-(0x0004)	Specified size extends outside of specified memory object or was zero.
QMbad_location-(0x0005)	Specified page offset is not within specified memory object, or it is not on a page boundary.
QMno_page-(0x0012)	A real storage page frame of the desired type was not available for assignment to satisfy the request.
QMwrap_count-(0x0021)	The call could not be processed because the memory lock is at its maximum; that is, there are 255 tasks currently executing within Memory Manager code referring to this object.
QMprocess_dead-(0x0022)	The process, in whose address space the object was defined, is in the final stages of the Memory Manager's process removal. No new requests to modify memory objects can be honored. This should not normally occur, particularly if the Resource Manager resource provider facilities have been used.
QMuser_commit_error-(0x002C)	Specified address range includes previously committed pages.
QMbusy_fork-(0x0035)	The process, in which the sparse object exists, is performing a fork operation. The request cannot be accommodated at this time.
QMcommit_FNP-(0x0037)	The page or pages being committed contain a Force Not Present page.

Memory Manager Generated Faults

It is possible that the memory manager will detect internal or system errors during the processing of this call. If this occurs, an internal error code is either returned to the caller, or is transmitted to the caller's fault handler as a result of the Memory Manager issuing a CPFaultTask call. In addition, certain fault conditions relating to memory objects may be detected asynchronously by the page fault handler and reported to the task's fault handler. See *SPL Volume 4: Memory Manager* for a description of the error codes that can occur.

CPDecommit

```
Syntax -

     long int CPDecommit(Type,Offset,Size,OffsetPage)

     unsigned long int Type;
     void *Offset;
     unsigned long int Size;
     void *OffsetPage;


Type      This value contains information about the attributes of the memory object being
          referenced.

Offset    Offset of the sparse object in which pages are to be decommitted.

Size      Number of bytes to be decommitted at offset specified by OffsetPage.  This must be
          an integral number pages, that is a multiple of 4K bytes.

OffsetPage
          Offset of the page (must be within object defined by Offset) where the page or
          pages are to be decommitted.
```

Usage Notes

This function decommits one or more real storage pages within a sparse object. Offset is the address of the beginning of the sparse object and identifies the object. OffsetPage is the address of a page that is part of the object and identifies the first page to be decommitted. If Size is 4096, one page is decommitted. If Size is 12288, three pages are decommitted, providing that all three pages are within the sparse object.

The Type parameter defines the following:

QMold_eff
This is used to indicate that the incoming object is to be defined in the address space of the caller's effective address space.

QMold_current
This is used to indicate that the incoming object is to be defined in the caller's current address space.

Implementation Notes

None

Return Codes

The Memory Manager provides a return code on all calls. Except for a success return code (QMsuccess), which has a value of 0, all Memory Manager return codes have the form **0x8002xxxx**. The possible return codes for this call, along with their low order 16 bits, are listed below.

QMsuccess	Request was successful, a page or pages have been decommitted.
QMbad_offset-(0x0001)	Specified offset is not that of a valid memory object.

QMbad_type-(0x0002)	Type field error. The object is not sparse, or the caller does not have privilege level required to access object.
QMbad_location-(0x0005)	Specified page offset is not within specified memory object or not on a page boundary.
QMbad_size-(0x0004)	Specified size extends outside of specified memory object or was zero.
QMwrap_count-(0x0021)	The call could not be processed because the memory lock is at its maximum; that is, there are 255 tasks currently executing within Memory Manager code referring to this object.
QMprocess_dead-(0x0022)	The process, in whose address space the object was defined, is in the final stages of the Memory Manager's process removal. No new requests to modify memory objects can be honored. This should not normally occur, particularly if the Resource Manager resource provider facilities have been used.
QMuser_commit_error-(0x002C)	Specified address range contains a page or pages not previously committed.
QMbusy_fork-(0x0035)	The process, in which the sparse object exists, is performing a fork operation. The request cannot be accommodated at this time.

Memory Manager Generated Faults

It is possible that the memory manager will detect internal or system errors during the processing of this call. If this occurs, an internal error code is either returned to the caller, or is transmitted to the caller's fault handler as a result of the Memory Manager issuing a CPFaultTask call. In addition, certain fault conditions relating to memory objects may be detected asynchronously by the page fault handler and reported to the task's fault handler. See *SPL Volume 4: Memory Manager* for a description of the error codes that can occur.

Chapter 18. Memory Verification Functions

CPFreeVerify

```
Syntax -

     long int CPFreeVerify(Type,Offset)

     unsigned long int Type;
     void *Offset;

Type      This value determines the address space of the incoming object.

Offset    Offset of the object or the start of an area within an object to have a verify
          released.
```

Usage Notes

This function is used to reverse the effect of a previous call to CPVerify. The function can be used from either privilege level. A verify count is maintained for supervisor level. The counter is decremented for supervisor level callers. It is treated as an information only call for user level callers.

It is strongly recommended that the CPFreeVerify call be made from the same piece of code that made the CPVerify call. If the object has been marked “remove as soon as possible” and the conditions for removal are now satisfied, a message is sent to the memory task to complete the removal.

The Type parameter is used to define the incoming object's address space, as follows:

QMold_eff
This is used to indicate that the incoming object is to be defined in the address space of the caller's effective address space.

QMold_current
This is used to indicate that the incoming object is to be defined in the caller's current address space.

Implementation Notes

None

Return Codes

The Memory Manager provides a return code on all calls. Except for a success return code (QMsuccess), which has a value of 0, all Memory Manager return codes have the form **0x8002xxxx**. The possible return codes for this call, along with their low order 16 bits, are listed below.

QMsuccess	Request was successful and verify count has been decremented.

QMaccess_info-(0x001D)	Requestor does not have supervisor privilege. Verify count has not been decremented.
QMbad_offset-(0x0001)	Specified offset is not within a valid memory object.
QMbad_type-(0x0002)	Type field error. The caller is trying to verify supervisor level object without supervisor privilege.
QMuser_verify_error-(0x002A)	Verify counter was zero at the time of call from supervisor level caller. This indicates one of four possibilities: • A CPFreeVerify without a matching prior CPVerify • CPVerify and CPFreeVerify calls made at differing privilege levels • A bad offset (not the one verified) • An incorrect address space
QMwrap_count-(0x0021)	The call could not be processed because the memory lock is at its maximum; that is, there are 255 tasks currently executing within Memory Manager code referring to this object.

Memory Manager Generated Faults

It is possible that the memory manager will detect internal or system errors during the processing of this call. If this occurs, an internal error code is either returned to the caller, or is transmitted to the caller's fault handler as a result of the Memory Manager issuing a CPFaultTask call. In addition, certain fault conditions relating to memory objects may be detected asynchronously by the page fault handler and reported to the task's fault handler. See *SPL Volume 4: Memory Manager* for a description of the error codes that can occur.

CPVerify

```
Syntax -

     long int CPVerify(Type,Offset,Size)

     unsigned long int Type;
     void *Offset;
     unsigned long int Size;


Type      This value determines the address space of the incoming object and whether read or
          read/write access is to be checked.

Offset    Offset of the object or the start of an area within an object to be verified.

Size      This determines the size in bytes.  The area from Offset to (Offset + (Size - 1))
          is verified and must occur within a single memory object.
```

Usage Notes

This function is used to verify that the caller has read or read/write access to a defined portion of a memory object. This call can be used to verify an entire object or a portion of an object. If it refers to a part of an object, it must be wholly contained in a single object. The function can be used from either privilege level. A verify count in the range 0 through 255 is reserved for supervisor level. The counter is incremented for supervisor level callers.

As a result of this call, the verify count the incoming object is incremented only if the caller has supervisor privilege. Requests which would alter the size or character of a object are then disallowed, thus guaranteeing the requested access until the verify count is decremented. The decrement should be performed by the same piece of code that issued the increment, so that the correct verify count is updated. Callers with user privilege should regard this as information-only as to the current status of the object at the time of the call.

The Type parameter is used to define the incoming object's address space and whether read or read/write access is to be verified. The Type field of the incoming object is used to define the privilege level at which the read or read/write access is to be checked.

The Type parameter defines the following:

QMold_eff
: This is used to indicate that the incoming object is to be defined in the address space of the caller's effective address space.

QMold_current
: This is used to indicate that the incoming object is to be defined in the caller's current address space.

QMuser This indicates the privilege level to be used in testing callers access to the area being verified. This is used to indicate user privilege level, and it should be set to privilege of process triggering this request.

QMsupervisor	This indicates the privilege level to be used in testing callers access to the area being verified. This is used to indicate supervisor privilege level, and it should be set to privilege of process triggering this request.
QMwrite	This is used to indicate read/write access should be verified.
QMread	This is used to indicate read-only access should be verified.

Implementation Notes

None

Return Codes

The Memory Manager provides a return code on all calls. Except for a success return code (QMsuccess), which has a value of 0, all Memory Manager return codes have the form **0x8002xxxx**. The possible return codes for this call, along with their low order 16 bits, are listed below.

QMsuccess	Request was successful, and the caller has required access and verify count has been incremented.
QMbad_offset-(0x0001)	Specified offset is not within a valid memory object.
QMbad_type-(0x0002)	Type field error. The caller is trying to verify supervisor level object without supervisor privilege.
QMbad_size-(0x0004)	Specified size not contained entirely within a single memory object or was zero.
QMno_access-(0x0011)	Access verification requested, and the caller does not have requested access.
QMaccess_info-(0x001D)	Requestor has required access but does not have supervisor privilege. Verify count has not been incremented.
QMwrap_count-(0x0021)	The verify count is at its maximum, or the call could not be processed because the memory lock is at its maximum; that is, there are 255 tasks currently executing within Memory Manager code referring to this object.

Memory Manager Generated Faults

It is possible that the memory manager will detect internal or system errors during the processing of this call. If this occurs, an internal error code is either returned to the caller, or is transmitted to the caller's fault handler as a result of the Memory Manager issuing a CPFaultTask call. In addition, certain fault conditions relating to memory objects may be detected asynchronously by the page fault handler and reported to the task's fault handler. See *SPL Volume 4: Memory Manager* for a description of the error codes that can occur.

Chapter 19. Memory Fixing Functions

CPFix

```
Syntax -

     long int CPFix(Type,Offset,Size,Fix_List,&Fix_Handle)

     unsigned long int Type;
     void *Offset;
     unsigned long int Size;
     void *Fix_List;
     unsigned long int Fix_Handle;
```

Type	This value determines the address space of the incoming object and whether read or read/write access is to be checked.
Offset	Offset of the start of the area to be fixed. It must be within a valid memory object.
Size	Size of the area to be fixed. The entire area to be fixed must occur within a single valid memory object.
Fix_List	The caller must have write access to sufficient storage at this address to store the maximum size list possible for the area the caller wants fixed. The size in double words of the list can be computed from the following formula using integer arithmetic: N = 1 + 2 * (2 + (Size / 4096)) where Size is the size of the portion of the object to be fixed.
Fix_Handle	Value returned by this call to function CPFix, which is later specified in the corresponding call to function CPUnfix to release this fix.

Usage Notes

This function is used to fix a defined portion of a object in real storage. Optionally, it can verify that the caller has read or read/write access to that portion of the object. The caller must have I/O privilege. Furthermore, it is recommended that the privilege level of the incoming objects's as specified in the Type parameter be adjusted by the caller to reflect the privilege of the code which triggered the request.

As a result of this call, the caller's access is verified if requested, and the fix count for the incoming object is incremented, and the real addresses of the specified portion of the object are returned. Other requests which would alter the size or character of an object or change its physical address are then disallowed. This guarantees that the requested access and physical address remain valid until the fix count is decremented.

Because the pages underlying the area to be fixed are not contiguous, a list of real storage addresses is returned. The caller must ensure that adequate space is provided to contain the largest possible list of addresses for the request.

Each fix request must be unfixed by a matching request for each portion of the object that was fixed. It is not possible to make a series of fix requests for adjacent/overlapping storage areas and to remove the fix with a single unfix request. A fix handle is returned for each call, and defines a “pseudo object” which is used to manage the fix. This handle must be specified in the call that releases the fix. A maximum of 255 fix requests can be outstanding, at any one time, against a particular object.

The Type parameter is used to define the incoming object's address space and whether read or read/write access is being queried.

If the object was previously allocated such that real storage page frames are only assigned on use of the object (which is the default), then real storage page frames are assigned as necessary to ensure that all of the specified range is allocated, and these pages are then fixed.

This call can be viewed as having two distinct parts. The first is optional and verifies that the caller has access. In this part, the offset and size are used to determine if the caller has access to the area within the object as defined by Offset and Size. The second part fixes only the area defined by Offset and Size. A separate call is available to fix an entire object.

The Type parameter defines the following:

QMreverse_fixlist
This is used to indicate that the alternate form of the fix list is requested.

QMold_eff
This is used to indicate that the incoming object is to be defined in the address space of the caller's effective address space.

QMold_current
This is used to indicate that the incoming object is to be defined in the caller's current address space.

QMlong_fix
This is used to indicate a long term fix on CPFix calls.

QMshort_fix
This is used to indicate a short term fix on CPFix calls.

QMverify_fix
This is used to indicate that a verify or free verify operation is to be performed in conjunction with a CPFix or CPUnfix call.

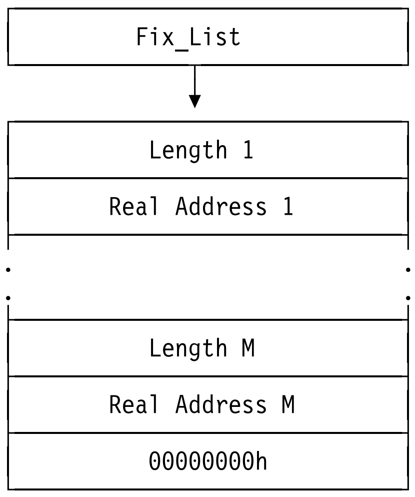
QMuser This is used to determine privilege level to be used in testing callers access to the area being fixed. This indicates user level and should be set to the privilege of process triggering this request.

QMsupervisor
This is used to determine the privilege level to be used in testing callers access to the area being fixed. This indicates supervisor level and should be set to the privilege of process triggering this request.

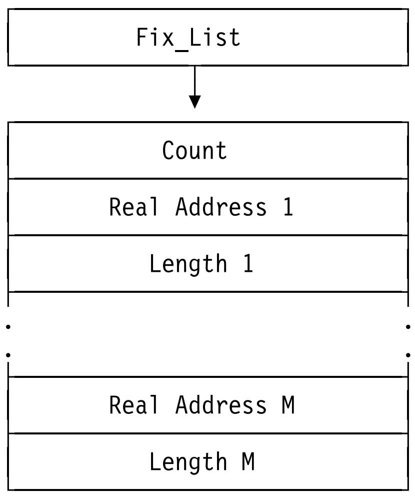
QMwrite Valid if verification is requested and determines type of access to be verified. This indicates write access.

QMread Valid if verification is requested and determines type of access to be verified. This indicates read-only access.

Upon return, the Fix_Handle to be used on any subsequent Unfix call has been provided, and the real storage addresses of the fixed area are placed in the fix list pointed to by Fix_List. There are two forms of the address list. The standard format of the address list is:



The alternate format of the address list is as follows:



Real Address 1 is the real address of the byte defined by Offset (in the address space determined from the Type parameter). Length 1 is the number of bytes beginning at Real Address 1. The list continues until the number of bytes specified in Size has been reached. For example, Size = Length 1 + ... + Length M. In the standard format, the address list is terminated by a length of 00000000h. In the alternate format, the list begins with a count of the number of pairs of address and length that are utilized in the list.

Note: The number of entries in the address list (M) does not necessarily equal the maximum possible size of the list (N). The actual number of entries used depends on the alignment of the linear addresses underlying the area and the contiguity of the real addresses of the page frames underlying the area.

Implementation Notes

If the area to be fixed contains copy on write memory, a unique copy is created at the time the CPFix call is processed. This ensures that when a subsequent I/O is performed all the pages are in memory and fixed.

Return Codes

The Memory Manager provides a return code on all calls. Except for a success return code (QMsuccess), which has a value of 0, all Memory Manager return codes have the form **0x8002xxxx**. The possible return codes for this call, along with their low order 16 bits, are listed below.

QMsuccess	Request was successful, access verified if requested, and area fixed.
QMbad_offset-(0x0001)	Specified offset is not within a valid memory object.
QMbad_type-(0x0002)	Type field error. The caller is trying to fix supervisor level object without supervisor privilege.
QMbad_size-(0x0004)	Specified size is not contained entirely within a single memory object, or the size was specified as zero.
QMbad_fixlist-(0x0009)	Caller did not have access to a fixlist of sufficient size.
QMno_access-(0x0011)	Access verification requested and caller does not have requested access.
QMno_page-(0x0012)	A real storage page frame of the desired type was not available for assignment to satisfy the request.
QMwrap_count-(0x0021)	The maximum number of fixes is already outstanding against this object, or a verify was requested, and the maximum number of verifies is already outstanding, or the memory lock is at its maximum; that is, there are 255 tasks currently executing within Memory Manager code referring to this object.
QMfix_sparse-(0x0028)	The area being fixed, within a sparse object, was not entirely committed.
QMrestricted_function-(0x0030)	Caller did not have I/O privilege.
QMfix_FNP-(0x0032)	The area being fixed contains a Force Not Present page.
QMbusy_fork-(0x0035)	The process, in which the area to be fixed exists, is performing a fork operation. The request cannot be accommodated at this time.

Memory Manager Generated Faults

It is possible that the memory manager will detect internal or system errors during the processing of this call. If this occurs, an internal error code is either returned to the caller, or is transmitted to the caller's fault handler as a result of the Memory Manager issuing a CPFaultTask call. In addition, certain fault conditions relating to memory objects may be detected asynchronously by the page fault handler and reported to the task's fault handler. See *SPL Volume 4: Memory Manager* for a description of the error codes that can occur.

CPFixObj

```
Syntax -

     long int CPFixObj(Type,Offset,Fix_List)

     unsigned long int Type;
     void *Offset;
     void *Fix_List;


Type      This value determines the address space of the incoming object and whether read or
          read/write access is to be checked.

Offset    Offset of the object to be fixed.

Fix_List  The caller must have write access to sufficient storage at this address to store
          the maximum size list possible for the area the caller wants fixed.  The size in
          double words of the list can be computed from the following formula using integer
          arithmetic:
          N = 1 + 2 * (2 + (Size / 4096))
          where Size is the size of the portion of the object to be fixed.
```

Usage Notes

This function is used to fix an entire memory object in real storage. Optionally, it can verify that the caller has read or read/write access to that object. The caller must have I/O privilege. Furthermore, it is recommended that the privilege level of the incoming object's as specified in the Type parameter be adjusted by the caller to reflect the privilege of the code which triggered the request.

As a result of this call, the caller's access is verified if requested, and the fix count for the incoming object is incremented and the real addresses of the object are returned. Other requests which would alter the size or character of an object, or change its physical address, are then disallowed. This guarantees that the requested access and physical address remain valid until the fix count is decremented.

As the pages underlying the area to be fixed are not contiguous, a list of real storage addresses is returned. The caller must ensure that adequate space is provided to contain the largest possible list of addresses for the request.

Each fix request must be unfixed by a matching request for object that was fixed. A maximum of 255 fix requests can be outstanding at any one time against a particular object.

If the object was previously allocated such that real storage page frames are only assigned on use of the object (the default), then real storage page frames are assigned and fixed as necessary.

This call can be viewed as having two distinct parts. The first (optional) verifies that the caller has access. The second part fixes the object. A separate call is available to fix part of an object.

The Type parameter is used to define the incoming object's address space and whether read or read/write access is being verified, as follows:

QMreverse_fixlist
This is used to indicate that the alternate form of the fix list is requested.

QMold_eff
This is used to indicate that the incoming object is to be defined in the address space of the caller's effective address space.

QMold_current
This is used to indicate that the incoming object is to be defined in the caller's current address space.

QMlong_fix
This is used to indicate a long term fix on CPFixObj calls.

QMshort_fix
This is used to indicate a short term fix on CPFixObj calls.

QMverify_fix
This is used to indicate that a verify or free verify operation is to be performed in conjunction with a CPFixObj or CPUnfixObj call.

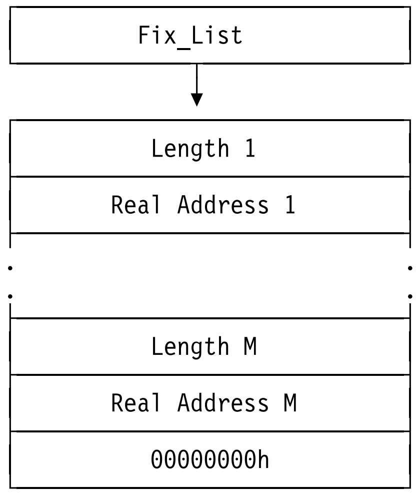
QMuser This is used to determine privilege level to be used in testing callers access to the area being fixed. This indicates user level and should be set to the privilege of process triggering this request.

QMsupervisor
This is used to determine privilege level to be used in testing callers access to the area being fixed. This indicates supervisor level and should be set to the privilege of process triggering this request.

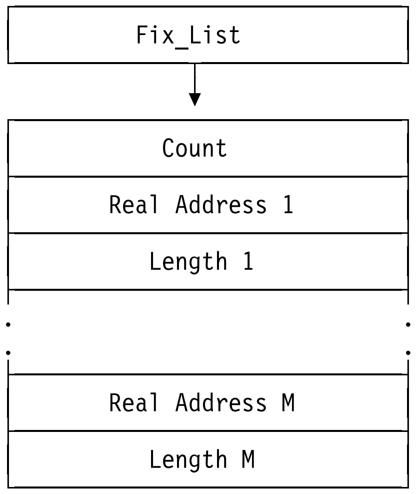
QMwrite Valid if verification is requested and determines type of access to be verified. This indicates write access.

QMread Valid if verification is requested and determines type of access to be verified. This indicates read-only access.

On return the real storage addresses of the fixed area are placed in the fix list pointed to by Fix_List. There are two forms of the address list. The standard format of the address list is:



The alternate format of the address list is as follows:



Note: The length fields in the fix list are unnecessary, because objects are page multiples on page boundaries. Therefore, the lengths are 4K. This format is used for compatibility with the format of the fix list for the other function CPFix.

Real Address 1 is the real address of the byte defined by Offset (in the address space determined from the Type parameter). Length 1 is the number of bytes beginning at Real Address 1. The list continues until the number of bytes specified in Size has been reached. This means Size = Length 1 + ... + Length M. In the standard format, the address list is terminated by a length of 00000000h. In the alternate format, the list begins with a count of the number of pairs of address and length that are utilized in the list.

Note: The number of entries in the address list (M) does not necessarily equal the maximum possible size of the list (N). The actual number of entries used depends on the alignment of the linear addresses underlying the area and on the contiguity of the real addresses of the page frames underlying the area.

Implementation Notes

If the object to be fixed contains copy-on-write memory, a unique copy is created at the time the CPFix call is processed. This is to ensure that when subsequent I/O is performed all the pages are in memory and fixed.

Return Codes

The Memory Manager provides a return code on all calls. Except for a success return code (QMsuccess), which has a value of 0, all Memory Manager return codes have the form **0x8002xxxx**. The possible return codes for this call, along with their low order 16 bits, are listed below.

QMsuccess	Request was successful, access verified if requested, and the object is fixed.
QMbad_offset-(0x0001)	Specified offset is not that of a valid memory object.
QMbad_type-(0x0002)	Type field error. The caller is trying to fix supervisor level object without supervisor privilege.
QMbad_fixlist-(0x0009)	Caller did not have access to a fixlist of sufficient size.

QMno_access-(0x0011)	Access verification requested, and the caller does not have requested access.
QMno_page-(0x0012)	A real storage page frame of the desired type was not available for assignment to satisfy the request.
QMwrap_count-(0x0021)	The maximum number of fixes is already outstanding against this object, or a verify was requested, and the maximum number of verifies is already outstanding, or the call could not be processed because the memory lock is at its maximum; that is, there are 255 tasks currently executing within Memory Manager code referring to this object.
QMfix_sparse-(0x0028)	A sparse object being fixed was not entirely committed.
QMrestricted_function-(0x0030)	Caller does not have the privilege level required for this request.
QMfix_FNP-(0x0032)	The area being fixed contains a Force Not Present page.
QMbusy_fork-(0x0035)	The process, in which the object to be fixed exists, is performing a fork operation. The request cannot be accommodated at this time.

Memory Manager Generated Faults

It is possible that the memory manager will detect internal or system errors during the processing of this call. If this occurs, an internal error code is either returned to the caller, or is transmitted to the caller's fault handler as a result of the Memory Manager issuing a CPFaultTask call. In addition, certain fault conditions relating to memory objects may be detected asynchronously by the page fault handler and reported to the task's fault handler. See *SPL Volume 4: Memory Manager* for a description of the error codes that can occur.

CPUnfix

```
Syntax -

     long int CPUnfix(Type,Fix_Handle)

     unsigned long int Type;
     unsigned long int Fix_Handle;


Type      This value determines the address space of the incoming object and whether the
          verify count is also to be decremented.

Fix_Handle
          The Fix_Handle that was returned by a previous call to CPFix,which represent the
          defined portion of a memory object whose fix is now to be released.
```

Usage Notes

This function is used to release a fix on the real storage page frames underlying a defined portion of a memory object. This means it reverses the effect of a previous call to CPFix. The caller requires I/O privilege.

Each CPUnfix request must match a previous call to CPFix, and it must use the value of Fix_Handle that was returned by that call to CPFix. As a result of this call, the fix count for the incoming object is decremented. Optionally, the verify count of the object might also be decremented by this call.

The Type parameter is used to define the incoming object's address space and whether the verify count is to be decremented, as follows:

QMold_eff
This is used to indicate that the incoming object is to be defined in the address space of the caller's effective address space.

QMold_current
This is used to indicate that the incoming object is to be defined in the caller's current address space.

QMverify_fix
This is used to indicate that a free verify operation is to be performed in conjunction with the CPUnfix call.

Implementation Notes

None

Return Codes

The Memory Manager provides a return code on all calls. Except for a success return code (QMsuccess), which has a value of 0, all Memory Manager return codes have the form **0x8002xxxx**. The possible return codes for this call, along with their low order 16 bits, are listed below.

QMsuccess	Request was successful, fix released.

QMbad_offset-(0x0001)	The specified fix_handle was interpreted as an offset to an entire memory object and was found to be either an invalid memory object or a memory object not previously fixed. The caller should specify a fix_handle that was returned by a previous call to CPFix.
QMbad_type-(0x0002)	Type field error. The caller is trying to unfix supervisor level object without supervisor privilege.
QMbad_fixhandle-(0x000A)	The specified fix_handle is not that of a valid memory object, or it did not refer to an area within an object that was previously fixed.
QMwrap_count-(0x0021)	The call could not be processed because the memory lock is at its maximum; that is, there are 255 tasks currently executing within Memory Manager code referring to this object.
QMunfixNP-(0x0027)	A not present page was found during unfixing. This could indicate unmatched fix/unfix requests.
QMuser_fix_error-(0x0029)	Fix count was zero when this request was received. This indicates an unfix without a matching prior fix or incorrect address space. It is possible that the error was caused by an unfix done by another task, not necessarily of higher priority than the caller. This is an application error.
QMuser_verify_error-(0x002A)	Verify count was zero when this request was received. This indicates a free verify without a matching prior verify, or incorrect address space. It is possible that the error was caused by a free verify done by another task, not necessarily of higher priority than the caller. This is an application error. This can only occur if QMverify_fix was specified in the Type field.
QMrestricted_function-(0x0030)	Caller does not have the privilege level required for this request.
QMbusy_fork-(0x0035)	The process, in which the area to be unfixed exists, is performing a fork operation. The request cannot be accommodated at this time.

Memory Manager Generated Faults

It is possible that the memory manager will detect internal or system errors during the processing of this call. If this occurs, an internal error code is either returned to the caller, or is transmitted to the caller's fault handler as a result of the Memory Manager issuing a CPFaultTask call. In addition, certain fault conditions relating to memory objects may be detected asynchronously by the page fault handler and reported to the task's fault handler. See *SPL Volume 4: Memory Manager* for a description of the error codes that can occur.

CPUnfixObj

```
Syntax -

     long int CPUnfixObj(Type,Offset)

     unsigned long int Type;
     void *Offset;


Type      This value determines the address space of the incoming object and whether the
          verify count is also to be decremented.

Offset    Offset of the entire memory object whose fix is to be released.
```

Usage Notes

This function is used to release a fix on the real storage page frames underlying an entire memory object. This means it reverses the effect of a previous call to CPFixObj. The caller requires I/O privilege.

Each CPUnfixObj request must match a previous call to CPFixObj and must use the value of Offset that was used in that call to CPFixObj. As a result of this call, the fix count for the incoming object is decremented. Optionally, the verify count of the object may also be decremented by this call.

The Type parameter is used to define the incoming object's address space and whether the verify count is to be decremented, as follows:

QMold_eff
: This is used to indicate that the incoming object is to be defined in the address space of the caller's effective address space.

QMold_current
: This is used to indicate that the incoming object is to be defined in the caller's current address space.

QMverify_fix
: This is used to indicate that a free verify operation is to be performed in conjunction with the CPUnfixObj call.

Implementation Notes

None

Return Codes

The Memory Manager provides a return code on all calls. Except for a success return code (QMsuccess), which has a value of 0, all Memory Manager return codes have the form **0x8002xxxx**. The possible return codes for this call, along with their low order 16 bits, are listed below.

QMsuccess	Request was successful, fix released.
QMbad_offset-(0x0001)	Specified offset is not that of a valid memory object, or it did not refer to an object that was previously fixed.

QMbad_type-(0x0002)	Type field error. The caller is trying to unfix supervisor level object without supervisor privilege.
QMbad_fixhandle-(0x000A)	The specified offset was found to be a fix_handle representing a portion of a memory object and was found to refer either to an invalid memory object or to an area within an object that was not previously fixed. The caller should specify an offset that was used in a previous call to CPFixObj.
QMwrap_count-(0x0021)	The call could not be processed because the memory lock is at its maximum; that is, there are 255 tasks currently executing within Memory Manager code referring to this object.
QMunfixNP-(0x0027)	A not present page was found during unfixing. This could indicate unmatched fix/unfix requests.
QMuser_fix_error-(0x0029)	Fix count was zero when this request was received. This indicates an unfix without a matching prior fix or incorrect address space. It is possible that the error was caused by an unfix done by another task, not necessarily of higher priority than the caller. This is an application error.
QMuser_verify_error-(0x002A)	Verify count was zero when this request was received. This indicates a free verify without a matching prior verify or incorrect address space. It is possible that the error was caused by a free verify done by another task, not necessarily of higher priority than the caller. This is an application error. This can only occur if QMverify_fix was specified in the Type field.
QMrestricted_function-(0x0030)	Caller does not have the privilege level required for this request.
QMbusy_fork-(0x0035)	The process, in which the object to be unfixed exists, is performing a fork operation. The request cannot be accommodated at this time.

Memory Manager Generated Faults

It is possible that the memory manager will detect internal or system errors during the processing of this call. If this occurs, an internal error code is either returned to the caller, or is transmitted to the caller's fault handler as a result of the Memory Manager issuing a CPFaultTask call. In addition, certain fault conditions relating to memory objects may be detected asynchronously by the page fault handler and reported to the task's fault handler. See *SPL Volume 4: Memory Manager* for a description of the error codes that can occur.

Chapter 20. Miscellaneous Functions

CPAlterRange

```
Syntax -

     long int CPAlterRange(Type,Real_Address)

     unsigned long int Type;
     unsigned long int Real_Address;


Type      This value defines the type of memory and its attributes.

Real_Address
          The address in real memory of the first byte of the range to be altered.
```

Usage Notes

This function is used to alter the character of a range of existing real memory. The existing range must be adapter memory; it is changed to general purpose memory. The reverse alteration is not possible. The entire range of adapter memory must be free at the time of the call. The real memory type must be specified, and applies to all the page frames in the range. The real address must be that of the beginning of an existing range of adapter memory. The range is marked cachable.

In order for the alteration to succeed, the range must be within the first 64 MB (physical address).

This function is only available to code having I/O privilege. General purpose memory is assumed to be entirely present and functional, and it is put in immediate service by the Memory Manager. Conversely, the adapter memory need not all be present.

The Type parameter defines the following:

QMtype1, QMtype2 or QMtype3
A request for a specific memory type, that is, use only the memory type that is specified. This must be specified.

Implementation Notes

None

Return Codes

The Memory Manager provides a return code on all calls. Except for a success return code (QMsuccess), which has a value of 0, all Memory Manager return codes have the form **0x8002xxxx**. The possible return codes for this call, along with their low order 16 bits, are listed below.

QMsuccess	Request was successful.
QMbad_type-(0x0002)	The real memory type was not specified.

QMbad_addr-(0x0005)	The real address was not page aligned.
QMbad_range-(0x0008)	The real address is not that of an existing range of adapter memory, or the range specified extends beyond the first 64 MB.
QMno_page-(0x0012)	A new page table was required to map this memory and no free page frame exists to provide this page table.
QMreal_alloc-(0x0016)	The range was not entirely free.
QMrestricted_function-(0x0030)	Caller does not have I/O privilege.

Memory Manager Generated Faults

It is possible that the memory manager will detect internal or system errors during the processing of this call. If this occurs, an internal error code is either returned to the caller, or is transmitted to the caller's fault handler as a result of the Memory Manager issuing a CPFaultTask call. In addition, certain fault conditions relating to memory objects may be detected asynchronously by the page fault handler and reported to the task's fault handler. See *SPL Volume 4: Memory Manager* for a description of the error codes that can occur.

CPCreateRange

```
Syntax -

     long int CPCreateRange(Type,Size,Real_Address)

     unsigned long int Type;
     unsigned long int Size;
     unsigned long int Real_Address;


Type      This value defines the type of memory and its attributes.

Size      This is the number of bytes in the range.

Real_Address
          The address in real memory of the first byte of the range being defined.
```

Usage Notes

This function is used to create a new range of available real memory. If the memory is general purpose memory, it is made available to satisfy normal allocation requests. The explicit real memory type must be specified for general purpose memory. Because the default real memory type is type 1, normally you would specify type 1 on the create range request. Instead, if the memory is adapter memory, it is reserved and used only to satisfy CPAllocBase and CPAllocRange calls. The starting real address must be on a page boundary and the size a page multiple.

The cachability of the range may also be specified. It should be noted that these attributes apply to all the page frames in the range. General purpose memory must be located at physical (real) addresses below 64 MB. However, it does not need to be contiguous as there could be several ranges all below 64MB.

This function is only available to code having I/O privilege.

The Type parameter defines the following:

QMtype1, QMtype2 or QMtype3
A request for a specific memory type, that is, use only the memory type that is specified.

QMmap_adapter
Indicates range is adapter memory with mapped option.

QMcache_mem
Indicates range is cacheable memory.

QMnoncache_mem
Marks memory as noncacheable.

QMadapter_mem
Indicates range is adapter memory; otherwise, it is treated as general purpose storage and added to the pool of free page frames.

QMadapter_mem
Indicates range is adapter memory; otherwise, it is treated as general purpose storage and added to the pool of free page frames.

Implementation Notes

All general purpose storage is assumed by the code to be cacheable. Only adapter memory requests monitor the cacheability attribute. This could be changed, but doing so would add path length to all allocation calls and, currently, there does not appear to any justification for this.

There are two categories of adapter memory. Normal adapter memory can be located anywhere in the physical address space. It is not mapped in the kernel's DRMM (Direct Real Memory Map), and when allocations are made through the CPAllocBase of CPAllocRange call, it must be accessed by the offset returned by that call. Conversely, mapped adapter memory is like general purpose memory and must be located in the first 64MB of physical address space. It is mapped in the kernel's DRMM, and when allocations are made through the CPAllocBase call, it can be accessed by the offset returned by that call, but it also can be accessed directly by supervisor level code using the underlying real addresses. This direct access is limited to supervisor code and is inherently more risky than access via the returned offset. Furthermore, this only works with systems in which the DRMM is located at offset 0, which is not guaranteed. General purpose memory is assumed to be entirely present and functional and is put in immediate service by the Memory Manager. Conversely, adapter memory need not all be present. The Memory Manager does not examine adapter memory in any way, but it assumes that any one requesting the memory by specific real address does understand the nature of the underlying memory.

Return Codes

The Memory Manager provides a return code on all calls. Except for a success return code (QMsuccess), which has a value of 0, all Memory Manager return codes have the form **0x8002xxxx**. The possible return codes for this call, along with their low order 16 bits, are listed below.

QMsuccess	Request was successful.
QMbad_type-(0x0002)	The range is general purpose memory and the explicit real memory type was not specified.
QMbad_size-(0x0004)	The size requested was zero or was not page aligned.
QMbad_addr-(0x0005)	The real address range specified for general purpose memory was greater than 64MB.
QMbad_range-(0x0008)	The real address range specified conflicts with an existing range of real memory.
QMno_page-(0x0012)	A new page table was required to map this memory and no free page frame exists to provide this page table.
QMno_range-(0x0014)	No free range available in system range map table to accommodate this request.
QMno_PFD-(0x0015)	No free PFDs available in system to define real storage specified in this request.
QMrestricted_function-(0x0030)	Caller does not have I/O privilege.

Memory Manager Generated Faults

It is possible that the memory manager will detect internal or system errors during the processing of this call. If this occurs, an internal error code is either returned to the caller, or is transmitted to the caller's fault handler as a result of the Memory Manager issuing a CPFaultTask call. In addition, certain fault conditions relating to memory objects may be detected asynchronously by the page fault handler and reported to the task's fault handler. See *SPL Volume 4: Memory Manager* for a description of the error codes that can occur.

CPForceNPPages

```
Syntax -

     long int CPForceNPPages(Type,Offset)

     unsigned long int Type;
     void *Offset;


Type      This value contains information about the attributes of the memory object being
          referenced.

Offset    Offset of the page (must be within a valid memory object) to be marked
          force-not-present.
```

Usage Notes

This function forces a page within an object to be marked force-not-present. As a result of this action, a page frame is not assigned as a result of a page fault caused by accessing this page. Instead, this is treated as an error, and the task is faulted. This is useful, for example, to force a failure on stack overrun to achieve isolation of data within an address space.

The Type parameter defines the following:

QMold_eff
: This is used to indicate that the incoming object is to be defined in the address space of the caller's effective address space.

QMold_current
: This is used to indicate that the incoming object is to be defined in the caller's current address space.

Implementation Notes

In systems built with a memory manager which was compiled with the forced assignment option, this call is effectively a “No-Op”.

Return Codes

The Memory Manager provides a return code on all calls. Except for a success return code (QMsuccess), which has a value of 0, all Memory Manager return codes have the form **0x8002xxxx**. The possible return codes for this call, along with their low order 16 bits, are listed below.

QMsuccess	Request was successful, page marked.
QMbad_offset-(0x0001)	Specified offset is not within a valid memory object, or the page frame referenced is a present page. Page not marked not present.
QMbad_type-(0x0002)	Type field error, user level caller referencing supervisor object.
QMwrap_count-(0x0021)	The call could not be processed because the memory lock is at its maximum; that is, there are 255 tasks currently executing within Memory Manager code referring to this object.

QMprocess_dead-(0x0022)	The process, in whose address space the object was defined, is in the final stages of the Memory Manager's process removal. No new requests to modify memory objects can be honored. This should not normally occur, particularly if the Resource Manager resource provider facilities have been used.
QMbusy_fork-(0x0035)	The process, in which the referenced object exists, is performing a fork operation. The request cannot be accommodated at this time.

Memory Manager Generated Faults

It is possible that the memory manager will detect internal or system errors during the processing of this call. If this occurs, an internal error code is either returned to the caller, or is transmitted to the caller's fault handler as a result of the Memory Manager issuing a CPFaultTask call. In addition, certain fault conditions relating to memory objects may be detected asynchronously by the page fault handler and reported to the task's fault handler. See *SPL Volume 4: Memory Manager* for a description of the error codes that can occur.

CPGetVersion

```
Syntax -

     long int CPGetVersion(&Version,&Release)

     unsigned long int Version;
     unsigned long int Release;

Version   Variable in which to return the version number.

Release   Variable in which to return the release number.
```

Usage Notes

This function returns the version number and release number of the Memory Manager.

Implementation Notes

None

Return Codes

The Memory Manager provides a return code on all calls. Except for a success return code (QMsuccess), which has a value of 0, all Memory Manager return codes have the form **0x8002xxxx**. The possible return codes for this call, along with their low order 16 bits, are listed below.

QMsuccess	Request was successful, requested information was returned.

Memory Manager Generated Faults

It is possible that the memory manager will detect internal or system errors during the processing of this call. If this occurs, an internal error code is either returned to the caller, or is transmitted to the caller's fault handler as a result of the Memory Manager issuing a CPFaultTask call. In addition, certain fault conditions relating to memory objects may be detected asynchronously by the page fault handler and reported to the task's fault handler. See *SPL Volume 4: Memory Manager* for a description of the error codes that can occur.

CPNoSwap

```
Syntax -

     long int CPNoSwap(Type,Offset)

     unsigned long int Type;
     void *Offset;


Type      This value determines the address space of the incoming object.

Offset    Offset of the object to be made non-swappable.
```

Usage Notes

This function marks an object non-swappable.

The Type parameter is used to define the incoming object's address space, as follows:

QMold_eff
: This is used to indicate that the incoming object is to be defined in the address space of the caller's effective address space.

QMold_current
: This is used to indicate that the incoming object is to be defined in the caller's current address space.

Implementation Notes

This is a non-functional stub at this time. Currently, no plans exist to provide any swapping or “page turning” mechanism in CP/Q. The stub is provided to allow code which might be sensitive to any future swapping mechanism to make provision for it.

Return Codes

The Memory Manager provides a return code on all calls. Except for a success return code (QMsuccess), which has a value of 0, all Memory Manager return codes have the form **0x8002xxxx**. The possible return codes for this call, along with their low order 16 bits, are listed below.

QMsuccess	Request was successful, object now non-swappable.
QMbad_offset-(0x0001)	Specified offset is not that of a valid memory object.
QMbad_type-(0x0002)	Type field error. The caller is trying to mark supervisor level object without supervisor privilege.
QMwrap_count-(0x0021)	The call could not be processed because the memory lock is at its maximum; that is, there are 255 tasks currently executing within Memory Manager code referring to this object.

Memory Manager Generated Faults

It is possible that the memory manager will detect internal or system errors during the processing of this call. If this occurs, an internal error code is either returned to the caller, or is transmitted to the caller's fault handler as a result of the Memory Manager issuing a CPFaultTask call. In addition, certain fault conditions relating to memory objects may be detected asynchronously by the page fault handler and reported to the task's fault handler. See *SPL Volume 4: Memory Manager* for a description of the error codes that can occur.

CPQueryOwn

```
Syntax -

     long int CPQueryOwn(Offset_In,&Size,&ObjProcID,&ObjType,&Offset_Out)

     void *Offset_In;
     unsigned long int Size;
     unsigned long int ObjProcID;
     unsigned long int ObjType;
     void *Offset_Out;


Offset_In The caller specifies this either as the offset of the memory object being
          referenced, or as an offset within the memory object.  The memory object being
          referenced must be defined in the caller's effective process.

Size      This input determines the size, in bytes, to check.  The area from Offset_In to
          (Offset_In + (Size - 1)) is verified and must be within a single memory object.
          Upon successful return, this variable contains the size (from Offset_In) of the
          underlying physical block of storage.

ObjProcID Variable in which to return the Process ID of the object's owner.

ObjType   Variable in which to return type information about the object.

Offset_Out
          Variable in which to return the offset of (the beginning of) the memory object.
```

Usage Notes

This function requests the owner of the memory object defined by Offset_In or containing Offset_In, and provides additional status information about the object.

This call returns the offset of the beginning of the memory object in the variable Offset_out.

The call returns the size of the block of storage in the parameter Size. The size returned is the number of bytes from Offset_In to the end of the physically allocated block, that is the lesser of the size to the end of the block, the Size given on the call, or the size to the end of the object.

Upon return, the ObjType parameter contains information about the memory object. It consists of a number of bit and bit fields of information. The bit fields must be isolated by the appropriate masks before they are tested against a value. It is defined as follows:

The class of the object can be isolated with **QMclass_bits** and is defined as:

QMglobal	This indicates a Global class object.
QMprivate	This indicates a Private class object.
QMalias	This indicates object is an alias.
QMcommon	This indicates a Common class object.

The following additional bits apply only to Common objects:

- Privilege Level Limit

  QMshare_supvr
  This is set to indicate the limit of access to be allowed to other processes is supervisor access. If clear, then limit of access is user access, that is, **QMshare_user** was specified when the object was created.

- Read/Write Access Limit

  QMshare_read
  This is set to indicate the limit of access to be allowed to other processes is read only access. If clear, then limit of access is read/write access, that is, **QMshare_write** was specified when the object was created.

- Give/Get Access

  QMget This is set to indicate the access mechanism for Common class objects is through the CPGetMem call. If clear, then it indicates the access mechanism for Common class objects through the CPGiveMem call, that is, **QMgive** was specified when the object was created.

- Copy/Share Mode

  QMcopy This is set to indicate copy mode for the object. If clear, then share mode is to be used, that is, **QMshare** was specified when the object was created.

  QM_COW This indicates that the copy is to be performed by copy on write. Immediate copy is performed if this is not specified.

The real memory type can be isolated with **QMtype_mask** and is defined as:

QMtype_default
A request for the default memory type, that is, use memory type 1 if available; otherwise memory types 2 or 3 are used in that order.

QMtype1, QMtype2 or QMtype3
A request for a specific memory type, that is, use only the memory type that is specified.

The fix count can be isolated with **QMfix_count** and contains the number of outstanding fix requests on this object shifted left 8 bits. This means FixCount = (ObjType & QMfix_count)>>8.

The following individual bits indicate further object status:

QMsparse This indicates if the object is sparse.

QMoriginal This bit is set if this object is an “original” object. This means it is set when an object is initially created by an allocate call, and it is clear on objects obtained by a give, get, or alias call.

QMshared This bit is set if this object is sharing physical access (by alias or shared mode give or get) and is not the original.

QMpage_allocated
Set if page frame is allocated at Offset_In, clear if frame is not present.

QMpendfree
Set if this object has been marked to be freed as soon as possible. This can occur during process termination.

QMadapter_mem
This indicates that the object maps to I/O (adapter) memory.

QMuser This is used to indicate user privilege level.

QMsupervisor
This is used to indicate supervisor privilege level.

QMwrite This is used to indicate read/write access.

QMread This is used to indicate read-only access.

Implementation Notes

The real storage state of the referenced memory is returned by this call and is determined by the following steps. First, the page underlying the linear address specified by Offset_In is examined to determine if a real storage page frame or backing store has been allocated to it. The status of that frame is returned in ObjType by setting **QMpage_allocated** if a frame is allocated, or if a frame is not allocated, when it is clear. Second, any additional page frames underlying the linear address range up to (Offset_In + (Size - 1)) are examined sequentially until the first occurrence of one of the following conditions:

- A page is found whose allocation status differs from the first page examined
- The page containing the linear address (Offset_In + (Size - 1)) is reached
- The page containing the end of the object is reached

The call returns the size of the block of storage in Size. The size is the number of bytes from Offset_in to the end of the block, that is, the lesser of the size to the end of the block, the size given on the call, or the size to the end of the object.

Return Codes

The Memory Manager provides a return code on all calls. Except for a success return code (QMsuccess), which has a value of 0, all Memory Manager return codes have the form **0x8002xxxx**. The possible return codes for this call, along with their low order 16 bits, are listed below.

QMsuccess	Request was successful, requested information was returned.
QMbad_offset-(0x0001)	Offset_In not that of a valid memory object.
QMbad_type-(0x0002)	Caller does not have proper privilege to access the object.
QMbad_size-(0x0004)	Offset_In was that of a valid memory object, but size extends beyond end of object.
QMwrap_count-(0x0021)	The call could not be processed because the memory lock is at its maximum; that is, there are 255 tasks currently executing within Memory Manager code referring to this object.

Memory Manager Generated Faults

It is possible that the memory manager will detect internal or system errors during the processing of this call. If this occurs, an internal error code is either returned to the caller, or is transmitted to the caller's fault handler as a result of the Memory Manager issuing a CPFaultTask call. In addition, certain fault conditions relating to memory objects may be detected asynchronously by the page fault handler and reported to the task's fault handler. See *SPL Volume 4: Memory Manager* for a description of the error codes that can occur.

CPQueryObjectState

```
Syntax -

     long int CPQueryObjectState(Offset_In,&Size,&ObjProcID,&ObjType,&Offset_Out)

     void *Offset_In;
     unsigned long int Size;
     unsigned long int ObjProcID;
     unsigned long int ObjType;
     void *Offset_Out;


Offset_In The caller specifies this either as the offset of the memory object being
          referenced, or as an offset within the memory object.  The memory object being
          referenced must be defined in the caller's effective process.

Size      This input determines the size, in bytes, to check.  The area from Offset_In to
          (Offset_In + (Size - 1)) will be verified and must be within a single memory
          object.  Upon successful return, this variable contains the size of the entire
          memory object starting at Offset_Out.

ObjProcID Variable in which to return the Process ID of the object's owner.

ObjType   Variable in which to return type information about the object.

Offset_Out
          Variable in which to return the offset of (the beginning of) the memory object.
```

Usage Notes

This function requests the base offset and size of the memory object defined by or containing Offset_In, and it provides additional status information about the object.

This function is similar to the function CPQueryOwn, except that the returned Size for CPQueryObjectState is the size of the entire object, and the real storage state of the referenced memory is not returned.

This call returns in Offset_Out the offset of the beginning of the memory object.

The call returns in the parameter Size the size of the *entire* object. This is the number of bytes from Offset_Out, the beginning of the memory object, to the end of the memory object.

Upon return, the ObjType parameter contains information about the memory object. It consists of a number of bit and bit fields of information. The bit fields must be isolated by the appropriate masks before they are tested against a value. It is defined as follows:

The class of the object can be isolated with **QMclass_bits** and is defined as:

QMglobal	This indicates a Global class object.
QMprivate	This indicates a Private class object.
QMalias	This indicates object is an alias.

QMcommon
This indicates a Common class object.

The following additional bits apply only to Common objects:

- Privilege Level Limit

  QMshare_supvr
  This is set indicate the limit of access to be allowed to other processes is supervisor access. If clear, then limit of access is user access, that is, **QMshare_user** was specified when the object was created.

- Read/Write Access Limit

  QMshare_read
  This is set to indicate the limit of access to be allowed to other processes is read only access. If clear, then limit of access is read/write access, that is, **QMshare_write** was specified when the object was created.

- Give/Get Access

  QMget This is set to indicate the access mechanism for Common class objects is via the CPGetMem call. If clear, then it indicates the access mechanism for Common class objects through the CPGiveMem call, that is **QMgive** was specified when the object was created.

- Copy/Share Mode

  QMcopy This is set to indicate copy mode for the object. If clear, then share mode is to be used, that is, **QMshare** was specified when the object was created.

  QM_COW This indicates that the copy is to be performed by copy on write. Immediate copy is performed if this is not specified.

The real memory type can be isolated with **QMtype_mask** and is defined as:

QMtype_default
A request for the default memory type, that is, use memory type 1 if available; otherwise memory types 2 or 3 are used in that order.

QMtype1, QMtype2 or QMtype3
A request for a specific memory type, that is, use only the memory type that is specified.

The fix count can be isolated with **QMfix_count** and contains the number of outstanding fix requests on this object shifted left 8 bits. This means FixCount = (ObjType & QMfix_count)>>8.

The following individual bits indicate further object status:

QMsparse This indicates if the object is sparse.

QMoriginal	This bit is set if this object is an “original” object. This means it is set when an object is initially created by an allocate call, and it is clear on objects obtained by a give, get, or alias call.
QMshared	This bit is set if this object is sharing physical access (via alias or shared mode give or get) and is not the original.
QMpendfree	Set if this object has been marked to be freed as soon as possible. This can occur during process termination.
QMadapter_mem	This indicates that the object maps to I/O (adapter) memory.
QMuser	This is used to indicate user privilege level.
QMsupervisor	This is used to indicate supervisor privilege level.
QMwrite	This is used to indicate read/write access.
QMread	This is used to indicate read-only access.

Implementation Notes

None

Return Codes

The Memory Manager provides a return code on all calls. Except for a success return code (QMsuccess), which has a value of 0, all Memory Manager return codes have the form **0x8002xxxx**. The possible return codes for this call, along with their low order 16 bits, are listed below.

QMsuccess	Request was successful, requested information was returned.
QMbad_offset-(0x0001)	Offset_In not that of a valid memory object.
QMbad_type-(0x0002)	Caller does not have proper privilege to access the object.
QMbad_size-(0x0004)	Offset_In was that of a valid memory object, but size extends beyond end of object.
QMwrap_count-(0x0021)	The call could not be processed because the memory lock is at its maximum; that is, there are 255 tasks currently executing within Memory Manager code referring to this object.

Memory Manager Generated Faults

It is possible that the memory manager will detect internal or system errors during the processing of this call. If this occurs, an internal error code is either returned to the caller, or is transmitted to the caller's fault handler as a result of the Memory Manager issuing a CPFaultTask call. In addition, certain fault conditions relating to memory objects may be detected asynchronously by the page fault handler and reported to the task's fault handler. See *SPL Volume 4: Memory Manager* for a description of the error codes that can occur.

CPRealAddr

```
Syntax  -

      long int CPRealAddr(Type,Offset,&Real_Address)

      unsigned long int Type;
      void *Offset;
      void *Real_Address;


Type      This value determines the address space of the incoming object.

Offset    Offset of the memory object whose real address is requested, or an offset within
          the memory object.

Real_Address
          Variable in which is returned the real storage address of the object associated
          with Offset, if there is real storage assigned to that Offset.
```

Usage Notes

This function returns the Real_Address associated with the Offset provided on input to the call, if there is real storage assigned to that Offset. This call requires I/O privilege. This call has limited value as it does not fix the page, nor does it assign a page frame if the page is currently marked not present. Thus, there is no assurance that the page frame remains assigned, that is, the object could be deleted successfully in spite of this call.

The Type parameter is used to define the incoming object's address space, as follows:

QMold_eff
This is used to indicate that the incoming object is defined in the address space of the caller's effective address space.

QMold_current
This is used to indicate that the incoming object is defined in the caller's address space.

Implementation Notes

None

Return Codes

The Memory Manager provides a return code on all calls. Except for a success return code (QMsuccess), which has a value of 0, all Memory Manager return codes have the form **0x8002xxxx**. The possible return codes for this call, along with their low order 16 bits, are listed below.

QMsuccess	Request was successful, Real_Address returned.
QMbad_offset-(0x0001)	Specified Offset is not within a valid memory object.
QMno_page-(0x0012)	There is no real storage page frame assigned at this Offset.

QMwrap_count-(0x0021)	The call could not be processed because the memory lock is at its maximum; that is, there are 255 tasks currently executing within Memory Manager code referring to this object.
QMrestricted_function-(0x0030)	The caller does not have I/O privilege.

Memory Manager Generated Faults

It is possible that the memory manager will detect internal or system errors during the processing of this call. If this occurs, an internal error code is either returned to the caller, or is transmitted to the caller's fault handler as a result of the Memory Manager issuing a CPFaultTask call. In addition, certain fault conditions relating to memory objects may be detected asynchronously by the page fault handler and reported to the task's fault handler. See *SPL Volume 4: Memory Manager* for a description of the error codes that can occur.

CPRemoveNPPages

```
Syntax -

     long int CPRemoveNPPages(Type,Offset)

     unsigned long int Type;
     void *Offset;


Type      This value contains information about the attributes of the memory object being
          referenced.

Offset    Offset of the page (must be within a valid memory object) to be have
          force-not-present state removed.
```

Usage Notes

This function reverses the effect of a previous call to CPForceNPPages. It removes the force-not-present state from a page frame, thus allowing a page frame to be assigned on a subsequent page fault.

The Type parameter defines the following:

QMold_eff
: This is used to indicate that the incoming object is defined in the address space of the caller's effective address space.

QMold_current
: This is used to indicate that the incoming object is defined in the caller's address space.

Implementation Notes

In systems built with a memory manager which was compiled with the forced assignment option, this call is effectively a “No-Op”.

Return Codes

The Memory Manager provides a return code on all calls. Except for a success return code (QMsuccess), which has a value of 0, all Memory Manager return codes have the form **0x8002xxxx**. The possible return codes for this call, along with their low order 16 bits, are listed below.

QMsuccess	Request was successful, page no longer force-not-present.
QMbad_offset-(0x0001)	Specified offset is not within a valid memory object, or page frame referenced is a present page, or page frame not marked force-not-present.
QMbad_type-(0x0002)	Type field error, user level caller referencing supervisor level object.
QMwrap_count-(0x0021)	The call could not be processed because the memory lock is at its maximum; that is, there are 255 tasks currently executing within Memory Manager code referring to this object.

QMprocess_dead-(0x0022)	The process, in whose address space the object was defined, is in the final stages of the Memory Manager's process removal. No new requests to modify memory objects can be honored. This should not normally occur, particularly if the Resource Manager resource provider facilities have been used.
QMbusy_fork-(0x0035)	The process, in which the referenced object exists, is performing a fork operation. The request cannot be accommodated at this time.

Memory Manager Generated Faults

It is possible that the memory manager will detect internal or system errors during the processing of this call. If this occurs, an internal error code is either returned to the caller, or is transmitted to the caller's fault handler as a result of the Memory Manager issuing a CPFaultTask call. In addition, certain fault conditions relating to memory objects may be detected asynchronously by the page fault handler and reported to the task's fault handler. See *SPL Volume 4: Memory Manager* for a description of the error codes that can occur.

CPSwap

```
Syntax -

     long int CPSwap(Type,Offset)

     unsigned long int Type;
     void *Offset;


Type      This value determines the address space of the incoming object.

Offset    Offset of the object to be made swappable.
```

Usage Notes

This function marks an object swappable. This means it reverses the effect of a previous call to CPNoSwap.

The Type parameter is used to define the incoming object's address space, as follows:

QMold_eff
This is used to indicate that the incoming object is defined in the address space of the caller's effective address space.

QMold_current
This is used to indicate that the incoming object is defined in the caller's address space.

Implementation Notes

This is a non-functional stub at this time. Currently, no plans exist to provide any swapping mechanism in CP/Q. The stub is provided to allow code which might be sensitive to any future swapping mechanism to make provision for it.

Return Codes

The Memory Manager provides a return code on all calls. Except for a success return code (QMsuccess), which has a value of 0, all Memory Manager return codes have the form **0x8002xxxx**. The possible return codes for this call, along with their low order 16 bits, are listed below.

QMsuccess	Request was successful, object now swappable.
QMbad_offset-(0x0001)	Specified offset is not that of a valid memory object.
QMbad_type-(0x0002)	Type field error. The caller is trying to mark supervisor level object without supervisor privilege.
QMwrap_count-(0x0021)	The call could not be processed because the memory lock is at its maximum; that is, there are 255 tasks currently executing within Memory Manager code referring to this object.

Memory Manager Generated Faults

It is possible that the memory manager will detect internal or system errors during the processing of this call. If this occurs, an internal error code is either returned to the caller, or is transmitted to the caller's fault handler as a result of the Memory Manager issuing a CPFaultTask call. In addition, certain fault conditions relating to memory objects may be detected asynchronously by the page fault handler and reported to the task's fault handler. See *SPL Volume 4: Memory Manager* for a description of the error codes that can occur.

Resource Manager Calls

Chapter 21. Section Notes

Each of the following descriptions lists the Resource Manager generated faults for the call. These faults are presented to the fault handler for the task, which was assigned when the task was created. Please refer to *SPL Volume 3: SVC Handler and System Data Area* for more information about faults and fault handlers.

All routines in this section require that *cpqlib.h* be included in the calling program.

All structures and typedefs which appear in this section are defined in *cpqlib.h*.

Note: The Resource Manager may enable interrupts while processing a request. Applications which disable interrupts should take this into consideration.

The Resource Manager claims the SDA semaphore immediately after performing initial validity checks and setup for creation, deletion and tracking functions. Once the Resource Manager function has been performed, the SDA semaphore is released. This may cause the calling task to be suspended until the semaphore becomes available.

Chapter 22. Query Process States

The functions described in this chapter allow a task to query the state of other processes. These functions are:

- **CPQueryAuth -** This function informs a caller if a task has the authority to act on a process.
- **CPQueryEffProc -** This function returns the effective process of a task.
- **CPGetProcName -** This function is used to obtain the name and ID of a process. The target process can be identified directly by its ID or indirectly as the process that owns a system resource.
- **CPGetProcID -** This function is used to obtain the ID of a process given its eight-character name.
- **CPQueryProc -** This function returns the state and type attributes of a process, the name of the youngest child process (if any), and the name of the next older sibling process (if any). The latter two items allow a process to traverse the process hierarchy if required.
- **CPQueryProcRes -** This function returns the current minimum, maximum and in-use values for system resources held by the specified process.
- **CPQueryProcSVIDs -** This function returns the list of SVids that are attached to a process.

Function prototypes are listed in cpqlib.h. Structure definitions for applicable query functions are listed in cpqlib.h, and constant definition macros are defined in the file qrm_cnst.h.

CPQueryAuth

```
Syntax -

#include "cpqlib.h"

     long int CPQueryAuth(TaskID,ProcID)

     unsigned long int TaskID;
     unsigned long int ProcID;

          TaskID   - SVid of task that is being checked for authorization
                     to ProcID.
          ProcID   - process ID which task is querying authorization to.
```

Usage Notes

This function informs the caller if a task has the authority to act on a process, and it is available to any requestor (independent of the relationship of the requestor to the process being queried).

The purpose of this function is to allow a task or a third party (such as a server) to validate a client request made to another process before continuing.

A task has the authority to act on behalf of a process if at least one of the following conditions is true:

- The proposed process to be acted on is a descendant of the caller.
- The task has “set-effective-process” privilege (the *TCBffenvpriv* in the TCB must be set).

This call always returns *QRMgood* if the caller has set-effective-process privilege, unless the caller is not in the “SYSTEM” process and is attempting to act on the “SYSTEM” process.

The only returned information is a return code specifying whether it is legal or not for the task to act on the process.

Note: This function does not give or take away authorization to the task, but it verifies if the task can act on the process.

Implementation Notes

None

Return Codes

The Resource Manager provides a return code on all calls. A success return code *QRMgood* has the value 0; all other Resource Manager return codes are of the form **0x8003xxxx**. The possible Resource Manager return codes for this call, along with their low order 16 bits, are listed below.

QRMgood	The task specified by the task ID has authorization to act on the process specified by the process ID.
QRMbadproc-(0x000E)	Invalid process specification in *ProcID*.

QRMbadsvt-(0x000F)	Invalid task SVid specification in *TaskID*.
QRMillegal-(0x0019)	The task specified by the task ID does not have authorization to act on the process specified by the process ID. A request to alter the state of the process using the task as the effective process is rejected.

CPQueryEffProc

```
Syntax -

#include "cpqlib.h"
#include "qrm_cnst.h"

     long int CPQueryEffProc(ID,EffProc)

     unsigned long int ID;
     unsigned long int *EffProc;

          ID       - SVid of task to query.
          EffProc  - variable in which the effective process ID
                     is returned.
```

Usage Notes

This function is used to obtain the effective process of a task.

Note: A task does not need set-effective-process privilege to execute this call.

If the *ID* parameter is set to *QRMproctask*, then the effective process field of the caller is returned.

This Resource Manager call does not assign set-effective-process privilege or change a task's effective process. Instead, this Resource Manager call only returns the effective process of a task.

Upon successful completion of the call, *EffProc* contains the effective process ID.

Implementation Notes

None

Return Codes

The Resource Manager provides a return code on all calls. A success return code *QRMgood* has the value 0; all other Resource Manager return codes are of the form **0x8003xxxx**. The possible Resource Manager return codes for this call, along with their low order 16 bits, are listed below.

QRMgood	Function completed.
QRMbadsvt-(0x000F)	Not a valid SVid in variable *ID*, or *ID* is not the SVid of a task.

Note: A task obtains the privilege to change effective process, for purposes of attaching various resources to other processes, in three ways:

- The system builder specified to SLEEP that the *TCBffenvpriv* value is set in the PRIV field of the TASK command.
- The task was created by another task that has this privilege.
- A task with this privilege requests that the privilege also be assigned to another (already defined) task.

CPGetProcName

```
Syntax -

#include "cpqlib.h"
#include "qrm_cnst.h"

     long int CPGetProcName(ID,Mode,ProcID,NameBuf)

     unsigned long int ID;
     unsigned long int Mode;
     unsigned long int *ProcID;
     char *NameBuf;

          ID       - Process ID or SVid (depends on Mode).
          Mode     - Used to determine process to be queried (see below).
          ProcID   - Variable in which the process ID is
                     returned.
          NameBuf  - Near pointer to an 8 character name buffer
                     used to return the name.
```

Usage Notes

This function is used to obtain the name of a specified process. The target process can be specified directly (through its process ID), or indirectly as the process to which a system object belongs (through the SVid of the object). The process ID is also returned in *ProcID* upon successful completion of the call.

The *Mode* parameter determines the process to be queried. *Mode* values include:

QRMproctask	Use process of caller
QRMprocefftask	Use *effective* process of caller
QRMprocsvcid	Use process to which the SVT entry specified in *ID* belongs
QRMprocid	Use process whose process ID is in *ID*

Implementation Notes

None

Return Codes

The Resource Manager provides a return code on all calls. A success return code *QRMgood* has the value 0; all other Resource Manager return codes are of the form **0x8003xxxx**. The possible Resource Manager return codes for this call, along with their low order 16 bits, are listed below.

QRMgood	Function completed.
QRMbadproc-(0x000E)	Invalid process ID specified.
QRMnowacc-(0x0029)	The caller does not have write access to the offset pointed to by *Namebuf*.

CPGetProcID

```
Syntax -

#include "cpqlib.h"

     long int CPGetProcID(Name,ProcID)

     char *Name;
     unsigned long int *ProcID;

          Name     - Near pointer to 8 character name.  If the
                     name is less than 8 characters, and string
                     functions are used to fill it in, be sure
                     that characters after the terminating null
                     are either blank or null out to the 8
                     character field length.
          ProcID   - Variable in which the process ID is
                     returned.
```

Usage Notes

This function is used to obtain a process ID from a process name, regardless of whether it is named or unnamed. The Resource Manager puts the incoming name into normalized format (internally) before beginning the search, but it does not alter the name pointed to by *Name*. The function fails if there is no *named* or *unnamed* process whose name matches.

Implementation Notes

None

Return Codes

The Resource Manager provides a return code on all calls. A success return code *QRMgood* has the value 0; all other Resource Manager return codes are of the form **0x8003xxxx**. The possible Resource Manager return codes for this call, along with their low order 16 bits, are listed below.

QRMgood	Function completed.
QRMbadname-(0x000D)	Name field contains illegal characters, or no process exists with this name.

CPQueryProc

```
Syntax -

#include "cpqlib.h"
#include "qrm_cnst.h"

     long int CPQueryProc(ID,Mode,QPBAddr)

     unsigned long int ID;
     unsigned long int Mode;
     void *QPBAddr;

          ID       - Process ID or SVid (depends on Mode).
          Mode     - Used to determine process to be queried (see below).
          QPBAddr  - Near pointer to output buffer.
```

Usage Notes

This function is used to obtain the current characteristics of the designated process or processes. The process IDs of the youngest child and next older sibling (if any) are also returned. To get the current characteristics of the other processes and sub-processes, it is necessary for the caller to traverse the tree and call this function for each process it finds in the tree.

The *Mode* parameter determines the process to be queried. *Mode* values include:

QRMproctask	Use process of caller
QRMprocefftask	Use *effective* process of caller
QRMprocsvcid	Use process to which the SVT entry specified in *ID* belongs
QRMprocid	Use process whose process ID is in *ID*

Output Buffer

The parameter *QPBAddr* points to the output buffer whose structure is defined in cpqlib.h as QPBLOCK. Constant definition macros are defined in the Resource Manager user header file qrm_cnst.h.

```
Query Process Return Block Format -

     typedef struct qpblock {
       unsigned short int PCBcount; /* PCBs in use                 */
       unsigned short int PCBtotal; /* Total PCBs in system        */
       unsigned long int ProcID;    /* Process ID                  */
       unsigned long int ParentID;  /* Process ID of parent        */
       unsigned char Type;          /* Process type flags:         */
       unsigned char State;         /* Current process state:      */
       unsigned char PrioBase;      /* Priority base value         */
       unsigned char rsvd1;         /* Reserved (should be 0)      */
       unsigned long int SiblingID; /* Process ID of the next      */
                                    /*   older sibling             */
       unsigned long int ChildID;   /* Process ID of youngest child */
       unsigned long Reserved;      /* reserved field              */
       char ProcName[8];            /* Process name                */
     } QPBLOCK;
```

Figure 22-1. Query Process - output buffer format

PCBcount	Number of PCBs currently in use
PCBtotal	Total number of PCBs in the system
ProcID	This is the process ID of the process whose characteristics are being returned in this record.
ParentID	This is the process ID of the parent process, or else is the process ID of the closest ancestor (grandparent, great-grandparent, and so on) that is within the scope of the query (that is, either the base process or some sub-process of the base). If no such process exists, this field is zeroed. In particular, this field is always zero in the first record.
Type	Process type flags; valid values are :

QRMqryrmvpend	Process removal is pending for this process.
QRMqryrp	Process is a resource provider.
QRMqryrmvable	Process is removable.
QRMqrynamed	Process is named.

State Current state of this process; valid values are:

QRMnorm	"Normal" status.
QRMproc	Process creation underway for this creator process.
QRMfork	Process is being forked.
QRMcrea	Process is currently being created.
QRMexec	Program exec is currently underway in this process.
QRMremove	Process is being removed.
QRMfail	Process (or parent) has faulted and termination is in progress.

PrioBase	Base priority of this process.
rsvd1	Reserved, this should be 0.
SiblingID	Process ID of next older sibling. Returned in order that the caller can continue with a preorder search of the tree.
ChildID	Process ID of the youngest child of the process being queried. Returned in order that the caller can continue with a preorder search of the tree.
rsvd2	Reserved, this should be 0.
ProcName	Process name (eight characters in length).

Note: The data returned represents the state at the time the function is called; the information usually changes over time.

Implementation Notes

None

Return Codes

The Resource Manager provides a return code on all calls. A success return code *QRMgood* has the value 0; all other Resource Manager return codes are of the form **0x8003xxxx**. The possible Resource Manager return codes for this call, along with their low order 16 bits, are listed below.

QRMgood	Function completed.
QRMbadproc-(0x000E)	Invalid process specification in input registers.
QRMbadlength-(0x0015)	Output buffer length is not valid (length given is too small for information to be returned).
QRMnowacc-(0x0029)	The caller does not have write access to the offset pointed to by *QPBAddr*.

The Resource Manager calls the Memory Manager in order to complete this function. A return code other than *QMsuccess* from the Memory Manager is returned to the caller. Memory Manager return codes are in the form **0x8002xxxx**. The possible Memory Manager return codes for this call, along with their low order 16 bits, are listed below.

QMbad_offset-(0x0001)	Parameter list is invalid, for example *QPBAddr* does not point into a valid memory object.

Note: It is the intent that the information returned by this function be sufficient to allow a dynamically specified user interface program, running at outer privilege level, to correctly intercept user requests to modify processes without having access to system or Resource Manager data.

The process name is returned, whether or not the process is named.

CPQueryProcRes

```
Syntax -

#include "cpqlib.h"

     long int CPQueryProcRes(ID,Mode,QPRBAddr)

     unsigned long int ID;
     unsigned long int Mode;
     void *QPRBAddr;

          ID       - Process ID or SVid (depends on Mode).
          Mode     - Used to determine process to be queried (see below).
          QPRBAddr - Near pointer to output buffer.
```

Usage Notes

This function is used to obtain the current system resources and limits of the designated process. Information is returned for named and unnamed processes.

The *Mode* parameter determines the process to be queried. *Mode* values include:

QRMproctask	Use process of caller.
QRMprocefftask	Use *effective* process of caller.
QRMprocsvcid	Use process to which the SVT entry specified in *ID* belongs.
QRMprocid	Use process whose process ID is in *ID*.

Output Buffer

The parameter *QPRBAddr* points to the output buffer whose structure is defined in cpqlib.h as QPRESBLOCK.

```
Query Process Resources Return Block Format -

     typedef struct qpresblock {
       unsigned long int ProcID;    /* Process ID                        */
       unsigned short int SVTMin;   /* Minimum SVT entries reserved      */
       unsigned short int SVTMax;   /* Maximum SVT entries allowed       */
       unsigned short int SVTInUse; /* Current SVT usage                 */
       unsigned short int TCBMin;   /* Minimum TCB entries reserved      */
       unsigned short int TCBMax;   /* Maximum TCB entries allowed       */
       unsigned short int TCBInUse; /* Current TCB usage                 */
       unsigned short int PCBMin;   /* Minimum PCB entries reserved      */
       unsigned short int PCBMax;   /* Maximum PCB entries allowed       */
       unsigned short int PCBInUse; /* Current PCB usage                 */
       unsigned short int TimerMin; /* Minimum Timer blk. entries reserved */
       unsigned short int TimerMax; /* Maximum Timer blk. entries allowed  */
       unsigned short int TimerInUse; /* Current Timer blk. usage        */
       unsigned long int MEMMax;    /* Maximum Memory allowed            */
       unsigned long int MEMInUse;  /* Current Virtual Memory usage      */
     } QPRESBLOCK;
```

Figure 22-2. Query Process Resources - output buffer format

Procid	ID of the process whose system resource limits are being returned.
SVTMin	Number of SVT entries required by this process.

SVTMax	Maximum number of SVT entries that can be acquired by this process.
SVTInUse	Number of SVT entries that are in use by this process.
TCBMin	Number of TCB blocks required by this process.
TCBMax	Maximum number of TCBs that can be acquired by this process.
TCBInUse	Number of TCBs (tasks) that are in use by this process.
PCBMin	Number of PCB blocks required by this process.
PCBMax	Maximum number of PCBs that can be acquired by this process.
PCBInUse	Number of PCBs (processes) that are in use by this process.
TimerMin	Number of timer blocks required by this process.
TimerMax	Maximum number of timer blocks that can be acquired by this process.
TimerInUse	Number of timer blocks that are in use by this process.
MEMMax	Maximum amount (in bytes) of free storage that can be allocated to this process.
MEMInUse	Current virtual storage (in bytes) used by this process. If *MEMMax* for a process equals *QRMnomemmax* as defined in the header file QRM_CNST.H, then *MEMInUse* always equals zero for that process because the CP/Q Memory Manager has no need to track memory usage for that process.

Note: The data returned represents the state at the time the function was called. The usage counts usually change over time, but the maximum and minimum limits never change.

Implementation Notes

None

Return Codes

The Resource Manager provides a return code on all calls. A success return code *QRMgood* has the value 0; all other Resource Manager return codes are of the form **0x8003xxxx**. The possible Resource Manager return codes for this call, along with their low order 16 bits, are listed below.

QRMgood	Function completed.
QRMbadproc-(0x000E)	Invalid process specification in input registers.
QRMnowacc-(0x0029)	The caller does not have write access to the offset pointed to by *QPRBAddr*.

The Resource Manager calls the Memory Manager in order to complete this function. A return code other than *QMsuccess* from the Memory Manager is returned to the caller. Memory Manager return codes are in the form **0x8002xxxx**. The possible Memory Manager return codes for this call, along with their low order 16 bits, are listed below.

QMbad_offset-(0x0001)	Parameter list is invalid, for example *QPRBAddr* does not point into a valid memory object.

CPQueryProcSVIDs

```
Syntax -

#include "cpqlib.h"
#include "qrm_cnst.h"

     long int CPQueryProcSVIDs(ID,Mode,QPSBAddr,QPSBLen)

     unsigned long int ID;
     unsigned long int Mode;
     void *QPSBAddr;
     unsigned long int QPSBLen;

          ID       - Process ID or SVid (depends on Mode).
          Mode     - Used to determine process to be queried
                       and SVids to be returned (see below).
          QPSBAddr - Near pointer to output buffer.
          QPSBLen  - Length of buffer in bytes (4 bytes minimum).
```

Usage Notes

This function is used to obtain a list of the SVids that belong to the designated process, and is available to any requestor (independent of the relationship of the requestor process to the process being queried). Information is returned for named and unnamed processes.

The *Mode* parameter determines:

- The process to be queried.
- The types of SVids to return.

Mode values to determine the process to be queried include:

QRMproctask	Use process of caller.
QRMprocefftask	Use *effective* process of caller.
QRMprocsvcid	Use process to which the SVT entry specified in *ID* belongs.
QRMprocid	Use process whose process ID is in *ID*.

Mode values to determine the types of SVids to return include:

QRMqrytask	Return SVids of tasks.
QRMqrysersem	Return SVids of serialization semaphores.
QRMqrysynsem	Return SVids of synchronization semaphores.
QRMqryuitem	Return SVids of user items.
QRMqrymsgq	Return SVids of message queues.

Any or all *Mode* options to return SVids can be set to indicate the SVids to be returned. If none of the *Mode* options are set for returning SVids, then all SVids attached to the process queried are returned.

The returned information is written to the caller-supplied buffer area; it consists of an 4-byte header followed by as many SVids as will fit and are attached to the process. The number of items is computed by the Resource Manager from the size of the buffer (specified by the caller).

Output Buffer

The parameter *QPSBAddr* points to the output buffer whose structure is defined in cpqlib.h as QPSVIDBLOCK. Constant definition macros are defined in the Resource Manager user header file qrm_cnst.h.

```
Query Process SVIDs Return Block Format -

     typedef struct qpsvidblock {
       unsigned short int SVidCnt;  /* Number of SVids attached     */
       unsigned short int SVidRtn;  /* Number of SVids returned     */
       unsigned long int SVid[1];   /* First element of SVID array
     } QPSVIDBLOCK;
```

Figure 22-3. Query SVids Attached to a Process - output buffer

SVidCnt	Total number of SVT entries attached to the process indicated
SVidRtn	Number of SVids written into this buffer
SVid(*)	An array of SVids; the number of valid items written on return with a good return code is given in the field SVidRtn.

Implementation Notes

None

Return Codes

The Resource Manager provides a return code on all calls. A success return code *QRMgood* has the value 0; all other Resource Manager return codes are of the form **0x8003xxxx**. The possible Resource Manager return codes for this call, along with their low order 16 bits, are listed below.

QRMgood	Function completed.
QRMbadproc-(0x000E)	Invalid process specification in input registers.
QRMbadlength-(0x0015)	Output buffer length is not valid (length given is too small for header information to be returned.)
QRMnowacc-(0x0029)	The caller does not have write access to the offset pointed to by *QPSBAddr*.

The Resource Manager calls the Memory Manager in order to complete this function. A return code other than *QMsuccess* from the Memory Manager is returned to the caller. Memory Manager return codes are in the form **0x8002xxxx**. The possible Memory Manager return codes for this call, along with their low order 16 bits, are listed below.

QMbad_offset-(0x0001)	Parameter list is not valid, for example *QPSBAddr* does not point into a valid memory object.

Chapter 23. Set Process Functions

The functions described in this chapter allow a task to change the process resource defaults or to manipulate the effective process of a task. These functions are:

- **CPGiveEffProc -** Grant a specified task the privilege to change its effective process.
- **CPSetEffProc** - Manipulate the effective process field of the caller.
- **CPSetProcDfltLimit** - Change the values of the process default limits for PCBs, TCBs, Timers, SVTs, or memory storage.
- **CPSetProcMaxLimit -** Change the values of the process maximum limits for PCBs, TCBs, Timers, SVTs, or memory storage.

Function prototypes are listed in cpqlib.h. Applicable constant definition macros are defined in the file qrm_cnst.h.

CPGiveEffProc

```
Syntax -

#include "cpqlib.h"

     long int CPGiveEffProc(SVid)

     unsigned long int SVid;

          SVid     - SVid of the recipient task.
```

Usage Notes

This call is used to grant a specified task the privilege to change its own effective process. The recipient task can change or restore the *TCBeffproc* field (for example, to acquire memory in a different address space).

This call only completes successfully if the caller has set-effective-process privilege (the *TCBffenvpriv* value in the TCB must be set).

Implementation Notes

None.

Return Codes

The Resource Manager provides a return code on all calls. A success return code (QRMgood) has the value of 0; all other Resource Manager return codes are in the form **0x8003xxxx**. The possible Resource Manager return codes for this call, along with their low order 16 bits, are listed below.

QRMgood	Function completed.
QRMbadsvt-(0x000F)	Not a valid SVid in *SVid* or *SVid* is not the SVid of a task.
QRMillegal-(0x0019)	Caller does not have the privilege required for this function (that is, does not have set-effective-process privilege).

Note: A task obtains the privilege to change effective process, for attaching various resources to other processes, in three ways:

- The system builder specified to SLEEP that the *TCBffenvpriv* value is set in the PRIV field of the TASK command.
- A task which has this privilege creates a new task with this privilege.
- A task which has this privilege requests that the privilege also be assigned to another (already defined) task.

CPSetEffProc

```
Syntax -

#include "cpqlib.h"
#include "qrm_cnst.h"

     long int CPSetEffProc(ID,Mode)

     unsigned long int ID;
     unsigned long int Mode;

          ID       - Process ID or SVid (depends on Mode).
          Mode     - Used to determine effective process (see below).
```

Usage Notes

This call is used to manipulate the effective process field of the caller. The caller can change or restore his *TCBeffproc* field (for example, in order to acquire memory in a different address space).

The *Mode* parameter determines the effective process. *Mode* values include:

QRMproctask	Use process of caller.
QRMprocefftask	Use *effective* process of caller. This is effectively a null operation.
QRMprocsvcid	Use process to which the SVT entry specified in *ID* belongs.
QRMprocid	Use process whose process ID is in *ID*.

This call is always completed if the caller has set-effective-process privilege (the *TCBffenvpriv* value in the TCB must be set), unless the caller is not in the “SYSTEM” process and is attempting to change effective process to “SYSTEM.” Furthermore, even if the caller does not have the privilege in general, requests are honored if the proposed effective process is the caller's, or some descendant of the caller.

Implementation Notes

None.

Return Codes

The Resource Manager provides a return code on all calls. A success return code (QRMgood) has a value of 0; all other Resource Manager return codes are in the form **0x8003xxxx**. The possible Resource Manager return codes for this call, along with their low order 16 bits, are listed below.

QRMgood	Function completed.
QRMbadproc-(0x000E)	Invalid process specification.
QRMbadsvt-(0x000F)	Not a valid SVid in *SVid*, or *SVid* is not the SVid of a task.
QRMillegal-(0x0019)	Caller does not have the privilege required for this function (for example, does not have set-effective-process privilege), or caller is attempting to

change effective process to “SYSTEM” and is not in the “SYSTEM” process.

Note: A task obtains the privilege to change effective process, for attaching various resources to other processes, in three ways:

- The system builder specified to SLEEP that the *TCBffenvpriv* value is set in the PRIV field of the TASK command.
- A task which has this privilege creates a new task with this privilege.
- A task which has this privilege requests that the privilege also be assigned to another (already defined) task.

CPSetProcDfltLimit

```
Syntax -

#include "cpqlib.h"

      long int CPSetProcDfltLimit(ProcDefault,DefaultNum)

      unsigned long int ProcDefault;
      unsigned long int DefaultNumber;

           ProcDefault  - process default to set (see below).
           DefaultNum   - The new process default limit number.
```

Usage Notes

This call is used to change the process default limits that are stored in the System Data Area. The caller must have "SYSTEM" privilege (*TCBsyspriv* value in the TCB set) for this call to be successful.

The *ProcDefault* parameter determines the process default to set. Values include:

QRMsetPCBdflt
Set the process default limit for PCBs. The process default limit for PCBs must be less than or equal to the process maximum limit for PCBs.

QRMsetSVTdflt
Set the process default limit for SVTs. The process default limit for SVTs must be less than or equal to the process maximum limit for SVTs.

QRMsetTCBdflt
Set the process default limit for TCBs. The process default limit for TCBs must be less than or equal to the process maximum limit for TCBs.

QRMsetTimerdflt
Set the process default limit for timer blocks. The process default limit for timer blocks must be less than or equal to the process maximum limit for timer blocks.

QRMsetMEMdflt
Set the process default limit for memory storage. The process default limit for memory storage must be less than or equal to the process maximum limit for memory storage.

Implementation Notes

None.

Return Codes

The Resource Manager provides a return code on all calls. A success return code (QRMgood) has a value of 0; all other Resource Manager return codes are in the form **0x8003xxxx**. The possible Resource Manager return codes for this call, along with their low order 16 bits, are listed below.

QRMgood	Function completed.
QRMillegal-(0x0019)	Caller does not have the privilege required for this function (that is, does not have “SYSTEM” privilege)
QRMbadopt-(0x001A)	The value of the proposed process default passed to the Resource Manager is not valid. For example, the proposed process default is larger than the current process maximum limit for the specified resource.

CPSetProcMaxLimit

```
Syntax -

#include "cpqlib.h"

      long int CPSetProcMaxLimit(ProcDefault,DefaultNum)

      unsigned long int ProcDefault;
      unsigned long int DefaultNumber;

           ProcDefault  - process maximum to set (see below).
           DefaultNum   - The new process maximum limit number.
```

Usage Notes

This call is used to change the process maximum limits that are stored in the System Data Area. The caller must have “SYSTEM” privilege (*TCBsyspriv* value in the TCB set) for this call to be successful.

The process maximum limit defines the limit on the amount of a resource that can be specified by a process creation request. The process maximum limit must be equal to or greater than the corresponding process default limit for that resource.

The *ProcDefault* parameter determines the process maximum to set. Values include:

QRMsetPCBmax
: Set the process maximum limit for PCBs. A process maximum limit of *QRMnomax* means that no maximum limit is enforced.

QRMsetSVTmax
: Set the process maximum limit for SVTs. A process maximum limit of *QRMnomax* means that no maximum limit is enforced.

QRMsetTCBmax
: Set the process maximim limit for TCBs. A process maximum limit of *QRMnomax* means that no maximum limit is enforced.

QRMsetTimermax
: Set the process maximum limit for Timer blocks. A process maximum limit of *QRMnomax* means that no maximum limit is enforced.

QRMsetMEMmax
: Set the process maximum limit for memory storage. A process maximum limit of *QRMnomemmax* means that no maximum limit is enforced.

Implementation Notes

None

Return Codes

The Resource Manager provides a return code on all calls. A success return code (QRMgood) has a value of 0; all other Resource Manager return codes are in the form **0x8003xxxx**. The possible Resource Manager return codes for this call, along with their low order 16 bits, are listed below.

QRMgood	Function completed.
QRMillegal-(0x0019)	Caller does not have the privilege required for this function (that is, does not have “SYSTEM” privilege)
QRMbadopt-(0x001A)	The value of the proposed process maximum limit passed to the Resource Manager is not valid. For example, the proposed maximum limit passed to the Resource Manager is smaller than the current process default limit for that resource.

Chapter 24. Creating System Resources

The functions described in this chapter allow the creation of system-defined resources. The functions are:

- **CPCreateMsgQ** - Create a message queue.
- **CPCreateMsgQAcc** - Create a message queue with access options.
- **CPCreateProc** - Create a process.
- **CPCreateProcAcc** - Create a process with access options.
- **CPFork** - Fork a process.
- **CPCreateSerSem** - Create a serialization sempahore.
- **CPCreateSerSemAcc** - Create a serialization sempahore with access options.
- **CPCreateSyncSem** - Create a synchronization semaphore.
- **CPCreateSyncSemAcc** - Create a synchronization semaphore with access options.
- **CPCreateTask** - Create a task.
- **CPCreateTaskAcc** - Create a task with access options.
- **CPCreateTimerBlock** - Create a timer block.
- **CPCreateTimerBlockAcc** - Create a timer block with access options.
- **CPCreateUItem** - Create a user-defined SVT item.
- **CPCreateUItemAcc** - Create a user-defined SVT item with access options.
- **CPCreateUserInt** - Create (Install) a user-defined interrupt routine.
- **CPCreateUserIntAcc** - Create (Install) a user-defined interrupt routine with access options.

Objects are created by calling the appropriate Resource Manager create function, with an appropriate parameter list. Usually, the information required is either resource management information or else is used to define the access rights to SVids enforced by the SVC Handler.

Creator process and creator task

The “creator process” (defined in the parameter list) is the process to which the new object is to be attached. If a new process is being created, it is a child of the creator process.

The “creator task” is the caller of the create function.

Creation access checks

The task calling the create function must be related to the creator process in one of the following ways:

- The creator process is the same as, or is a descendant of, the caller's process.
- The creator process is the effective process of the caller.

If none of these conditions are met, the function cannot be completed and *QRMillegal* is returned to the caller.

Creating resource providers

A creator process must be a resource provider in order to have a child process be created as a resource provider. See the chapter titled "Resource Tracking" for more information on resource providers.

Resource limit checks

The resource limits enforced by the Resource Manager relate to the minimum, maximum, and in use PCB fields for TCBs, SVTs, PCBs, timer blocks, and storage.

The TCB, SVT,PCB and timer resource limit values are:

min	The number of entries required by the process. This number is used only at the time of process creation to preallocate the resources to the new process.
max	The absolute limit to the number of entries a process can acquire.
inuse	Number of entries currently in use by this process.

Note: TCBs and SVTs are counted separately. For example, if three TCBs must be reserved, the number of SVTs reserved should be three *plus* the number of non-task SVTs required.

Create functions fail if the *inuse* count is equal to the *max* count for the creator process, or if the *min* count for the object to be created cannot be preallocated by the system.

The system crashes if a resource cannot be allocated when *min* < *inuse*, because the Resource Manager could not create a resource for the caller that was preallocated.

Removable system objects

Any object that is created can be designated removable, whether or not the process to which it is attached is removable.

Function prototypes are listed in cpqlib.h. Structure definitions for applicable creation functions are listed in qrm_crea.h. Constant definition macros are defined in the file qrm_cnst.h.

CPCreateMsgQ

```
Syntax -

#include "cpqlib.h"

     long int CPCreateMsgQ(Name,RdAccess,WrAccess,SVid)

     char Name[8];
     unsigned long int RdAccess;
     unsigned long int WrAccess;
     unsigned long int *SVid;

       Name     - near pointer to 8 character name.
       RdAccess - 0 means no access restrictions enforced.
                  ProcID means only that process has read access.
       WrAccess - 0 means no access restrictions enforced.
                  ProcID means only that process has write access.
       SVid     - variable in which the message queue's SVid is returned.
```

Usage Notes

This function creates a message queue attached to the specified process. The creator process must be a descendant of that of the caller, or the caller's effective process. Once created, manipulation of elements in the queue can only be accomplished through the SVC Handler.

Implementation Notes

None.

Return Codes

The Resource Manager provides a return code on all calls. A success return code (QRMgood) has a value of 0; all other Resource Manager return codes are in the form **0x8003xxxx**. The possible Resource Manager return codes for this call, along with their low order 16 bits, are listed below.

QRMgood	Function completed.
QRMnosvt-(0x0002)	No free SVTs.
QRMsvtmax-(0x0008)	Creator process has maximum SVT allocation.
QRMbadname-(0x000D)	Duplicate name, the name contains invalid characters, or the object is named but the name is all blank.
QRMbadproc-(0x000E)	Not a valid creator process.
QRMbadsvt-(0x000F)	Not a valid *SVid* specified, or *SVid* is not the SVid of a task.
QRMprocremv-(0x0012)	The caller's effective process is being removed, request was denied.
QRMbadprocacc-(0x0016)	Accessing process is not valid.

QRMillegal-(0x0019)	Caller cannot attach objects to the designated process.
QRMforcedremv-(0x0023)	Function completed successfully, but resource created was forced to be removable.
QRMprocfork-(0x002C)	The caller's effective process is performing a process fork operation. Request was denied, try again later.
QRMprocexec-(0x002D)	The caller's effective process is preforming a program exec operation. Request was denied, try again later.
QRMprocproc-(0x002E)	The caller's effective process is performing a process create operation. Request was denied, try again later.
QRMproccreat-(0x002F)	The caller's effective process is itself being created. Request was denied, try again later.

The Resource Manager calls the Memory Manager in order to complete this function. A return code other than *QMsuccess* from the Memory Manager is returned to the caller. Memory Manager return codes are in the form **0x8002xxxx**. The possible Memory Manager return codes for this call, along with their low order 16 bits, are listed below.

QMbad_offset-(0x0001)	Parameter list is not valid.

The Resource Manager calls the SVC Handler in order to complete this function. A return code other than *QSVCgood* from the SVC Handler is returned to the caller. SVC Handler return codes are in the form **0x8001xxxx**. The possible SVC Handler return codes for this call, along with their low order 16 bits, are listed below.

QSVCduplnam-(0x0001)	Non-null name specified, which is not unique.

CPCreateMsgQAcc

```
Syntax -

#include "cpqlib.h"

     long int CPCreateMsgQAcc(ID,Mode,Name,RdAccess,WrAccess,SVid)

     unsigned long int ID;
     unsigned long int Mode;
     char Name[8];
     unsigned long int RdAccess;
     unsigned long int WrAccess;
     unsigned long int *SVid;

       ID       - process ID to which the message queue will belong
                  (depends on Mode).
       Mode     - determines creator process and access restrictions
                  (see below).
       Name     - near pointer to 8 character name.
       RdAccess - 0 means no read access restrictions enforced.
                  If either flags QRMcrtprocrd or QRMcrttaskrd are set in
                  the Mode field, then possible values for this field
                  are an SVid or process ID to which read access is to be
                  restricted.
       WrAccess - 0 means no write access restrictions enforced.
                  If either flags QRMcrtprocwr or QRMcrttaskwr are set in
                  the Mode field, then possible values for this field
                  are an SVid or process ID to which read access is to be
                  restricted.
       SVid     - variable in which the message queue's SVid is returned.
```

Usage Notes

This function creates a message queue, attached to the creator process specified by the *ID* value and the *Mode* parameter. The **CPCreateMsgQAcc** function differs slightly from the **CPCreateMsgQ** function in that more options are exposed to the caller, particularly, options dealing with ownership and access of the message queue. The caller must have privilege to attach objects to the creator process, that is, that caller can act on its own process, its own effective process, or any descendant of the caller.

The *Mode* parameter determines:

- The creator process that owns the message queue
- The access restrictions for the message queue

Mode values to determine the creator process include:

QRMproctask	Use process of caller
QRMprocefftask	Use *effective* process of caller
QRMprocsvcid	Use process to which the SVT entry specified in *ID* belongs
QRMprocid	Use process whose process ID is in *ID*

Mode values to determine the access restrictions upon the message queue include:

QRMcrtnamed	Create message queue as named.
QRMcrtremv	Create message queue as removable.
QRMcrtsuperrd	Set if read access to message queue is restricted to supervisor mode code.
QRMcrtprocrd	Set if read access to message queue is restricted to the process whose process ID is in *RdAccess*.
QRMcrttaskrd	Set if read access to message queue is restricted to the task whose SVid is in *RdAccess*.
QRMcrtsuperwr	Set if write access to message queue is restricted to supervisor mode code.
QRMcrtprocwr	Set if write access to message queue is restricted to the process whose process ID is in *WrAccess*.
QRMcrttaskwr	Set if write access to message queue is restricted to the task whose SVid is in *WrAccess*.

Message queues can be named or unnamed and can have access restrictions based on process or privilege level.

If the message queue is to be created as non-removable, the creator process to which the message queue is to be attached must also be non-removable.

The address space in which the message queue resides is maintained by the SVC Handler. Once created, manipulation of elements in the queue can only be accomplished through the SVC Handler.

Implementation Notes

None.

Return Codes

The Resource Manager provides a return code on all calls. A success return code (QRMgood) has a value of 0; all other Resource Manager return codes are in the form **0x8003xxxx**. The possible Resource Manager return codes for this call, along with their low order 16 bits, are listed below.

QRMgood	Function completed.
QRMnosvt-(0x0002)	No free SVTs.
QRMsvtmax-(0x0008)	Creator process has maximum SVT allocation.
QRMbadname-(0x000D)	Duplicate name, the name contains invalid characters, or the object is named but the name is all blank.
QRMbadproc-(0x000E)	Not a valid creator process.
QRMbadsvt-(0x000F)	Not a valid *SVid* specified, or *SVid* is not the SVid of a task.
QRMprocremv-(0x0012)	The caller's effective process is being removed. Request was denied.
QRMbadprocacc-(0x0016)	Accessing process is not valid
QRMillegal-(0x0019)	Caller cannot attach objects to the designated process.

QRMforcedremv-(0x0023)	Function completed successfully, but resource created was forced to be removable.
QRMprocfork-(0x002C)	The caller's effective process is performing a process fork operation. Request was denied, try again later.
QRMprocexec-(0x002D)	The caller's effective process is performing a program exec operation. Request was denied, try again later.
QRMprocproc-(0x002E)	The caller's effective process is performing a process create operation. Request was denied, try again later.
QRMproccreat-(0x002F)	The caller's effective process is itself being created. Request was denied, try again later.

The Resource Manager calls the Memory Manager in order to complete this function. A return code other than *QMsuccess* from the Memory Manager is returned to the caller. Memory Manager return codes are in the form **0x8002xxxx**. The possible Memory Manager return codes for this call, along with their low order 16 bits, are listed below.

QMbad_offset-(0x0001)	Parameter list is not valid.

The Resource Manager calls the SVC Handler in order to complete this function. A return code other than *QSVCgood* from the SVC Handler is returned to the caller. SVC Handler return codes are in the form **0x8001xxxx**. The possible SVC Handler return codes for this call, along with their low order 16 bits, are listed below.

QSVCduplnam-(0x0001)	Non-null name specified which is not unique.

CPCreateProc

```
Syntax -

#include "cpqlib.h"

     long int CPCreateProc(Name,Priobase,ProcOpts,
                           minpcb,minsvt,mintcb,mintimer,
                           maxpcb,maxsvt,maxtcb,maxtimer,maxmem,ProcID);

     char Name[8];
     unsigned long int Priobase;
     unsigned long int ProcOpts;
     unsigned long int minpcb;
     unsigned long int minsvt;
     unsigned long int mintcb;
     unsigned long int mintimer;
     unsigned long int maxpcb;
     unsigned long int maxsvt;
     unsigned long int maxtcb;
     unsigned long int maxtimer;
     unsigned long int maxmem;
     unsigned long int *ProcID;

       Name     - Near pointer to an (up to) 8-character name for the process.
       Priobase - Task dispatch priority offset.  This is a numeric value which,
                  when added to the base priority of the caller's effective process,
                  represents the most favored task dispatch priority allowed to tasks
                  in the new process being created.
       ProcOpts - Process option field.
                -  QRMcrtrp: if set, new process is a resource provider.
       minpcb   - Number of PCB entries preallocated to process.
       minsvt   - Number of SVT entries preallocated to process.
       mintcb   - Number of TCB entries preallocated to process.
       mintimer - Number of Timer entries preallocated to process.
       maxpcb   - Maximum number of PCB entries allowed to process.
       maxsvt   - Maximum number of SVT entries allowed to process.
       maxtcb   - Maximum number of TCB entries allowed to process.
       maxtimer - Maximum number of Timer entries allowed to process.
       maxmem   - Maximum amount of memory (in bytes) allowed to process.
       ProcID   - Variable which, on return, contains the process ID of the new process.
```

Usage Notes

This function is used to create only a new process. No initial task is created. The new process is created as a child of the caller's effective process, with the same base priority. If the *Priobase* field specified on the call has a nonzero value, and the tasks in the newly created process have a less favored priority. The new process is created as removable. The *QRMcrtrp* value can be set in *ProcOpts* only if the caller's effective process is also a resource provider.

The values *minpcb*, *minsvt*, *mintcb*, and *mintimer* specify the minimum number of PCB, SVT, TCB, and timer entries required by this new process. The system attempts to preallocate the minimum number of entries to the new process. If one of these fields is specified zero on the call, then no corresponding entry for that field is preallocated.

The values *maxpcb*, *maxsvt*, *maxtcb*, and *maxtimer* specify the maximum number of PCB, SVT, TCB, and timer entries allowed to this process. When any of these fields is specified zero (QRMdfltlimit) on the call, the corresponding process default limit for that field is used. When any of these fields is specified as -1 on the call, the corresponding current process maximum limit for that field is used. If the caller's task has SYSTEM privilege, the caller can specify -2 for any of these fields, which means that no limit is enforced upon that field in the new process.

The value *maxmem* specifies the maximum memory allowed to the newly created process (in bytes). To get the process default memory limit, specify a *maxmem* value of zero (QRMmemdfltlimit) on this call. To get the current process maximum memory limit, specify a *maxmem* value of -1 on this call. If the caller's task has SYSTEM privilege, the caller can specify a *maxmem* value of -2 on this call, which means that no limit is enforced when creating memory in the new process.

If the resource limits for PCBs, SVTs, TCBs or Timer blocks are exceeded, or if the system pool of control blocks is exhausted, the function fails.

For an explanation of resource provider processing during process creation, see *SPL Volume 5: Resource Manager*, "Resource Manager Tracking Notification Messages" chapter.

Implementation Notes

This create function differs from others, in that the created object is not necessarily usable without further processing. No initial tasks are created within the created PCB using this call. The caller must create tasks within the newly created PCB by calling the Resource Manager **CPCreateTask** function.

Return codes

The Resource Manager provides a return code on all calls. A success return code (QRMgood) has a value of 0; all other Resource Manager return codes are in the form **0x8003xxxx**. The possible Resource Manager return codes for this call, along with their low order 16 bits, are listed below.

QRMgood	Function completed.
QRMnopcb-(0x0001)	Number of free PCBs is insufficient.
QRMnosvt-(0x0002)	Number of free SVTs is insufficient.
QRMnotcb-(0x0003)	Number of free TCBs is insufficient.
QRMnotimer-(0x0004)	Number of timer blocks is insufficient.
QRMpcbmax-(0x0007)	Creator process has maximum PCB allocation.
QRMsvtmax-(0x0008)	Creator process has maximum SVT allocation.
QRMtcbmax-(0x0009)	Creator process has maximum TCB allocation.
QRMbadname-(0x000D)	Duplicate name, the name contains invalid characters, or the object is named but the name is all blank.
QRMbadproc-(0x000E)	Creator process is not valid.

QRMinvalid-(0x0010)	A process creation was attempted whereby the newly created process was to be a resource provider. However, the creator process is not a resource provider.
QRMprocremv-(0x0012)	The caller's effective process is being removed. Request was denied.
QRMillegal-(0x0019)	Caller does not have the privilege to create tasks from the designated creator process.
QRMforcedremv-(0x0023)	Function completed successfully, but resource created was forced to be removable.
QRMprocfork-(0x002C)	The caller's effective process is performing a process fork operation. Request was denied, try again later.
QRMprocexec-(0x002D)	The caller's effective process is performing a program exec operation. Request was denied, try again later.
QRMprocproc-(0x002E)	The caller's effective process is performing a process create operation. Request was denied, try again later.
QRMproccreat-(0x002F)	The caller's effective process is itself being created. Request was denied, try again later.

The Resource Manager calls the Memory Manager in order to complete this function. A return code other than *QMsuccess* from the Memory Manager is returned to the caller. Memory Manager return codes are in the form **0x8002xxxx**. The possible Memory Manager return codes for this call, along with their low order 16 bits, are listed below.

QMbad_offset-(0x0001)	Parameter list is not valid.
QMno_page-(0x0012)	Insufficient real storage is available to create the required page directory and page tables for the new process.

Resource Manager Generated Faults

It is possible that the Resource Manager faults the caller during processing of this call. If this occurs, the return code is transmitted to the caller's fault handler as a result of the Resource Manager issuing a **CPFaultTask** call. The cause or causes of the fault are listed below along with the error codes.

QRMxnopriv-(0x0081)	The caller attempted to create a process with unlimited resources, and did not have “SYSTEM” privilege.

Resource guarantees and limits, new process creation

SVTs, TCBs, PCBs and Timer Blocks

The minimum and maximum values for SVTs, TCBs, PCBs and Timer blocks when a new process is created are calculated from the parameter list and the system defaults. To describe the algorithm used, know these terms:

rqmax	Requested maximum for new process
rqmin	Requested minimum for new process
pdlmax	Process default limit
pmlmax	Process maximum limit

smlmax	System maximum limit
sysavail	Number of items available from the system pool

If *rqmax* is 0, use *pdlmax* instead.

If *rqmax* is -1 , then *rqmax* = *pmlmax*.

If *rqmax* is -2 , then *rqmax* = "unlimited".

If *rqmax* ≤ *rqmin*, then *rqmax* = *rqmin*.

Then, verify that the following conditions exist:

$$sysavail \geq rqmin$$
$$pmlmax \geq rqmax$$

If these relations are satisfied, the creation can proceed.

Memory Resources

The minimum and maximum limits for free storage requirements are handled as described under the memory-related fields in the parameter list. Although the Resource Manager controls the minimum, in use, and maximum PCB fields, the Memory Manager controls the in use and system available values, which vary over time. The Resource Manager reads these values to test for the conditions described with interrupts disabled.

CPCreateProcAcc

```
Syntax -

#include "cpqlib.h"

     long int CPCreateProcAcc(ID,Mode,Name,Priobase,ProcOpts,
                              minpcb,minsvt,mintcb,mintimer,
                              maxpcb,maxsvt,maxtcb,maxtimer,maxmem,ProcID);

     unsigned long int ID;
     unsigned long int Mode;
     char Name[8];
     unsigned long int Priobase;
     unsigned long int ProcOpts;
     unsigned long int minpcb;
     unsigned long int minsvt;
     unsigned long int mintcb;
     unsigned long int mintimer;
     unsigned long int maxpcb;
     unsigned long int maxsvt;
     unsigned long int maxtcb;
     unsigned long int maxtimer;
     unsigned long int maxmem;
     unsigned long int *ProcID;

       ID       - Process ID to which the newly created process will
                  belong (depends on Mode).
       Mode     - determines creator process and process characteristics.
       Name     - Near pointer to an (up to) 8-character name for the process.
       Priobase - Task dispatch priority offset.  This is a numeric value which,
                  when added to the base priority of the caller's effective process,
                  represents the most favored task dispatch priority allowed to tasks
                  in the new process being created.
       ProcOpts - Process option flags
                - QRMcrtrp -  new process is a resource provider.
       minpcb   - Number of PCB entries preallocated to process.
       minsvt   - Number of SVT entries preallocated to process.
       mintcb   - Number of TCB entries preallocated to process.
       mintimer - Number of Timer entries preallocated to process.
       maxpcb   - Maximum number of PCB entries allowed to process.
       maxsvt   - Maximum number of SVT entries allowed to process.
       maxtcb   - Maximum number of TCB entries allowed to process.
       maxtimer - Maximum number of Timer entries allowed to process.
       maxmem   - Maximum amount of memory (in bytes) allowed to process.
       ProcID   - Variable which, on return, contains the process ID of the new process.
```

Usage Notes

This function is used to create only a new process. No initial task is created. The new process is created as a child of the creator process, specified by the *ID* value and the *Mode* parameter with the same base priority. The **CPCreateProcAcc** function differs slightly from the **CPCreateProc** function in that more options are exposed to the caller, particularly, options dealing with ownership and characteristics of the new process. The caller must have privilege to attach objects to the creator process, that is, that caller can act on his own process, his own effective process, or any descendant of the caller.

The *Mode* parameter determines:

- The creator process that owns the newly created child process
- The process options for the created child process

Mode values to determine the creator process include:

QRMproctask	Use process of caller
QRMprocefftask	Use *effective* process of caller
QRMprocsvcid	Use process to which the SVT entry specified in *ID* belongs
QRMprocid	Use process whose process ID is in *ID*

Mode values to determine the access restrictions upon the created process include:

QRMcrtnamed	Create child process as named
QRMcrtremv	Create child process as removable

If the *Priobase* field specified on the call has a nonzero value, the tasks in the newly created process have a less favored priority. The new process can be created removable or non-removable. If the creator process is removable, the newly created process is created removable independent of setinng this flag. The resource provider flag *QRMcrtrp* can be set in *ProcOpts* only if the creator process is also a resource provider.

The values *minpcb*, *minsvt*, *mintcb*, and *mintimer* specify the minimum number of PCB, SVT, TCB, and timer entries required by this new process. The system attempts to preallocate the minimum number of entries to the new process. If one of these fields is specified zero on the call, then no corresponding entry for that field is preallocated.

The values *maxpcb*, *maxsvt*, *maxtcb*, and *maxtimer* specify the maximum number of PCB, SVT, TCB, and timer entries allowed to this process. When any field is specified as zero (QRMdfltlimit) on the call, the corresponding process default limit for that field is used. When any field is specified as -1 on the call, the corresponding current process maximum limit for that field is used. If the caller's task has SYSTEM privilege, the caller may specify -2 for any of these fields, which means that no limit is enforced upon that field in the new process.

The value *maxmem* specifies the maximum memory allowed to the newly created process (in bytes). To get the process default memory limit, specify a *maxmem* value of zero (QRMmemdfltlimit) on this call. To get the current process maximum memory limit, specify a *maxmem* value of -1 on this call. If the caller's task has SYSTEM privilege, the caller can specify a *maxmem* value of -2 on this call, which means that no limit is enforced when creating memory in the new process.

If the resource limits for PCBs, SVTs, TCBs or Timer blocks are exceeded, or if the system pool of control blocks is exhausted, the function fails.

The function fails if the memory required is not available.

Implementation Notes

None.

Return Codes

The Resource Manager provides a return code on all calls. A success return code (QRMgood) has a value of 0; all other Resource Manager return codes are in the form **0x8003xxxx**. The possible Resource Manager return codes for this call, along with their low order 16 bits, are listed below.

QRMgood	Function completed
QRMnopcb-(0x0001)	Number of free PCBs is insufficient
QRMnosvt-(0x0002)	Number of free SVTs is insufficient
QRMnotcb-(0x0003)	Number of free TCBs is insufficient
QRMnotimer-(0x0004)	Number of timer blocks is insufficient
QRMpcbmax-(0x0007)	Creator process has maximum PCB allocation
QRMsvtmax-(0x0008)	Creator process has maximum SVT allocation
QRMtcbmax-(0x0009)	Creator process has maximum TCB allocation
QRMbadname-(0x000D)	Duplicate name, the name contains invalid characters, or the object is named but the name is all blank.
QRMbadproc-(0x000E)	Creator process is not valid.
QRMinvalid-(0x0010)	A process creation was attempted whereby the newly created process was to be a resource provider. However, the creator process is not a resource provider.
QRMprocremv-(0x0012)	The caller's effective process is being removed. Request was denied.
QRMillegal-(0x0019)	Caller does not have the privilege to create tasks from the designated creator process.
QRMforcedremv-(0x0023)	Function completed successfully, but resource created was forced to be removable.
QRMprocfork-(0x002C)	The caller's effective process is performing a process fork operation. Request was denied, try again later.
QRMprocexec-(0x002D)	The caller's effective process is performing a program exec operation. Request was denied, try again later.
QRMprocproc-(0x002E)	The caller's effective process is performing a process create operation. Request was denied, try again later.
QRMproccreat-(0x002F)	The caller's effective process is itself being created. Request was denied, try again later.

The Resource Manager calls the Memory Manager in order to complete this function. A return code other than *QMsuccess* from the Memory Manager is returned to the caller. Memory Manager return codes are in the form **0x8002xxxx**. The possible Memory Manager return codes for this call, along with their low order 16 bits, are listed below.

QMbad_offset-(0x0001)	Parameter list is not valid.
QMno_page-(0x0012)	Insufficient real storage available to create the required page directory and page tables for the new process.

Resource Manager Generated Faults

It is possible that the Resource Manager can fault the caller during processing of this call. If this occurs, the return code is transmitted to the caller's fault handler as a result of the Resource Manager issuing a **CPFaultTask** call. The cause or causes of the fault are listed below along with the error codes.

QRMxnopriv-(0x0081)	The caller attempted to create a process with unlimited resources, and did not have “SYSTEM” privilege.

Resource guarantees and limits, new process creation

SVTs, TCBs, PCBs and Timer Blocks

The minimum and maximum values for SVTs, TCBs, PCBs and Timer blocks when a new process is created are calculated from the parameter list and the system defaults. To describe the algorithm used, use these terms:

rqmax	Requested maximum for new process
rqmin	Requested minimum for new process
pdlmax	Process default limit
pmlmax	Process maximum limit
smlmax	System maximum limit
sysavail	Number of items available from the system pool

If *rqmax* is 0, use *pdlmax* instead.

If *rqmax* is -1 , then *rqmax* = *pmlmax*.

If *rqmax* is -2 , then *rqmax* = "unlimited".

If $rqmax \leq rqmin$, then $rqmax = rqmin$.

Then, verify that the following conditions exist:

$$sysavail \geq rqmin$$
$$pmlmax \geq rqmax$$

If these relations are satisfied, the creation can proceed.

Memory Resources

The minimum and maximum limits for free storage requirements are handled as described under the memory-related fields in the parameter list. Although the Resource Manager controls the minimum, in use, and maximum PCB fields, the Memory Manager controls the in use and system available values, which vary over time. The Resource Manager reads these values to test for the conditions described with interrupts disabled.

CPCreateSerSem

```
Syntax -

#include "cpqlib.h"

     long int CPCreateSerSem(Name,Access,ClaimID,SVid)

     char Name[8];
     unsigned long int Access;
     unsigned long int ClaimID;
     unsigned long int *SVid;

       Name    - Near pointer to 8 character name.
       Access  - 0 means no access restrictions enforced.
                 ProcID means only that process has access.
       ClaimID - 0 means create as not claimed (that is, free).
                 Task SVid means create as claimed by that task.
       SVid    - Variable in which the serialization semaphore's
                        SVid is returned.
```

Usage Notes

This function creates a semaphore for serializing access to common data or other resources. This type of semaphore is called a serialization semaphore. The serialization semaphore is attached to the caller's effective process.

The serialization semaphore can exist in one of two states: “claimed” by a task, or “ not claimed” (that is, free). The following CP/Q system calls can manipulate and query the state of a serialization semaphore:

- CPSemClaim
- CPSemQuery
- CPSemRelease

Function descriptions for the above CP/Q system calls can be found in the "Resource/Serialization Semaphore Operations" chapter of *Application Programming Library Volume 1: Application Programming Reference*.

The semaphore can be created “pre-claimed” by a task by providing the task's SVid in the *ClaimID* argument. If the *ClaimID* field is zero, the serialization semaphore is created “not claimed.”

Implementation Notes

None.

Return Codes

The Resource Manager provides a return code on all calls. A success return code (QRMgood) has a value of 0; all other Resource Manager return codes are in the form **0x8003xxxx**. The possible Resource Manager return codes for this call, along with their low order 16 bits, are listed below.

QRMgood	Function completed.

QRMnosvt-(0x0002)	No free SVTs in system pool.
QRMsvtmax-(0x0008)	Creator process has maximum SVT allocation.
QRMbadname-(0x000D)	Duplicate name, the name contains invalid characters, or the object is named but the name is all blank.
QRMbadproc-(0x000E)	Creator process not valid.
QRMbadsvt-(0x000F)	*SVid* specified is not valid, or *SVid* is not the SVid of a task.
QRMprocremv-(0x0012)	The caller's effective process is being removed. Request was denied.
QRMbadprocacc-(0x0016)	Accessing process is not valid.
QRMillegal-(0x0019)	Caller cannot create objects in the specified process, or caller does not have the authority to establish the designated task as a semaphore owner.
QRMforcedremv-(0x0023)	Function completed successfully, but resource created was forced to be removable.
QRMprocfork-(0x002C)	The caller's effective process is performing a process fork operation. Request was denied, try again later.
QRMprocexec-(0x002D)	The caller's effective process is performing a program exec operation. Request was denied, try again later.
QRMprocproc-(0x002E)	The caller's effective process is performing a process create operation. Request was denied, try again later.
QRMproccreat-(0x002F)	The caller's effective process is itself being created. Request was denied, try again later.

The Resource Manager calls the Memory Manager in order to complete this function. A return code other than *QMsuccess* from the Memory Manager is returned to the caller. Memory Manager return codes are in the form **0x8002xxxx**. The possible Memory Manager return codes for this call, along with their low order 16 bits, are listed below.

QMbad_offset-(0x0001)	Parameter list is not valid.

The Resource Manager calls the SVC Handler in order to complete this function. A return code other than *QSVCgood* from the SVC Handler is returned to the caller. SVC Handler return codes are in the form **0x8001xxxx**. The possible SVC Handler return codes for this call, along with their low order 16 bits, are listed below.

QSVCduplnam-(0x0001)	SVC Handler detected duplicate SVT name.
QSVCdeadSVid-(0x000C)	the *ClaimID* field in the parameter list is non-zero and specifies the SVid of a task that is being removed from the system.

CPCreateSerSemAcc

```
Syntax -

#include "cpqlib.h"

     long int CPCreateSerSemAcc(ID,Mode,Name,Access,ClaimID,SVid)

     unsigned long int ID;
     unsigned long int Mode;
     char Name[8];
     unsigned long int Access;
     unsigned long int ClaimID;
     unsigned long int *SVid;

       ID       - Process ID to which the serialization semaphore will
                  belong (depends on Mode).
       Mode     - Determines creator process, access restrictions
                  and initial semaphore state (see below).
       Name    - Near pointer to 8 character name.
       Access  - 0 means no access restrictions enforced.
                  If either flags QRMcrtprocrd or QRMcrttaskrd are set in
                  the Mode field, then possible values for this field
                  are an SVid or process ID to which access is to be
                  restricted.
       ClaimID - Has meaning only if QRMcrtclaimed is set in Mode value.
                 0 means create as claimed by the caller.
                 Task SVid means create as claimed by that task.
       SVid    - Variable in which the serialization semaphore's
                        SVid is returned.
```

Usage Notes

This function creates a semaphore for serializing access to common data or other resources, attached to the creator process specified by the *ID* value and the *Mode* parameter. This type of semaphore is called a serialization semaphore. The **CPCreateSerSemAcc** function differs slightly from the **CPCreateSerSem** function in that more options are exposed to the caller, particularly, options dealing with ownership and access of the semaphore. The caller must have privilege to attach objects to the creator process, that is, that caller can act on its own process, its own effective process, or any descendant of the caller.

The *Mode* parameter determines:

- The creator process that owns the semaphore
- The access restrictions for the semaphore

Mode values to determine the creator process include:

QRMproctask	Use process of caller
QRMprocefftask	Use *effective* process of caller
QRMprocsvcid	Use process to which the SVT entry specified in *ID* belongs
QRMprocid	Use process whose process ID is in *ID*

Mode values to determine the access restrictions upon the semaphore include:

QRMcrtnamed	Create semaphore as named.
QRMcrtremv	Create semaphore as removable.
QRMcrtsuperrd	Set if access to semaphore is restricted to supervisor mode code.
QRMcrtprocrd	Set if access to semaphore is restricted to the process whose process ID is in *Access*.
QRMcrttaskrd	Set if read access to semaphore is restricted to the task whose SVid is in *Access*.
QRMcrtclaimed	Create semaphore claimed by the task SVid provided in the *ClaimID* argument. If *ClaimID* is zero when this flag is set, the semaphore is created claimed by the calling task.
QRMcrtnotclaimed	Create semaphore “not claimed” (free).

If neither *QRMcrtclaimed* nor *QRMnotclaimed* is specified in the *Mode* parameter at the time of the call, the serialization semaphore is created “not claimed.”

The serialization semaphore can exist in one of two states: “claimed” by a task, or “not claimed” (that is, free). The following CP/Q system calls can manipulate and query the state of a serialization semaphore:

- CPSemClaim
- CPSemQuery
- CPSemRelease

Function descriptions for the above CP/Q system calls can be found in the "Resource/Serialization Semaphore Operations" chapter of *Application Programming Library Volume 1: Application Programming Reference*.

The serialization semaphore can be created “pre-claimed” by a task by providing the task's SVid in the *ClaimID* argument. If the *ClaimID* field is zero, the serialization semaphore is created “not claimed.”

Serialization semaphores can be named or unnamed and can have access restrictions based on process or privilege level.

If the serialization semaphore is to be created as non-removable, the creator process to which the serialization semaphore is to be attached must also be non-removable.

Implementation Notes

None.

Return Codes

The Resource Manager provides a return code on all calls. A success return code (QRMgood) has a value of 0; all other Resource Manager return codes are in the form **0x8003xxxx**. The possible Resource Manager return codes for this call, along with their low order 16 bits, are listed below.

QRMgood	Function completed.
QRMnosvt-(0x0002)	No free SVTs in system pool.
QRMsvtmax-(0x0008)	Creator process has maximum SVT allocation.

QRMbadname-(0x000D)	Duplicate name, the name contains invalid characters, or the object is named but the name is all blank.
QRMbadproc-(0x000E)	Creator process is not valid.
QRMbadsvt-(0x000F)	*SVid* specified is not valid, or *SVid* is not the SVid of a task.
QRMprocremv-(0x0012)	The caller's effective process is being removed. Request was denied.
QRMbadprocacc-(0x0016)	Accessing process is not valid.
QRMillegal-(0x0019)	Caller cannot create objects in the specified process, or caller does not have the authority to establish the designated task as a semaphore owner.
QRMforcedremv-(0x0023)	Function completed successfully, but resource created was forced to be removable.
QRMprocfork-(0x002C)	The caller's effective process is performing a process fork operation. Request was denied, try again later.
QRMprocexec-(0x002D)	The caller's effective process is performing a program exec operation. Request was denied, try again later.
QRMprocproc-(0x002E)	The caller's effective process is performing a process create operation. Request was denied, try again later.
QRMproccreat-(0x002F)	The caller's effective process is itself being created. Request was denied, try again later.

The Resource Manager calls the Memory Manager in order to complete this function. A return code other than *QMsuccess* from the Memory Manager is returned to the caller. Memory Manager return codes are in the form **0x8002xxxx**. The possible Memory Manager return codes for this call, along with their low order 16 bits, are listed below.

QMbad_offset-(0x0001)	Parameter list is not valid.

The Resource Manager calls the SVC Handler in order to complete this function. A return code other than *QSVCgood* from the SVC Handler is returned to the caller. SVC Handler return codes are in the form **0x8001xxxx**. The possible SVC Handler return codes for this call, along with their low order 16 bits, are listed below.

QSVCduplnam-(0x0001)	SVC Handler detected duplicate SVT name.
QSVCdeadSVid-(0x000C)	the *ClaimID* field in the parameter list is non-zero and specifies the SVid of a task that is being removed from the system.

CPCreateSyncSem

```
Syntax -

#include "cpqlib.h"

     long int CPCreateSyncSem(Name,Access,State,SVid)

     Name[8];
     unsigned long int Access;
     unisgned long int State;
     unsigned long int *SVid;

       Name     - near pointer to 8 character name.
       Access   - 0 means no access restrictions enforced.
                  ProcID means only that process has access.
       State    - state of the semaphore upon creation.
                  0 means create as set.
                  ≠0 means create as cleared by the calling task.
       SVid     - variable in which the synchronization semaphore's
                        SVid is returned.
```

Usage Notes

This function creates a semaphore for synchronizing an event. This type of semaphore is called a synchronization semaphore. The synchronization semaphore is attached to the caller's effective process.

Synchronization semaphores allow tasks to wait for a single event to occur. Once this event has occurred, all the tasks waiting on the semaphore are made dispatchable.

The synchronization semaphore can exist in one of two states: “set” (that is, not clear) by a task, or “clear.” The following CP/Q system calls can manipulate the state of a synchronization semaphore:

- CPSemClear
- CPSemSet
- CPSemSetWait
- CPSemWait

Function descriptions for the above CP/Q system calls can be found in the "Synchronization Semaphore Operations" chapter of *Application Programming Library Volume 1: Application Programming Reference*.

The semaphore can be created “pre-cleared by the caller” by setting the *State* field to a non-zero value. A zero value in the *State* field requests the synchronization semaphore be created “set” upon creation.

Implementation Notes

None.

Return Codes

The Resource Manager provides a return code on all calls. A success return code (QRMgood) has a value of 0; all other Resource Manager return codes are in the form **0x8003xxxx**. The possible Resource Manager return codes for this call, along with their low order 16 bits, are listed below.

QRMgood	Function completed.
QRMnosvt-(0x0002)	No free SVTs in system pool.
QRMsvtmax-(0x0008)	Creator process has maximum SVT allocation.
QRMbadname-(0x000D)	Duplicate name, the name contains invalid characters, or the object is named but the name is all blank.
QRMbadproc-(0x000E)	Creator process is not valid.
QRMbadsvt-(0x000F)	*SVid* in parameter list is not valid, or *SVid* is not the SVid of a task.
QRMprocremv-(0x0012)	The caller's effective process is being removed. Request was denied.
QRMbadprocacc-(0x0016)	Accessing process is not valid.
QRMillegal-(0x0019)	Caller cannot create objects in the specified process, or caller does not have the authority to establish the designated task as a semaphore owner.
QRMforcedremv-(0x0023)	Function completed successfully, but resource created was forced to be removable.
QRMprocfork-(0x002C)	The caller's effective process is performing a process fork operation. Request was denied, try again later.
QRMprocexec-(0x002D)	The caller's effective process is performing a program exec operation. Request was denied, try again later.
QRMprocproc-(0x002E)	The caller's effective process is performing a process create operation. Request was denied, try again later.
QRMproccreat-(0x002F)	The caller's effective process is itself being created. Request was denied, try again later.

The Resource Manager calls the Memory Manager in order to complete this function. A return code other than *QMsuccess* from the Memory Manager is returned to the caller. Memory Manager return codes are in the form **0x8002xxxx**. The possible Memory Manager return codes for this call, along with their low order 16 bits, are listed below.

QMbad_offset-(0x0001)	Parameter list is not valid.

The Resource Manager calls the SVC Handler in order to complete this function. A return code other than *QSVCgood* from the SVC Handler is returned to the caller.

SVC Handler return codes are in the form **0x8001xxxx**. The possible SVC Handler return codes for this call, along with their low order 16 bits, are listed below.

QSVCduplnam-(0x0001) SVC Handler detected duplicate SVT name.

CPCreateSyncSemAcc

```
Syntax -

#include "cpqlib.h"

     long int CPCreateSyncSemAcc(ID,Mode,Name,Access,SVid)

     unsigned long int ID;
     unsigned long int Mode;
     Name[8];
     unsigned long int Access;
     unsigned long int *SVid;

       ID       - Process ID to which the synchronization semaphore will
                  belong (depends on Mode).
       Mode     - Determines creator process and access restrictions
                  (see below).
       Name     - Near pointer to 8 character name.
       Access   - 0 means no access restrictions enforced.
                  ProcID means only that process has access.
       SVid     - Variable in which the synchronization semaphore's
                        SVid is returned.
```

Usage Notes

This function creates a semaphore for synchronizing an event, attached to the creator process specified by the *ID* value and the *Mode* parameter. This type of semaphore is called a synchronization semaphore. The **CPCreateSyncSemAcc** function differs slightly from the **CPCreateSyncSem** function in that more options are exposed to the caller, particularly, options dealing with ownership and access of the synchronization semaphore. The caller of this function must have privilege to attach objects to the creator process, that is, that caller can act on his own process, his own effective process, or any descendant of the caller. The *Mode* parameter determines:

- The creator process that owns the semaphore.
- The access restrictions for the semaphore.

Mode values to determine the creator process include:

QRMproctask	Use process of caller
QRMprocefftask	Use *effective* process of caller
QRMprocsvcid	Use process to which the SVT entry specified in *ID* belongs
QRMprocid	Use process whose process ID is in *ID*

Mode values to determine the access restrictions upon the semaphore include:

QRMcrtnamed	Create semaphore as named.
QRMcrtremv	Create semaphore as removable.
QRMcrtsuperrd	Set if access to semaphore is restricted to supervisor mode code.
QRMcrtprocrd	Set if access to semaphore is restricted to the process whose process ID is in *Access*.

QRMcrttaskrd	Set if read access to semaphore is restricted to the task whose SVid is in *Access*.
QRMcrtset	Create synchronization semaphore as set.
QRMcrtclear	Create synchorization semaphore as cleared by the calling task.

Synchronization semaphores allow tasks to wait for a single event to occur. Once this event has occurred all the tasks waiting on the semaphore are made dispatchable.

The synchronization semaphore can exist in one of two states: “set” (that is, not clear) by a task, or “clear.” The following CP/Q system calls can manipulate the state of a synchronization semaphore:

- CPSemClear
- CPSemSet
- CPSemSetWait
- CPSemWait

Function descriptions for the above CP/Q system calls can be found in the "Synchronization Semaphore Operations" chapter of *Application Programming Library Volume 1: Application Programming Reference*.

The semaphore can be created (pre-cleared by the caller) by setting the *QRMcrtclear* flag in the *Mode* field. The semaphore can be created “set” (that is, not clear) by setting the *QRMcrtset* flag in the *Mode* field. If neither *QRMcrtclear* nor *QRMcrtset* is specified at the time of the call, the synchronization semaphore is created as “cleared” by the calling task.

Synchronization semaphores can be named or unnamed and can have access restrictions based on process or privilege level.

If the synchronization semaphore is to be created as non-removable, the creator process to which the semaphore is to be attached must also be non-removable.

Implementation Notes

None.

Return Codes

The Resource Manager provides a return code on all calls. A success return code (QRMgood) has a value of 0; all other Resource Manager return codes are in the form **0x8003xxxx**. The possible Resource Manager return codes for this call, along with their low order 16 bits, are listed below.

QRMgood	Function completed.
QRMnosvt-(0x0002)	No free SVTs in system pool.
QRMsvtmax-(0x0008)	Creator process has maximum SVT allocation.
QRMbadname-(0x000D)	Duplicate name, the name contains invalid characters, or the object is named but the name is all blank.
QRMbadproc-(0x000E)	Creator process is not valid.
QRMbadsvt-(0x000F)	*SVid* in parameter list is not valid, or *SVid* is not the SVid of a task.

QRMprocremv-(0x0012)	The caller's effective process is being removed. Request was denied.
QRMbadprocacc-(0x0016)	Accessing process is not valid.
QRMillegal-(0x0019)	Caller cannot create objects in the specified process, or caller does not have the authority to establish the designated task as a semaphore owner.
QRMforcedremv-(0x0023)	Function completed successfully, but resource created was forced to be removable.
QRMprocfork-(0x002C)	The caller's effective process is performing a process fork operation. Request was denied, try again later.
QRMprocexec-(0x002D)	The caller's effective process is performing a program exec operation. Request was denied, try again later.
QRMprocproc-(0x002E)	The caller's effective process is performing a process create operation. Request was denied, try again later.
QRMproccreat-(0x002F)	The caller's effective process is itself being created. Request was denied, try again later.

The Resource Manager calls the Memory Manager in order to complete this function. A return code other than *QMsuccess* from the Memory Manager is returned to the caller. Memory Manager return codes are in the form **0x8002xxxx**. The possible Memory Manager return codes for this call, along with their low order 16 bits, are listed below.

QMbad_offset-(0x0001)	Parameter list is not valid.

The Resource Manager calls the SVC Handler in order to complete this function. A return code other than *QSVCgood* from the SVC Handler is returned to the caller. SVC Handler return codes are in the form **0x8001xxxx**. The possible SVC Handler return codes for this call, along with their low order 16 bits, are listed below.

QSVCduplnam-(0x0001)	SVC Handler detected duplicate SVT name.

CPCreateTask

```
Syntax -

#include "cpqlib.h"

     long int CPCreateTask(Name,RdAccess,WrAccess,Priv,
                           FaultID,Priority,Preempt,SVid);

     char Name[8];
     unsigned long int RdAccess;
     unsigned long int WrAccess;
     unsigned long int Priv;
     unsigned long int FaultID;
     unsigned long int Priority;
     unsigned long int Preempt;
     unsigned long int *SVid;

       Name     - near pointer to 8 character name.
       RdAccess - 0 means no access restrictions enforced.
                  ProcID means only that process has read access to the message queue.
       WrAccess - 0 means no access restrictions enforced.
                  ProcID means only that process has write access to the message queue.
       Priv     - Privilege field for the created task (see below).
       FaultID  - 0 means use the fault handler of the caller's effective process.
                  Task SVid means use that task as the fault handler.
       Priority - Offset to be added to the base priority.
       Preempt  - 0 means preemptable.
                  1 means non-preemptable within priority level.
       SVid     - Variable in which the task's SVid is returned.
```

Usage Notes

This function is used to add a new task to the caller's effective process. The dispatch status of the new task is always “stopped, not initialized”; **CPPTrace**, Action = QSVC_write_regs, must be used to set up the task's registers and then **CPGoTask** must be used to begin execution.

Note: One way to read a program file into a new task is to use the CP/Q Loader function **LDLoadMod**, in which case the task registers are setup by the CP/Q Loader. The CP/Q Loader must be built into the system in order to use the **LDLoadMod** call.

The *Priv* parameter specifies the privilege or privileges to be given to the created task. The caller of this function must have at least the same set of task privileges as specified in this call. This means a task cannot be created with more privilege than the creator task. *Priv* values include:

QRMtsksupv	Created task is to be a supervisor task.
QRMtsksystem	Created task is to have “system” privilege.
QRMtskeffprc	Created task is allowed to change its effective process.
QRMtskfault	Created task is allowed to issue **CPFaultTask** calls.
QRMtskstopgo	Created task is allowed to issue **CPGoTask** and **CPStopTask** calls.

If the resource limits for TCBs, SVTs or Timer blocks are exceeded, or if the system pool of control blocks is exhausted, the function fails.

For explanation concerning resource provider processing during task creation, see *SPL Volume 5: Resource Manager*, "Resource Manager Tracking Notification Messages" chapter.

Implementation Notes

This create function differs from others in that the created object is not necessarily usable without further processing. If the returned SVid is nonzero, the caller has obtained a TCB.

The task cannot be run until the registers have been set up by calling the SVC Handler.

Return Codes

The Resource Manager provides a return code on all calls. A success return code (QRMgood) has a value of 0; all other Resource Manager return codes are in the form **0x8003xxxx**. The possible Resource Manager return codes for this call, along with their low order 16 bits, are listed below.

QRMgood	Function completed.
QRMnotimer-(0x0004)	No timer blocks available for task creation.
QRMsvtmax-(0x0008)	Creator process has maximum SVT allocation.
QRMtcbmax-(0x0009)	Creator process has maximum TCB allocation.
QRMbadname-(0x000D)	Duplicate name, the name contains invalid characters, or the object is named but the name is all blank.
QRMbadproc-(0x000E)	Creator process is not valid.
QRMbadsvt-(0x000F)	*SVid* in parameter list is not valid, or *SVid* is not the SVid of a task.
QRMprocremv-(0x0012)	The caller's effective process is being removed. Request was denied.
QRMbadprocacc-(0x0016)	Accessing process is not valid.
QRMillegal-(0x0019)	Caller does not have the privilege to create tasks from the designated creator process.
QRMbadopt-(0x001A)	Tasks do not have sufficient privilege (stop, fault) or min, max, privilege is not valid, priority, or preempt.
QRMforcedremv-(0x0023)	Function completed successfully, but resource created was forced to be removable.
QRMnosupvstk-(0x002A)	No supervisor mode stack available for task creation.
QRMprocfork-(0x002C)	The caller's effective process is performing a process fork operation. Request was denied, try again later.
QRMprocexec-(0x002D)	The caller's effective process is performing a program exec operation. Request was denied, try again later.

QRMprocproc-(0x002E)	The caller's effective process is performing a process create operation. Request was denied, try again later.
QRMproccreat-(0x002F)	The caller's effective process is itself being created. Request was denied, try again later.

The Resource Manager calls the Memory Manager in order to complete this function. A return code other than *QMsuccess* from the Memory Manager is returned to the caller. Memory Manager return codes are in the form **0x8002xxxx**. The possible Memory Manager return codes for this call, along with their low order 16 bits, are listed below.

QMbad_offset-(0x0001)	Parameter list is not valid.

The Resource Manager calls the SVC Handler in order to complete this function. A return code other than *QSVCgood* from the SVC Handler is returned to the caller. SVC Handler return codes are in the form **0x8001xxxx**. The possible SVC Handler return codes for this call, along with their low order 16 bits, are listed below.

QSVCduplnam-(0x0001)	SVC Handler detected duplicate SVT name.

CPCreateTaskAcc

```
Syntax -

#include "cpqlib.h"

     long int CPCreateTaskAcc(ID,Mode,Name,RdAccess,WrAccess,
                                   Priv,FaultID,Priority,Preempt,SVid);
     unsigned long int ID;
     unsigned long int Mode;
     char Name[8];
     unsigned long int RdAccess;
     unsigned long int WrAccess;
     unsigned long int Priv;
     unsigned long int FaultID;
     unsigned long int Priority;
     unsigned long int Preempt;
     unsigned long int *SVid;

       ID       - process ID to which the newly created task will belong (see Mode).
       Mode     - determines creator process and access restrictions
                  (see below).
       Name     - near pointer to 8 character name.
       RdAccess - 0 means no access restrictions enforced.
                  ProcID means only that process has read access to the message queue.
       WrAccess - 0 means no access restrictions enforced.
                  ProcID means only that process has write access to the message queue.
       Priv     - Privilege flags, see below.
       FaultID  - 0 means use the fault handler of the caller's effective process.
                  Task SVid means use that task as the fault handler.
       Priority - Offset to be added to the base priority.
       Preempt  - QRMpreempt means preemptable.
                  QRMnopreempt means non-preemptable within priority level.
       SVid     - Variable in which the task's SVid is returned.
```

Usage Notes

This function is used to add a new task to the creator process specified by the *ID* value and the *Mode* parameter. The **CPCreateTaskAcc** function differs slightly from the **CPCreateTask** function in that more options are exposed to the caller, particularly, options dealing with ownership and access of the created task. The caller must have privilege to attach objects to the creator process, that is, that caller can act on his own process, his own effective process, or any descendant of the caller.

The *Mode* parameter determines:

- The creator process that owns the created task
- The access restrictions for the created task

Mode values to determine the creator process include:

QRMproctask	Use process of caller.
QRMprocefftask	Use *effective* process of caller.
QRMprocsvcid	Use process to which the SVT entry specified in *ID* belongs.

QRMprocid	Use process whose process ID is in *ID*.

Mode values to determine the access restrictions upon the task include:

QRMcrtnamed	Create task as named.
QRMcrtremv	Create task as removable.
QRMcrtsuperrd	Set if read access to task is restricted to supervisor mode code.
QRMcrtprocrd	Set if read access to task is restricted to the process whose process ID is in *RdAccess*.
QRMcrttaskrd	Set if read access to task is restricted to the task whose SVid is in *RdAccess*.
QRMcrtsuperwr	Set if write access to task is restricted to supervisor mode code.
QRMcrtprocwr	Set if write access to task is restricted to the process whose process ID is in *WrAccess*.
QRMcrttaskwr	Set if write access to task is restricted to the task whose SVid is in *WrAccess*.

The dispatch status of the new task is always “stopped, not initialized”; **CPPTrace**, Action = QSVC_write_regs, must be used to set up the task's registers and then **CPGoTask** must be used to begin execution.

Note: One way to read a program file into a new task is to use the CP/Q Loader function **LDLoadMod**, in which case the task registers are setup by the CP/Q Loader. The CP/Q Loader must be built into the system in order to use the **LDLoadMod** call.

The Priv value is defined as follows:

QRMtsksupv	Created task is to be a supervisor task.
QRMtsksystem	Created task is to have “system” privilege.
QRMtskeffprc	Created task is allowed to change its effective process.
QRMtskfault	Created task is allowed to issue **CPFaultTask** calls.
QRMtskstopgo	Created task is allowed to issue **CPGoTask** and **CPStopTask** calls.

Tasks can be named or unnamed, and can have access restrictions based on process or privilege level.

If the task is to be created as non-removable, the creator process to which the task is to be attached must also be non-removable.

If the resource limits for TCBs, SVTs or Timer blocks are exceeded, or if the system pool of control blocks is exhausted, the function fails.

Implementation Notes

None.

Return Codes

The Resource Manager provides a return code on all calls. A success return code (QRMgood) has a value of 0; all other Resource Manager return codes are in the form **0x8003xxxx.** The possible Resource Manager return codes for this call, along with their low order 16 bits, are listed below.

QRMgood	Function completed.

QRMnotimer-(0x0004)	No timer blocks available for task creation.
QRMsvtmax-(0x0008)	Creator process has maximum SVT allocation.
QRMtcbmax-(0x0009)	Creator process has maximum TCB allocation.
QRMbadname-(0x000D)	Duplicate name, the name contains invalid characters, or the object is named but the name is all blank.
QRMbadproc-(0x000E)	Creator process is not valid.
QRMbadsvt-(0x000F)	*SVid* in parameter list is not valid, or *SVid* is not the SVid of a task.
QRMprocremv-(0x0012)	The caller's effective process is being removed. Request was denied.
QRMbadprocacc-(0x0016)	Accessing process is not valid.
QRMillegal-(0x0019)	Caller does not have the privilege to create tasks from the designated creator process.
QRMbadopt-(0x001A)	Tasks do not have sufficient privilege (stop, fault) or min, max, privilege is not valid, priority, or preempt.
QRMforcedremv-(0x0023)	Function completed successfully, but resource created was forced to be removable.
QRMnosupvstk-(0x002A)	No supervisor mode stack available for task creation.
QRMprocfork-(0x002C)	The caller's effective process is performing a process fork operation. Request was denied, try again later.
QRMprocexec-(0x002D)	The caller's effective process is performing a program exec operation. Request was denied, try again later.
QRMprocproc-(0x002E)	The caller's effective process is performing a process create operation. Request was denied, try again later.
QRMproccreat-(0x002F)	The caller's effective process is itself being created. Request was denied, try again later.

The Resource Manager calls the Memory Manager in order to complete this function. A return code other than *QMsuccess* from the Memory Manager is returned to the caller. Memory Manager return codes are in the form **0x8002xxxx**. The possible Memory Manager return codes for this call, along with their low order 16 bits, are listed below.

QMbad_offset-(0x0001)	Parameter list is not valid.

The Resource Manager calls the SVC Handler in order to complete this function. A return code other than *QSVCgood* from the SVC Handler is returned to the caller. SVC Handler return codes are in the form **0x8001xxxx**. The possible SVC Handler return codes for this call, along with their low order 16 bits, are listed below.

QSVCduplnam-(0x0001)	SVC Handler detected duplicate SVT name.

CPCreateTimerBlock

```
Syntax -

#include "cpqlib.h"

     long int CPCreateTimerBlock(TimerID)

     unsigned long int *TimerID;


          TimerID     - variable in which the Timer ID is returned.
```

Usage Notes

This function creates a timer block for the calling task, which is used for **CPTimerSet** and **CPTimerCancel** calls.

The process owning the calling task will be charged for the timer block resource.

Implementation Notes

None.

Return Codes

The Resource Manager provides a return code on all calls. A success return code (QRMgood) has a value of 0; all other Resource Manager return codes are in the form **0x8003xxxx**. The possible Resource Manager return codes for this call, along with their low order 16 bits, are listed below.

QRMgood	Function completed.
QRMnotimer-(0x0004)	No free timer blocks.
QRMtimermax-(0x000A)	The process owning the calling task has maximum timer block allocation.
QRMprocremv-(0x0012)	The caller's effective process is being removed. Request was denied.
QRMprocfork-(0x002C)	The caller's effective process is performing a process fork operation. Request was denied, try again later.
QRMprocexec-(0x002D)	The caller's effective process is performing a program exec operation. Request was denied, try again later.
QRMprocproc-(0x002E)	The caller's effective process is performing a process create operation. Request was denied, try again later.
QRMproccreat-(0x002F)	The caller's effective process is itself being created. Request was denied, try again later.

The Resource Manager calls the Memory Manager in order to complete this function. A return code other than *QMsuccess* from the Memory Manager is returned to the caller. Memory Manager return codes are in the form **0x8002xxxx**.

The possible Memory Manager return codes for this call, along with their low order 16 bits, are listed below.

QMbad_offset-(0x0001) Parameter list is not valid.

The Resource Manager calls the SVC Handler in order to complete this function. A return code other than *QSVCgood* from the SVC Handler is returned to the caller. SVC Handler return codes are in the form **0x8001xxxx**. The possible SVC Handler return codes for this call, along with their low order 16 bits, are listed below.

QSVCnotimers-(0x0028) No free timer blocks available.

CPCreateTimerBlockAcc

```
Syntax -

#include "cpqlib.h"

     long int CPCreateTimerBlockAcc(SVid,Mode,TimerID)

     unsigned long int SVid;
     unsigned long int Mode;
     unsigned long int *TimerID;


       SVid     - SVid of the task to which the timer block will
                  belong (depends on Mode).
       Mode     - determines creator task (see below).
       TimerID  - variable in which the Timer ID is returned.
```

Usage Notes

This function creates a timer block for the calling task, which is used for **CPTimerSet** and **CPTimerCancel** calls. The timer block is attached to the creator task specified by the *SVid* value and the *Mode* parameter. The **CPCreateTimerBlockAcc** function differs slightly from the **CPCreateTimerBlock** function in that more options are exposed to the caller, particularly, options dealing with ownership of the timer block. The caller must have privilege to attach objects to the creator process, that is, that caller can act on his own process, his own effective process, or any descendant of the caller.

The *Mode* parameter determines:

- The creator task that owns the timer block.

Mode values to determine the creator task include:

QRMproctask	Use task of caller.
QRMprocsvcid	Use task to which the SVT entry specified in *ID* belongs.

The process owning the creator task is charged for the timer block resource.

Implementation Notes

None.

Return Codes

The Resource Manager provides a return code on all calls. A success return code (QRMgood) has a value of 0; all other Resource Manager return codes are in the form **0x8003xxxx**. The possible Resource Manager return codes for this call, along with their low order 16 bits, are listed below.

QRMgood	Function completed.
QRMnotimer-(0x0004)	No free timer blocks.

QRMtimermax-(0x000A)	The process owning the calling task has maximum timer block allocation.
QRMprocremv-(0x0012)	The caller's effective process is being removed. Request was denied.
QRMprocfork-(0x002C)	The caller's effective process is performing a process fork operation. Request was denied, try again later.
QRMprocexec-(0x002D)	The caller's effective process is performing a program exec operation. Request was denied, try again later.
QRMprocproc-(0x002E)	The caller's effective process is performing a process create operation. Request was denied, try again later.
QRMproccreat-(0x002F)	The caller's effective process is itself being created. Request was denied, try again later.

The Resource Manager calls the Memory Manager in order to complete this function. A return code other than *QMsuccess* from the Memory Manager is returned to the caller. Memory Manager return codes are in the form **0x8002xxxx**. The possible Memory Manager return codes for this call, along with their low order 16 bits, are listed below.

QMbad_offset-(0x0001)	Parameter list is not valid.

The Resource Manager calls the SVC Handler in order to complete this function. A return code other than *QSVCgood* from the SVC Handler is returned to the caller. SVC Handler return codes are in the form **0x8001xxxx**. The possible SVC Handler return codes for this call, along with their low order 16 bits, are listed below.

QSVCnotimers-(0x0028)	No free timer blocks available.

CPCreateUItem

```
Syntax -

#include "cpqlib.h"

     long int CPCreateUItem(Name,RdAccess,WrAccess,SVid);

     char Name[8];
     unsigned long int RdAccess;
     unsigned long int WrAccess;
     unsigned long int *SVid;

       Name     - near pointer to 8 character name.
       RdAccess - 0 means no access restrictions enforced.
                  ProcID means only that process has read access.
       WrAccess - 0 means no access restrictions enforced.
                  ProcID means only that process has write access.
       SVid     - variable in which the user item's SVid is returned.
```

Usage Notes

This function creates a user-defined SVT item, attached to the caller's effective process.

Implementation Notes

None.

Return Codes

The Resource Manager provides a return code on all calls. A success return code (QRMgood) has a value of 0; all other Resource Manager return codes are in the form **0x8003xxxx**. The possible Resource Manager return codes for this call, along with their low order 16 bits, are listed below.

QRMgood	Function completed.
QRMnosvt-(0x0002)	No free SVTs.
QRMsvtmax-(0x0008)	Caller's effective process has maximum SVT allocation.
QRMbadname-(0x000D)	Duplicate name, the name contains invalid characters, or the object is named but the name is all blank.
QRMprocremv-(0x0012)	The caller's effective process is being removed. Request was denied.
QRMforcedremv-(0x0023)	Function completed successfully, but resource created was forced to be removable.
QRMprocfork-(0x002C)	The caller's effective process is performing a process fork operation. Request was denied, try again later.
QRMprocexec-(0x002D)	The caller's effective process is performing a program exec operation. Request was denied, try again later.

QRMprocproc-(0x002E)	The caller's effective process is performing a process create operation. Request was denied, try again later.
QRMproccreat-(0x002F)	The caller's effective process is itself being created. Request was denied, try again later.

The Resource Manager calls the Memory Manager in order to complete this function. A return code other than *QMsuccess* from the Memory Manager is returned to the caller. Memory Manager return codes are in the form **0x8002xxxx**. The possible Memory Manager return codes for this call, along with their low order 16 bits, are listed below.

QMbad_offset-(0x0001)	Parameter list is not valid.

The Resource Manager calls the SVC Handler in order to complete this function. A return code other than *QSVCgood* from the SVC Handler is returned to the caller. SVC Handler return codes are in the form **0x8001xxxx**. The possible SVC Handler return codes for this call, along with their low order 16 bits, are listed below.

QSVCduplnam-(0x0001)	Non-null name specified which is not unique.

CPCreateUItemAcc

```
Syntax -

#include "cpqlib.h"

     long int CPCreateUItemAcc(ID,Mode,Name,RdAccess,WrAccess,SVid);

     unsigned long int ID;
     unsigned long int Mode;
     char Name[8];
     unsigned long int RdAccess;
     unsigned long int WrAccess;
     unsigned long int *SVid;

       ID       - process ID to which the user-defined SVT item
                  will belong (depends on Mode).
       Mode     - determines creator process and access restrictions
                  (see below).
       Name     - near pointer to 8 character name.
       RdAccess - 0 means no access restrictions enforced.
                  ProcID means only that process has read access.
       WrAccess - 0 means no access restrictions enforced.
                  ProcID means only that process has write access.
       SVid     - variable in which the user item's SVid is returned.
```

Usage Notes

This function creates a user-defined SVT item, attached to the creator process specified by the *ID value* and the *Mode* parameter. The **CPCreateUItemAcc** function differs slightly from the **CPCreateUItem** function in that more options are exposed to the caller, particularly, options dealing with ownership and access of the user-defined SVT item. The caller must have privilege to attach objects to the creator process, that is, that caller can act on his own process, his own effective process, or any descendant of the caller.

The *Mode* parameter determines:

- The creator process that owns the SVT item
- The access restrictions for the SVT item

Mode values to determine the creator process include:

QRMproctask	Use process of caller.
QRMprocefftask	Use *effective* process of caller.
QRMprocsvcid	Use process to which the SVT entry specified in *ID* belongs.
QRMprocid	Use process whose process ID is in *ID*.

Mode values to determine the access restrictions upon the SVT item include:

QRMcrtnamed	Create SVT item as named.
QRMcrtremv	Create SVT item as removable.
QRMcrtsuperrd	Set if read access to SVT item is restricted to supervisor mode code.

QRMcrtprocrd	Set if read access to SVT item is restricted to the process whose process ID is in *RdAccess*.
QRMcrttaskrd	Set if read access to SVT item is restricted to the task whose SVid is in *RdAccess*.
QRMcrtsuperwr	Set if write access to SVT item is restricted to supervisor mode code.
QRMcrtprocwr	Set if write access to SVT item is restricted to the process whose process ID is in *WrAccess*.
QRMcrttaskwr	Set if write access to SVT item is restricted to the task whose SVid is in *WrAccess*.

User-defined SVT items can be named or unnamed and can have access restrictions based on process or privilege level.

If the SVT item is to be created as non-removable, the creator process to which the SVT item is to be attached must also be non-removable.

Calls to read from and write to the user-defined SVT item are provided by the SVC Handler.

Implementation Notes

None.

Return Codes

The Resource Manager provides a return code on all calls. A success return code (QRMgood) has a value of 0; all other Resource Manager return codes are in the form **0x8003xxxx**. The possible Resource Manager return codes for this call, along with their low order 16 bits, are listed below.

QRMgood	Function completed.
QRMnosvt-(0x0002)	No free SVTs.
QRMsvtmax-(0x0008)	Caller's effective process has maximum SVT allocation.
QRMbadname-(0x000D)	Duplicate name, the name contains invalid characters, or the object is named but the name is all blank.
QRMprocremv-(0x0012)	The caller's effective process is being removed. Request was denied.
QRMforcedremv-(0x0023)	Function completed successfully, but resource created was forced to be removable.
QRMprocfork-(0x002C)	The caller's effective process is performing a process fork operation. Request was denied, try again later.
QRMprocexec-(0x002D)	The caller's effective process is performing a program exec operation. Request was denied, try again later.
QRMprocproc-(0x002E)	The caller's effective process is performing a process create operation. Request was denied, try again later.
QRMproccreat-(0x002F)	The caller's effective process is itself being created. Request was denied, try again later.

The Resource Manager calls the Memory Manager in order to complete this function. A return code other than *QMsuccess* from the Memory Manager is returned to the caller. Memory Manager return codes are in the form **0x8002xxxx**. The possible Memory Manager return codes for this call, along with their low order 16 bits, are listed below.

QMbad_offset-(0x0001)	Parameter list is not valid.

The Resource Manager calls the SVC Handler in order to complete this function. A return code other than *QSVCgood* from the SVC Handler is returned to the caller. SVC Handler return codes are in the form **0x8001xxxx**. The possible SVC Handler return codes for this call, along with their low order 16 bits, are listed below.

QSVCduplnam-(0x0001)	Non-null name specified which is not unique.

CPCreateUserInt

```
Syntax -

#include "cpqlib.h"

     long int CPCreateUserInt(Offset,Func)
     void *Offset;
     unsigned long int *Func;

          Offset - The address of the function routine in global memory
                   that is to be the kernel extension.
          Func   - The specification for the function code to be used
                   for this user-defined kernel extension routine.
                   The variable Func may be specified in one of two
                   ways when calling CPCreateUserInt:
                     Func = 0 on the call requests that the Resource Manager
                              assign the function code to be used, and
                              returns the assigned function code in the
                              variable Func.
                     Func ≠ 0 on the call means that the caller is specifying
                              the desired function code to be used.
                              The value must be in the range QRMukernstart
                              to QRMukernend (constants contained within
                              include file QRM_CNST.H).
```

Usage Notes

This function installs a user-defined installable kernel extension routine, attached to the caller's effective process. The system extension routine is accessible via INT 7Bh assembler instruction, using the assigned function code as returned by this call.

Specifying Func = 0 (asking the Resource Manager to assign the function code) is the preferred usage when calling **CPCreateUserInt**, for reasons of flexibility. However, if the other method is used, wherein the caller specifies the desired function code, the caller should start requesting numbers from the top of the function code range (specified by the macro *QRMukernend* defined within the include file QRM_CNST.H) in order to avoid conflicts with function code numbers already assigned by the Resource Manager.

The function routine provided to the call may be a C or assembler function routine. Information about the stack structure passed into the function is provided under "Implementation Notes" on page 24-43.

The rules for a user defined interrupt routine are as follows:

- It must have a register interface (unless carefully coded to extract parameters off the stack).
- The function code to enter the interrupt 7Bh must be in AH.
- The return code must be passed back in EAX.

The value of *Offset* must be the address of a function located in the global memory class, meaning that the system extension code to be installed must be created in global memory so that all tasks have access to the system extension.

If a task calls an user-defined SVC function code that is not valid, the caller is faulted.

Implementation Notes

The user-defined installable kernel extension is intended to allow users to extend the capability of the kernel with their own customized SVC functions. A user-defined kernel extension (also called a user-defined SVC) has access to the kernel data areas and system control blocks because it is installed as another kernel service. A user-defined SVC can be dynamically installed by calling either the Resource Manager *CPCreateUserInt()* or *CPCreateUserIntAcc()* functions. Conversely, a existing user-defined SVC can be removed be calling the Resource Manager *CPDeleteUserInt()* function. A user-defined SVC can also be dynamically removed if the creator process owning the user-defined SVC is removed by a call to the Resource Manager *CPDeleteProc()* function.

Details about the function code range, stack layout, and return code sequence for user-defined installable kernel extensions are implementation defined as described below:

INTEL implementation

User-defined installable kernel extensions have a register interface with the function code in AH. The SVCs to these extensions are made by using an INT `X'7B'` instruction, a different numbered interrupt than is used by the SVC Handler, Memory Manager, and Resource Manager (which use INT `X'7A'`) so that there exists a completely separate set of function codes. There is also a near CALL entry point available for kernel extension SVCs.

Near CALL Kernel Extension SVCs: The near CALL takes the caller to the SVC Handler entry code. This pushes onto the stack all the registers, and then CALLs according to the function code in the AH register. This implies that the code for the SVC need not worry about saving any of the registers.

Thus when the routine for the kernel extension is entered, the stack is as shown in Figure 24-1 on page 24-44:

It is assumed that **every** routine that implements a kernel extension SVC, called by the above sequence, is written as a function (presumably in C), which returns a value (the return code) in the EAX register. When each of these routines returns (to the SVC Handler), the registers saved on the stack are restored, with the exception of the saved EAX value which is discarded. The SVC Handler then returns to the caller.

INT 7B Kernel Extension SVCs: The INT instruction causes the flags, EIP and CS registers to be placed on the stack. As for SVC Handler entry by an INT `X'7A'`, the return from the SVC Handler is in this case by an IRETD instruction. Hence, the code for the INT entry pushes the address of an IRETD instruction onto the stack, and then drops through to the code for the near CALL (the IRETD for INT `X'7A'` is also used for INT `X'7B'`). Having pushed the address of the IRETD, the INT SVC proceeds exactly as for the near CALL. The near RET instruction (to

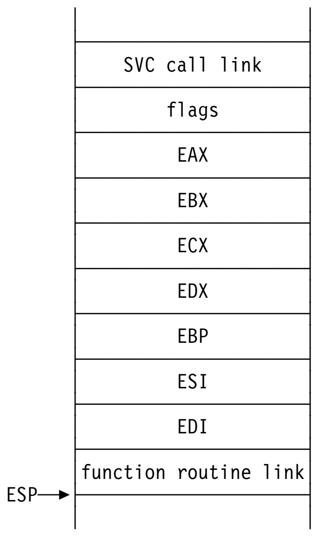


Figure 24-1. Stack Format for Near CALL Kernel Extension SVC

return to the caller for a near CALL SVC) jumps to the IRETD instruction, which in turn returns to the caller.

Thus for an INT kernel extension SVC, when the routine to implement it is entered, the stack is as shown in Figure 24-2 on page 24-45: The “old SS” and “old ESP” fields are present if and only if a privilege level transition occurred when the SVC was executed.

Warning:

1. The INT 7B function handler cannot change items in this stack frame above the first set of flags up from the bottom, that is the EIP, CS, flags, old ESP and old SS fields.
2. The INT 7B function handler cannot make nested or recursive SVCs, or there is a possibility of overrunning the end of the PL0 stack of a PL3 task.

Again, it is assumed that **every** routine to implement a kernel extension SVC, called by the above sequence, is written as a function (presumably in C), that returns a value (the return code) in the EAX register.

In certain cases, the kernel extension code for the SVC function needs to know the type of SVC entry (INT 7B or Near CALL), possibly for validation purposes by privilege level. This can be readily determined by looking at the stack frame. If the near CALL link on the stack is *not* the address of the IRETD instruction (this address is saved in the NDA, so the kernel extension code can determine what it is), then the SVC entry was by near CALL, and the caller is in supervisor mode. Otherwise, the SVC was by an INT instruction; in this case, the EIP and CS fields are present on the stack. If the least significant 2 bits of the CS field are both 1, then the caller is user mode (and the old SS and old ESP fields are also present); otherwise the caller is supervisor mode.

This information is also covered in the *Systems Programming Library Volume 3: SVC Handler* manual in the "CP/Q Kernel Supervisor Calls" chapter.

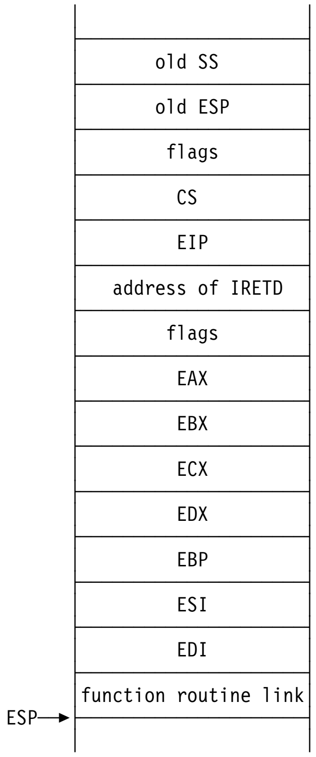


Figure 24-2. Stack Format for INT Kernel Extension SVC

Return Codes

The Resource Manager provides a return code on all calls. A success return code (QRMgood) has a value of 0; all other Resource Manager return codes are in the form **0x8003xxxx**. The possible Resource Manager return codes for this call, along with their low order 16 bits, are listed below.

QRMgood	Function completed.
QRMprocremv-(0x0012)	The caller's effective process is being removed. Request was denied.
QRMbadopt-(0x001A)	The function code specified was not within the range of allowable function codes.
QRMnofunc-(0x0024)	No more function codes available.
QRMnotglobal-(0x0025)	The value of Offset was not located in global memory.
QRMdupfunc-(0x0028)	Function code specified in call is already in use by another function.
QRMprocfork-(0x002C)	The caller's effective process is performing a process fork operation. Request was denied, try again later.
QRMprocexec-(0x002D)	The caller's effective process is performing a program exec operation. Request was denied, try again later.
QRMprocproc-(0x002E)	The caller's effective process is performing a process create operation. Request was denied, try again later.
QRMproccreat-(0x002F)	The caller's effective process is itself being created. Request was denied, try again later.

CPCreateUserIntAcc

```
Syntax -

#include "cpqlib.h"

     long int CPCreateUserIntAcc(ID,Mode,Offset,Func)

     unsigned long int ID;
     unsigned long int Mode;
     void *Offset;
     unsigned long int *Func;

       ID       - Process ID to which the kernel extension will belong
                  (depends on Mode).
       Mode     - Determines creator process (see below).
       Offset   - The address of the function routine in global memory
                  that is to be the kernel extension.
       Func     - The specification for the function code to be used
                  for this user-defined kernel extension routine.
                  The variable Func may be specified in one of two
                  ways when calling CPCreateUserIntAcc:
                    Func = 0 on the call requests that the Resource Manager
                             assign the function code to be used, and
                             returns the assigned function code in the
                             variable Func.
                    Func ≠ 0 on the call means that the caller is specifying
                             the desired function code to be used.
                             The value must be in the range
                             QRMukernstart to QRMukernend (constants
                             contained within include file QRM_CNST.H).
```

Usage Notes

This function installs a user-defined installable kernel extension routine, attached to the creator process specified by the *ID* value and the *Mode* parameter. The **CPCreateUserIntAcc** function differs slightly from the **CPCreateUserInt** function in that more options are exposed to the caller, particularly, options dealing with ownership of the kernel extension. The caller must have privilege to attach objects to the creator process, that is, that caller can act on his own process, his own effective process, or any descendant of the caller.

The *Mode* parameter determines:

- The creator process that owns the kernel extension.

Mode values to determine the creator process include:

QRMproctask	Use process of caller.
QRMprocefftask	Use *effective* process of caller.
QRMprocsvcid	Use process to which the SVT entry specified in *ID* belongs.
QRMprocid	Use process whose process ID is in *ID*.

The user interrupt routine is accessible via INT 7Bh assembler instruction, using the assigned function code as returned by this call.

Specifying Func = 0 (asking the Resource Manager to assign the function code) is the preferred usage when calling **CPCreateUserInt**, for reasons of flexibility. However, if the other method is used, wherein the caller specifies the desired function code, the caller should start requesting numbers from the top of the function code range (specified by the macro *QRMukernend* defined within the include file QRM_CNST.H) in order to avoid conflicts with function code numbers already assigned by the Resource Manager.

The function routine provided to the call may be a C or assembler function routine. Information about the stack structure passed into the function is provided under "Implementation Notes" on page 24-43.

The rules for a user defined interrupt routine are as follows:

- It must have a register interface (unless carefully coded to extract parameters off the stack).
- The function code to enter the interrupt 7Bh must be in AH.
- The return code must be passed back in EAX.

The value of *Offset* must be the address of a function located in the global memory class, meaning that the system extension code to be installed must be created in global memory so that all tasks have access to the system extension.

If a task calls an user-defined SVC function code that is not valid, the caller is faulted.

Implementation Notes

See "Implementation Notes" on page 24-43 for implementation information on user-installable kernel extensions.

Return Codes

The Resource Manager provides a return code on all calls. A success return code (QRMgood) has a value of 0; all other Resource Manager return codes are in the form **0x8003xxxx**. The possible Resource Manager return codes for this call, along with their low order 16 bits, are listed below.

QRMgood	Function completed.
QRMprocremv-(0x0012)	The caller's effective process is being removed. Request was denied.
QRMbadopt-(0x001A)	The function code specified was not within the range of allowable function codes.
QRMnofunc-(0x0024)	No more function codes available.
QRMnotglobal-(0x0025)	The value of Offset was not located in global memory.
QRMdupfunc-(0x0028)	Function code specified in call is already in use by another function.
QRMprocfork-(0x002C)	The caller's effective process is performing a process fork operation. Request was denied, try again later.

QRMprocexec-(0x002D)	The caller's effective process is performing a program exec operation. Request was denied, try again later.
QRMprocproc-(0x002E)	The caller's effective process is performing a process create operation. Request was denied, try again later.
QRMproccreat-(0x002F)	The caller's effective process is itself being created. Request was denied, try again later.

CPFork

```
Syntax -

#include "cpqlib.h"

     long int CPFork(SVid)

     unsigned long int *SVid;

       SVid     - variable in which the created task's SVid is returned.
```

Usage Notes

The **CPFork** system call creates a new process which is essentially a clone of the caller's process. In the case of a process with multiple tasks, only the task issuing the fork process call is cloned.

The new process created is a child of the process issuing the fork process call. Effective process settings are ignored.

Upon successful completion, both the calling task return from the **CPFork** call with a zero return code, and the *SVid* parameter set to the SVid of the created child task. The created child task returns with the return code *QRMforkchild*. The return code from **CPFork** distinguishes the caller from the created task.

The created child process inherits the following attribute from it's parent process:

- Address Space - the child gets the entire address space of the parent. However, a few points that must be noted:

  1. Ownership - All objects in the child process are owned by the child process regardless of the objects ownership in the parent process.
  2. Aliases - The memory object gets duplicated, but it is no longer an alias. This means it is marked an original object and is not linked on any alias chain. A physical copy is made.
  3. fixed objects - These are duplicated, but the object is not fixed in the child process. A physical copy must be made, as COW cannot be used, since we cannot detect if I/O changes the original instance of the object. **Note:** fixed objects are a potential problem as the real addresses of the original object are known as a result of the fix call. These are no longer valid for the child' copy.
  4. Common Objects - Shared objects continue to be shared. This means the child process is added to the pool of players sharing the object. Copy objects are copied as per the method specified when they were created.
  5. Copy on Write - COW is used wherever possible to perform the copying even for private objects.

  A number of the points above are required in order to enable the child process to terminate and be removed. It also enables the program exec function to remove unneeded memory objects.

- Environment - Child task inherits a copy of the caller's environment variables, heap, signal mask, signal stack state, and session ID.

- File descriptors - Child inherits a copy of the parent's open file descriptors, but they share a common file pointer for each file. This includes any session manager ones (assuming the CP/Q File System and CP/Q Session Manager are present in the system).
- Current Directory (assuming the CP/Q File System is present in the system).
- Root Directory - For current drive (assuming the CP/Q File System is present in the system).

Implementation Notes

A process and task are created for use by the clone. Upon entry to the **CPFork** call, the Resource Manager stops all tasks in the same process as the calling task so that tasks in the caller's process cannot alter memory while fork processing is going on. The process being forked has its PCB status field set to “process undergoing fork”

The Resource Manager creates the child process, setting the process status field of the created child process to “process is being created” The Resource Manager then calls the Memory Manager to create a new address space for the PCB. Upon successful completion, the Resource Manager creates a task in the created process and then call the Memory Manager once again to copy the contents of the caller's address space into the address space of the newly created process.

If no errors have occurred up to this point, the Resource Manager task commences asynchronous processing by resource providers who have registered for Notifications of Fork for the calling process. The RM task sends messages to resource providers to allow them to duplicate/share/inherit things as needed.

If all RPs complete successfully, RM will write the registers for the created task and then start the created child task. The PCB status fields of both the process containing the calling task and created process are set to “normal” The Resource Manager task sends out one-way notifications of process birth and task birth for the created process to SVids that have been registered for Notifications of Process Birth for the creator process and Notifications of Task Birth for the created child process.

If any resource provider fails, then process termination messages is sent for the created child process to all resource providers so they can “un-do” the fork; the Memory Manager would then be notified to remove the process' address space, and the Resource Manager would remove the created child task and process. The process status field of the process containing the calling task would be restored to “normal”

For explanation concerning resource provider processing during process fork, see *SPL Volume 5: Resource Manager*, "Resource Manager Tracking Notification Messages" chapter.

Return Codes

The Resource Manager provides a return code on all calls. A success return code (QRMgood) has a value of 0; all other Resource Manager return codes are in the form **0x8003xxxx**. The possible Resource Manager return codes for this call, along with their low order 16 bits, are listed below.

QRMgood	Function completed.

QRMnopcb-(0x0001)	Number of free PCBs is insufficient.
QRMnosvt-(0x0002)	Number of free SVTs is insufficient.
QRMnotcb-(0x0003)	Number of free TCBs is insufficient.
QRMnotimer-(0x0004)	Number of timer blocks is insufficient.
QRMpcbmax-(0x0007)	Creator process has maximum PCB allocation.
QRMsvtmax-(0x0008)	Creator process has maximum SVT allocation.
QRMtcbmax-(0x0009)	Creator process has maximum TCB allocation.
QRMforkchild-(0x000B)	Child task returning from **CPFork** call. This return code differentiates the parent task from the child task upon return from **CPFork**.
QRMbadname-(0x000D)	Duplicate name, the name contains invalid characters, or the object is named but the name is all blank.
QRMbadproc-(0x000E)	Creator process is not valid, or process is being removed.
QRMinvalid-(0x0010)	A process creation was attempted whereby the newly created process was to be a resource provider. However, the creator process is not a resource provider.
QRMprocremv-(0x0012)	Request was denied, creator process is being removed.
QRMillegal-(0x0019)	Caller does not have the privilege to create tasks from the designated creator process.
QRMnosupvstk-(0x002A)	No supervisor (PL0) mode stack available for task creation.
QRMprocfork-(0x002C)	Request was denied, creator process is already undergoing process fork. Try again later.
QRMprocexec-(0x002D)	Request was denied, creator process is undergoing program exec. Try again later.
QRMprocproc-(0x002E)	Request was denied, creator process is undergoing process creation. Try again later.
QRMproccreat-(0x002F)	Request was denied, process is being created. Try again later.

The Resource Manager calls the Memory Manager in order to complete this function. A return code other than *QMsuccess* from the Memory Manager is returned to the caller. Memory Manager return codes are in the form **0x8002xxxx**. The possible Memory Manager return codes for this call, along with their low order 16 bits, are listed below.

QMno_page-(0x0012)	Insufficient real storage available to create the required page directory and page tables for the new process.
QMwrap_count-(0x0021)	A common memory object is being replicated and there are already 255 process sharing this object.

Resource Manager Generated Faults

It is possible that the Resource Manager can fault the caller during processing of this call. If this occurs, the return code is transmitted to the caller's fault handler as a result of the Resource Manager issuing a **CPFaultTask** call. The cause or causes of the fault are listed below along with the error codes.

QRMxnopriv-(0x0081)	The caller with no limit on one of more resource maximums attempted to fork his process and did not have “SYSTEM” privilege.

Chapter 25. Removing System Resources

The functions described in this chapter allow the removal of system-defined resources. The functions are:

- **CPDelete** - removes a system object (task, semaphore, queue, and so on)
- **CPDeleteProc** - removes a process
- **CPDeleteTimer** - removes a timer block
- **CPDeleteUserInt** - removes a user-installable kernel extension

When performing calls asynchronously,the Resource Manager inserts the unique ID of the message requesting the removal into the message type field of the reply message. This allows the requestor to identify the reply message if the requestor is processing removals asynchronously.

Function prototypes are listed in cpqlib.h. Constant definition macros are defined in the file qrm_cnst.h.

CPDelete

```
Syntax -

#include "cpqlib.h"

     long int CPDelete(SVid,msg_idAddr)

     unsigned long int SVid;
     unsigned long int *msg_idAddr;

          SVid       - SVid of the object to delete.
          msg_idAddr - 0 means do the function synchronously.
                       Nonzero means do the function asynchronously,
                         and specifies the address of a variable in which
                         a msg_id is returned.  See discussion below.
```

Usage Notes

This function is used to delete system objects such as tasks, message queues, semaphores, and user-defined SVT items. The SVT entries for these items must have the “removable” value set otherwise this request fails. Also, the process to which the object being removed belongs must be the caller's process or effective process or a descendant of the caller's process. There are other restrictions based on what type of object is being removed.

The second argument to the function specifies whether the function is to be performed synchronously or asynchronously. When zero is specified as the second argument, the function does not return to the caller until it receives a response from the Resource Manager. When an address is specified as the second argument, the function returns immediately to the caller, without waiting for a response from the Resource Manager. For this asynchronous operation, the caller should then use the returned msg_id value in conjunction with a CPRecvMsg call to handle the eventual response from the Resource Manager.

Implementation Notes

When performing this call asynchronously, the Resource Manager inserts the unique ID of the message sent to the Resource Manager into the message type field of the reply message sent to the requestor. This allows the requestor to identify the reply message if the requestor is processing replies asynchronously.

Return Codes

The Resource Manager provides a return code on all calls. A success return code *QRMgood* has the value 0; all other Resource Manager return codes are of the form **0x8003xxxx**. The possible Resource Manager return codes for this call, along with their low order 16 bits, are listed below. 16 bits, are listed below.

QRMgood	Function completed.
QRMbadsvt-(0x000F)	The SVid in the parameter list is not valid.
QRMtwice-(0x0011)	A request to delete the object has already been received and is pending.

QRMprocremv-(0x0012)	The process containing the object is being removed. Request was denied.
QRMbadlength-(0x0015)	Parameter list too short.
QRMillegal-(0x0019)	The requestor does not have the authority to remove this object.
QRMbusy-(0x001E)	The object to be removed is a task which is a faulter or is being debugged.
QRMprocfork-(0x002C)	The process containing the object is performing a process fork operation. Request was denied, try again later.
QRMprocexec-(0x002D)	The process containing the object is performing a program exec operation. Request was denied, try again later.
QRMprocproc-(0x002E)	The process containing the object is performing a process create operation. Request was denied, try again later.
QRMproccreat-(0x002F)	The process containing the object is itself being created. Request was denied, try again later.

CPDeleteProc

```
Syntax -

#include "cpqlib.h"

     long int CPDeleteProc(ProcID,Flags,msg_idAddr)

     unsigned long int ProcID;
     unsigned long int Flags;
     unsigned long int *msg_idAddr;

          ProcID     - Determines the process to be deleted,
                       depending on the value in the Flags
                       field.
          Flags      - Flags field (see below).
          msg_idAddr - 0 means do the function synchronously.
                       Nonzero means do the function asynchronously,
                         and specifies the address of a variable in which
                         a msg_id is returned.  See discussion below.
```

Usage Notes

This function is used to delete a process from the system. It causes recovery processing to be initiated in the Resource Manager. The *ProcID* parameter indicates the process to remove depending on the value in the *Flags* parameter.

The *Flags* parameter determines:

- The process to be removed
- Process removal options

One or more of the *Flags* options might be set. Constant definition macros for *Flags* are defined in the include file qrm_cnst.h. The *Flags* field is defined as follows:

Process removal options:

QRMdel_pending	Set if process removal is to be pending. If the Resource Manager cannot remove the process right away for some reason (most likely, if there currently exist children processes), then the process is marked as “pending process removal” and *QRMpending* is returned to the caller by setting this option. At the point when all child processes have been removed, then process removal can continue. A second return code is not issued to the caller when the process pending removal is eventually removed. If this option is not set, the Resource Manager returns an error to the caller if the process to be removed has child processes.

The process to remove:

QRMproctask	Use process of caller
QRMprocefftask	Use *effective* process of caller
QRMprocsvcid	Use process to which the SVT entry specified in *ID* belongs.

QRMprocid	Use process whose process ID is in *ID*

The third argument to the function specifies whether the function is to be performed synchronously or asynchronously. When zero is specified as the second argument, the function does not return to the caller until it receives a response from the Resource Manager. When an address is specified as the second argument, the function returns immediately to the caller, without waiting for a response from the Resource Manager. For this asynchronous operation, the caller should then use the returned msg_id value in conjunction with a CPRecvMsg call to handle the eventual response from the Resource Manager.

See *SPL Volume 5: Resource Manager Tracking Notification Messages* for a detailed description of the processing performed during recovery and how to register to be notified when a process is being removed.

Implementation Notes

When performing this call asynchronously, the Resource Manager inserts the unique ID of the message sent to the Resource Manager into the message type field of the reply message sent to the requestor. This allows the requestor to identify the reply message if the requestor is processing replies asynchronously.

Return Codes

The Resource Manager provides a return code on all calls. A success return code *QRMgood* has the value 0; all other Resource Manager return codes are of the form **0x8003xxxx**. The possible Resource Manager return codes for this call, along with their low order 16 bits, are listed below.

QRMgood	Function completed.
QRMbadproc-(0x000E)	Specified processis not valid.
QRMinvalid-(0x0010)	The process is not removable. For example, the **QRMdel_pending** bit was not set, and the target process to be removed had child processes.
QRMprocremv-(0x0012)	A request to remove the process has already been received and is pending.
QRMillegal-(0x0019)	The requestor does not have the authority to remove this process.
QRMfailure-(0x001C)	A resource provider did not affirm removal of the process.
QRMnomsg-(0x0021)	Internal messaging failure, and no RQE was available for a tracking notification or message to a pending queue.
QRMprocfork-(0x002C) :287	The process to be deleted is performing a process fork operation. Request was denied, try again later.
QRMprocexec-(0x002D)	The process to be deleted is performing a program exec operation. Request was denied, try again later.
QRMprocproc-(0x002E)	The process to be deleted is performing a process create operation. Request was denied, try again later.
QRMproccreat-(0x002F)	The process to be deleted is itself being created. Request was denied, try again later.

QRMpending-(0x0030)	The process has been marked “removal pending”. The process is removed by the Resource Manager (when it is safe to do so).

CPDeleteTimer

```
Syntax -

     long int CPDeleteTimer(TimerID,msg_idAddr)

     unsigned long int TimerID;
     unsigned long int *msg_idAddr;

          TimerID    - Timer ID of the timer block to delete.
          msg_idAddr - 0 means do the function synchronously.
                       Nonzero means do the function asynchronously,
                         and specifies the address of a variable in which
                         a msg_id is returned.  See discussion below.
```

Usage Notes

This function is used to delete a timer block from the current task. The process that owns the task has its “timer in use” field decremented by one.

This function need not to be used in order to remove all allocated timer blocks from a task that is being removed. The Application Library function *CPDelete* deletes all allocated timer blocks when deleting a task.

The second argument to the function specifies whether the function is to be performed synchronously or asynchronously. When zero is specified as the second argument, the function does not return to the caller until it receives a response from the Resource Manager. When an address is specified as the second argument, the function returns immediately to the caller without waiting for a response from the Resource Manager. For this asynchronous operation, the caller should then use the returned msg_id value in conjunction with a CPRecvMsg call to handle the eventual response from the Resource Manager.

Implementation Notes

When performing this call asynchronously, the Resource Manager inserts the unique ID of the message sent to the Resource Manager into the message type field of the reply message sent to the requestor. This allows the requestor to identify the reply message if the requestor is processing replies asynchronously.

Return Codes

The Resource Manager provides a return code on all calls. A success return code *QRMgood* has the value 0; all other Resource Manager return codes are of the form **0x8003xxxx**. The possible Resource Manager return codes for this call, along with their low order 16 bits, are listed below.

QRMgood	Function completed.
QRMtwice-(0x0011)	A request to delete the object has already been received and is pending.
QRMprocremv-(0x0012)	The process containing the timer block is being removed. Request was denied.
QRMbadlength-(0x0015)	Parameter list too short.

QRMnomsg-(0x0021)	Internal messaging failure, no RQE was available for a tracking notification or message to a pending queue.
QRMbadtimer-(0x0026)	The *TimerID* specified is not valid or does not belong to the current task.
QRMprocfork-(0x002C)	The process containing the timer block is performing a process fork operation. Request was denied, try again later.
QRMprocexec-(0x002D)	The process containing the timer block is performing a program exec operation. Request was denied, try again later.
QRMprocproc-(0x002E)	The process containing the timer block is performing a process create operation. Request was denied, try again later.
QRMproccreat-(0x002F)	The process containing the timer block is itself being created. Request was denied, try again later.

CPDeleteUserInt

```
Syntax -

     long int CPDeleteUserInt(Func,msg_idAddr)

     unsigned long int Func;
     unsigned long int *msg_idAddr;

          Func       - Function code of the user defined interrupt
                       routine to delete.
          msg_idAddr - 0 means do the function synchronously.
                       Nonzero means do the function asynchronously,
                         and specifies the address of a variable in which
                         a msg_id is returned.  See discussion below.
```

Usage Notes

This function removes a user-defined interrupt routine (also called user-defined SVC) that was created by a call to *CPCreateUserInt()* or *CPCreateUserIntAcc()*. The user-defined interrupt routine being removed must be in the caller's process or effective process, or be a descendant of the caller's process.

The parameter *Func* specifies the user-defined SVC function code to remove. After this function code is removed, any tasks making calls to the user-defined SVC with this function code are faulted.

The second argument to the function specifies whether the function is to be performed synchronously or asynchronously. When zero is specified as the second argument, the function does not return to the caller until it receives a response from the Resource Manager. When an address is specified as the second argument, the function returns immediately to the caller, without waiting for a response from the Resource Manager. For this asynchronous operation, the caller should then use the returned msg_id value in conjunction with a CPRecvMsg call to handle the eventual response from the Resource Manager.

Implementation Notes

When performing this call asynchronously, the Resource Manager inserts the unique ID of the message sent to the Resource Manager into the message type field of the reply message sent to the requestor. This allows the requestor to identify the reply message if the requestor is processing replies asynchronously.

Return Codes

The Resource Manager provides a return code on all calls. A success return code *QRMgood* has the value 0; all other Resource Manager return codes are of the form **0x8003xxxx**. The possible Resource Manager return codes for this call, along with their low order 16 bits, are listed below.

QRMgood	Function completed.
QRMprocremv-(0x0012)	The process containing the user-defined interrupt routine is being removed. Request was denied.
QRMbadlength-(0x0015)	Parameter list too short.

QRMillegal-(0x0019)	The requestor does not have the authority to remove this object.
QRMnomsg-(0x0021)	Internal messaging failure, no RQE was available for a tracking notification or message to a pending queue.
QRMbadukern-(0x0027)	Specified function code is not valid.
QRMprocfork-(0x002C)	The process containing the user-defined interrupt routine is performing a process fork operation. Request was denied, try again later.
QRMprocexec-(0x002D)	The process containing the user-defined interrupt routine is performing a program exec operation. Request was denied, try again later.
QRMprocproc-(0x002E)	The process containing the user-defined interrupt routine is performing a process create operation. Request was denied, try again later.
QRMproccreat-(0x002F)	The process containing the user-defined interrupt routine is itself being created. Request was denied, try again later.

Chapter 26. Resource Tracking

The functions described in this chapter allow the caller to register interest in processes and to receive notification from the Resource Manager when particular events occur within a process. Such events include:

- Process birth
- Process creation
- Process fork
- Process removal
- Process death
- Program exec
- Task birth

These functions include:

- **CPTrkCancel** - This function cancels a tracking request previously made by calling **CPTrkTrkRequest**.
- **CPTrkCancelAll** - This function cancels a tracking request previously made by calling **CPTrkTrkRequestAll**.
- **CPTrkRequest** - This function requests notification when a particular event or set of events occur within a particular process.
- **CPTrkRequestAll** - This function requests notification when a particular event or set of events occur within any process in the system.

Function prototypes are listed in cpqlib.h. Structure definitions for applicable query functions are listed in qrm_trk.h. Constant definition macros are defined in the file qrm_cnst.h.

These functions are generally used by special processes called “resource providers,” which supply a service to client processes and need to know when a client disappears in order that they can clean up. The entire procedure is termed “resource tracking.”

CPTrkCancel

```
Syntax -

#include "cpqlib.h"
#include "qrm_cnst.h"

     long int CPTrkCancel(ID,SVid,Flags)

     unsigned long int ID;
     unsigned long int SVid;
     unsigned long int Flags;

          ID       - ID of the process being tracked via a call to
                     CPTrkRequest.
          SVid     - ID of the task or message queue (previously specified on a call
                     to CPTrkRequest) which was to receive the notification.
          Flags    - Flags field (see below)
```

Usage Notes

This function is used to undo one or more previous *CPTrkRequest* calls, for the event types selected. This function cancels a request for notification (to the specified task or message queue SVid), when a particular event or set of events occurs within the specified process.

The *Flags field* defines which event type or types to cancel tracking for.

One or more of the *Flags* options might be set. Constant definition macros for *Flags* are defined in the include file qrm_cnst.h. The *Flags* field is defined as follows:

Which event type or types to cancel tracking for:

QRMtrkprocrmv	Cancel notification of process removal of process *ProcID*.
QRMtrkprocdeath	Cancel notification of process death of process *ProcID*.
QRMtrkproccreat	Cancel notification of process creation of process *ProcID*.
QRMtrkprocbirth	Cancel notification of process birth of process *ProcID*.
QRMtrkprocfork	Cancel notification of process fork of process *ProcID*.
QRMtrktaskbirth	Cancel notification of task birth in process *ProcID*.
QRMtrkexec	Cancel notification of program exec in process *ProcID*.
QRMtrkcanall	Cancel all notification types for all processes that were being tracked by calls to *CPTrkRequest* that specified this particular *SVid*. **Note:** *ProcID* is ignored when this option is set.

The cancellation is for a particular ProcID and SVid pair, unless *QRMtrkcanall* is set within the *Flags* parameter. *QRMtrkcanall* specifies cancellation not only for all event types, but also for all previous calls to *CPTrkRequest* that used *SVid*.

Note: Requests for notification by process are discarded by the Resource Manager as part of process removal, so that this function should only be used to notify the Resource Manager that the caller is no longer interested in an existing process or processes.

Implementation Notes

None.

Return codes

The Resource Manager provides a return code on all calls. A success return code *QRMgood* has the value 0; all other Resource Manager return codes are of the form **0x8003xxxx.** The possible Resource Manager return codes for this call, along with their low order 16 bits, are listed below.

QRMgood	Function completed
QRMbadproc-(0x000E)	Specified *ProcID* is not valid or *ProcID* is the caller's process
QRMbadsvt-(0x000F)	Specified *SVid* is not valid or *SVid* is not in the caller's process
QRMnotfound-(0x0017)	Previous request not found

CPTrkCancelAll

```
Syntax -

#include "cpqlib.h"
#include "qrm_cnst.h"

     long int CPTrkCancelAll(SVid,Flags)

     unsigned long int SVid;
     unsigned long int Flags;

          SVid     - ID of the task or message queue (previously specified on a call
                     to CPTrkRequestAll) which was to receive the notification.
          Flags    - Flags field (see below)
```

Usage Notes

This function is used to undo all or a part of a previous *CPTrkRequestAll* call that specified the same *SVid*. This function cancels a request for notification (to the specified task or message queue SVid), when a particular event or set of events occurs within any process in the system.

The *Flags field* defines which event type or types to cancel tracking for. One or more of the *Flags* options can be set. Constant definition macros for *Flags* are defined in the include file qrm_cnst.h. The *Flags* field is defined as follows:

QRMtrkprocrmv	Cancel notification of process removals to *SVid*.
QRMtrkprocdeath	Cancel notification of process deaths to *SVid*.
QRMtrkproccreat	Cancel notification of process creations to *SVid*.
QRMtrkprocbirth	Cancel notification of process births to *SVid*.
QRMtrkprocfork	Cancel notification of process forks to *SVid*.
QRMtrktaskbirth	Cancel notification of task births to *SVid*.
QRMtrkexec	Cancel notification of program execs to *SVid*.
QRMtrkcanall	Cancel all notification types for all processes that were being tracked by calls to *CPTrkRequestAll* that specified this particular *SVid*.

If *QRMtrkcanall*is set, it expands the cancellation to include all the event types, effectively cancelling all of the effects of corresponding previous calls to *CPTrkCancelAll* that used *SVid*.

Implementation Notes

None.

Return codes

The Resource Manager provides a return code on all calls. A success return code *QRMgood* has the value 0; all other Resource Manager return codes are of the form **0x8003xxxx**. The possible Resource Manager return codes for this call, along with their low order 16 bits, are listed below.

QRMgood	Function completed.

QRMbadsvt-(0x000F)	Specified *SVid* not valid, or *SVid* is not in the caller's process.
QRMnotfound-(0x0017)	Previous request not found.

CPTrkRequest

```
Syntax -

#include "cpqlib.h"
#include "qrm_cnst.h"

     long int CPTrkRequest(ID,SVid,Data1,Data2,Flags)

     unsigned long int ID;
     unsigned long int SVid;
     unsigned long int Data1;
     unsigned long int Data2;
     unsigned long int Flags;

          ID       - ID of the process being tracked.
          SVid     - ID of the task or message queue which
                     is to receive the notification.
          Data1    - User data word; returned in process tracking
                     notification message.
          Data2    - User data word; returned in process tracking
                     notification message.
          Flags    - Flags field (see below)
```

Usage Notes

This function requests that the specified *SVid* (a task or message queue) be sent a message by the Resource Manager when a particular event occurs for a specified process, *ProcID*. Such events include process birth, process creation, process fork, process removal, process death, program exec, or task birth.

The *Flags field* defines which event type or types to track.

One or more of the *Flags* options can be set. Constant definition macros for *Flags* are defined in the include file qrm_cnst.h. The *Flags* field is defined as follows:

Which event type or types to track:

QRMtrkprocrmv	Request notification of process removal of process *ProcID*.
QRMtrkprocdeath	Request notification of process death of process *ProcID*.
QRMtrkproccreat	Request notification of process creation of process *ProcID*.
QRMtrkprocbirth	Request notification of process birth of process *ProcID*.
QRMtrkprocfork	Request notification of process fork of process *ProcID*.
QRMtrktaskbirth	Request notification of task birth in process *ProcID*.
QRMtrkexec	Request notification of program exec in process *ProcID*.

The calling process must have resource provider privilege in order to issue requests for notification of the following events called two-way notifications:

- Process removal
- Process creation
- Process fork
- Program exec

Also, for these two-way event notifications, the calling process must reply to any notification message that it receives before the event in progress can continue. The process being tracked cannot be the caller's process when requesting two-way notifications.

The remaining system events (task birth, process birth, and process death) are one-way notifications that do not require resource provider privilege and should not be replied to. The process being tracked can be the caller's process when requesting one-way notifications only.

If a previous request has been received for the same process/SVid combination, the new request replaces the old.

Implementation Notes

None.

Return codes

The Resource Manager provides a return code on all calls. A success return code *QRMgood* has the value 0; all other Resource Manager return codes are of the form **0x8003xxxx**. The possible Resource Manager return codes for this call, along with their low order 16 bits, are listed below.

QRMgood	Function completed.
QRMbadproc-(0x000E)	Specified *ProcID* is not valid, or a two-way notification was requested for the caller's process.
QRMbadsvt-(0x000F)	Specified *SVid* not valid, or *SVid* is not in the caller's process.
QRMinvalid-(0x0010)	Notification requested, but caller's process is not a resource provider.
QRMnonrb-(0x001F)	No free NRBs.
QRMglobalnrb-(0x0022)	A global request for the same SVid (to be notified) already exists.

CPTrkRequestAll

```
Syntax -

#include "cpqlib.h"
#include "qrm_cnst.h"

     long int CPTrkRequestAll (unsigned long int SVid,
                                 unsigned long int Data1,
                                 unsigned long int Data2,
                                 unsigned long int Flags)

          SVid     - ID of the task or message queue which
                     is to receive the notification.
          Data1    - User data word; returned in process tracking
                     notification message.
          Data2    - User data word; returned in process tracking
                     notification message.
          Flags    - Flags field (see below)
```

Usage Notes

This function requests that the specified *SVid* (either a task or message queue) be sent a message by the Resource Manager when a particular event occurs for any process in the system. Such events include program:

- Exec
- Process birth
- Process creation
- Process fork
- Process removal
- Process death
- Task birth

The *Flags field* defines which event type or types to track. One or more of the *Flags* options can be set. Constant definition macros for *Flags* are defined in the include file qrm_cnst.h. The *Flags* field is defined as follows:

QRMtrkprocrmv	Request notification of all process removals.
QRMtrkprocdeath	Request notification of all process deaths.
QRMtrkproccreat	Request notification of all process creations.
QRMtrkprocbirth	Request notification of all process births.
QRMtrkprocfork	Request notification of all process forks.
QRMtrktaskbirth	Request notification of all task births.
QRMtrkexec	Request notification of all program execs.

The calling process must have resource provider privilege in order to issue requests for notification of the following events called two-way notifications:

- Process removal
- Process creation
- Process fork
- Program exec

Also, for these two-way event notifications, the calling process must reply to any notification message that it receives before the event in progress can continue. The process being tracked cannot be the caller's process when requesting two-way notifications.

The remaining system events (task birth, process birth, and process death) are called one-way notifications that do not require resource provider privilege and should not be replied to. The process being tracked can be the caller's process when requesting one-way notifications only.

If a previous request has been received for the same *SVid*, the new request replaces the old.

Implementation Notes

None.

Return codes

The Resource Manager provides a return code on all calls. A success return code *QRMgood* has the value 0; all other Resource Manager return codes are of the form **0x8003xxxx**. The possible Resource Manager return codes for this call, along with their low order 16 bits, are listed below.

QRMgood	Function completed.
QRMbadsvt-(0x000F)	Specified *SVid* is not valid or *SVid* is not in the caller's process.
QRMinvalid-(0x0010)	Notification requested, but caller's process is not a resource provider.
QRMnonrb-(0x001F)	No free NRBs.

Appendices and Glossary

Appendix A. CP/Q SVC Handler Summary

SVC Functions

The following table is a list of all SVC Handler requests. In this table, "SVC" is the symbolic name of the SVC; "constant name" is the name of the program constant that defines the value to be placed in the AH register for that request. Programs should never code in the value of these constants (they should use the names), since it is quite possible that these values may be changed.

Each request is described in detail in the previous sections of this manual.

Table A-1 (Page 1 of 3). SVC Handler Function Codes

SVC name	Routine name	Constant name	#	Description
SVC_SVC	CPSVC	-	-	Issue any SVC Handler SVC (not needed on POWER architecture, not an SVC)
DISP_RET	CPYield	QSVCDSPRT	0	Yield processor (return to dispatcher), do not suspend
DISP_RET2	CPShortYield	QSVCDSPRT2	1	Yield processor (return to dispatcher), do not suspend
INT_WAIT	CPIntWait	QSVCIWAIT	2	Wait for a hardware interrupt
MUX_WAIT	CPMuxWait	QSVCMUXWT	3	Wait on a list of semaphores/queues
-	-	-	4	Not used
SEM_WAIT	CPSemWait	QSVCSEMWT	5	Wait for a synchronization semaphore to clear
SEM_CLEAR	CPSemClear	QSVCSEMCLR	6	Clear a synchronization semaphore
SEM_SET	CPSemSet	QSVCSEMSET	7	Set a synchronization semaphore
SEM_SETWAIT	CPSemSetWait	QSVCSEMSTWT	8	Set a synchronization semaphore, wait for it to clear
-	-	-	9	Not used
SEND_MESG	CPSendMsg	QSVCSEND	10	Send a message to another task or to a queue
SPLIT_SEND	CPSplitMsg	QSVCSPLIT	11	Send a message with a "split" buffer area to another task or to a queue
RECV_MESG	CPRecvMsg	QSVCRECV	12	Receive a message from a queue
SPLIT_RECV	CPSplitRecv	QSVCRCVSPLT	13	Receive a message from a queue, using a "split" buffer.
PEEK_MESG	CPPeekMsg	QSVCPEEK	14	"peek" a message from a queue without receiving it
COUNT_MESG	CPCountMsg	QSVCCNTMSG	15	Obtain the count of outstanding messages on a queue
SEND_RECV	CPSendRecvMsg	QSVCSENDRCV	16	Send a message and await a reply.
-	-	-	17	Not used
CLAIM_SEM	CPSemClaim	QSVCCLAIM	18	Claim a semaphore
REL_SEM	CPSemRelease	QSVCRLSEM	19	Free (release) a semaphore
QRY_SEM	CPSemQuery	QSVCQSEM	20	Query a semaphore
-	-	-	21	Not used
GET_ID	CPResolveName	QSVCGETID	22	Gets the SVid corresponding to an SVT item name.
ID_NAME	CPResolveID	QSVCIDNAME	23	Obtain the SVT name corresponding to an SVid
QRY_ID	CPQueryID	QSVCQRYID	24	Obtain the caller's SVid and process ID.

Table A-1 (Page 2 of 3). SVC Handler Function Codes

SVC name	Routine name	Constant name	#	Description
TRAN_ID	CPSVid2TCB CPTCB2SVid	QSVCTRANID	25	Translate TCB id into SVid and process ID.
CHECK_ID	CPCheckID	QSVCCHKID	26	Verify caller has access to an SVid.
-	-	-	27	Not used
-	-	-	28	Not used
-	-	-	29	Not used
GO_TASK	CPGoTask	QSVCSTART	30	Mark a task as started, that is the task is no longer “stopped”
STOP_TASK	CPStopTask	QSVCSTOP	31	Mark a task as stopped
FALT_TASK	CPFaultTask	QSVCFAULT	32	Fault a task, that is stop it and set a task error code
TASK_HALT	CPHaltTask	QSVCHALT	33	Task halt, with optional error code
-	-	-	34	Not used
-	-	-	35	Not used
SET_PRE	CPSetPreempt	QSVCPREMPT	36	Set a task preemptable flag
CRIT_ENTER	CPCritEnter	QSVCCRITENT	37	Enter a critical code section
CRIT_LEAVE	CPCritLeave	QSVCCRITLEV	38	Leave a critical code section
CHG_PRTY	CPChgPriority	QSVCCHPRTY	39	Change the priority of a task
-	-	-	40	Not used
-	-	-	41	Not used
P_TRACE	CPPTrace	QSVCPTRACE	42	Debugging facilities
-	CPTaskRegs	-	-	Architecture independent routine to initialize the registers of a task.
WRITE_SDA	CPWriteSDA	QSVCWRITENDA	43	Write user extension areas of the SDA
CRT_TASK	CPSysCreateThread	QSVCCRTASK	44	Create a task (Resource Manager only)
CRT_RES_SEM	CPSysCreate-SerSem	QSVCCRRSEM	45	Create a resource semaphore (Resource Manager only)
CRT_SYNC_SEM	CPSysCreate-SyncSem	QSVCCRSSEM	46	Create a synchronization semaphore (Resource Manager only)
CRT_MESGQ	CPSysCreateMsgQ	QSVCCRMSGQ	47	Create a message queue (Resource Manager only)
CRT_UITEM	CPSysCreateUItem	QSVCCRUITEM	48	Create a user SVT item (Resource Manager only)
DEL_ENT	CPSysDeleteEnt	QSVCREMOV	49	Delete an SVT entry from the system
CRT_SLIH	CPSysCreateSLIH	QSVCCRSLIH	50	Set up a second level interrupt handler for hardware interrupts
DEL_SLIH	CPSysDeleteSLIH	QSVCDELSLIH	51	Remove a second level interrupt handler for hardware interrupts
-	-	-	52	Not used
GET_TIME	CPGetTime	QSVCGETTIME	53	Get time, date, and so on.
SET_TIME	CPSetTime	QSVCSETTIME	54	Set current time
ADJUST_TIME	-	QSVCADJTIME	55	Adjust the time of day (forwards or backwards).
SET_DATE	CPSetDate	QSVCSETDATE	56	Set current date
SLEEP	CPSleep	QSVCSLEEP	57	Wait for specified time period
BEEP_IT	CPBeep	QSVCBEEP	58	Sound the beeper (Intel only)
TIMER_SET	CPTimerSet	QSVCSETEVNT	59	Set up a timer event
TIMER_TICK	CPTimerTick	QSVCTIKEVNT	60	Set up a timer event, time is in ticks.
TIMER_CNCL	CPTimerCancel	QSVCCNCLTIM	61	Cancel a timer event
ALLOC_TIMER	CPSysCreateTimer	QSVCALLCTIM	62	Allocate a timer
REL_TIMER	CPSysDeleteTimer	QSVCRELTIM	63	Release a timer
-	-	-	64	Not used
-	-	-	65	Not used
GET_SDA	CPGetCDA	QSVCGETSDA	66	Get address offset of the CDA alias page
SVCTRACE	CPSVCTrace	QSVCTRACE	67	Set or query SVC Handler trace flag

Table A-1 (Page 3 of 3). SVC Handler Function Codes

SVC name	Routine name	Constant name	#	Description
CMOS_READ	CPCMOSRead	QSVCRCMOS	68	Read CMOS locations
READ_UITEM	CPReadUitem	QSVCRUITEM	69	Read contents of a user item SVT entry
WRITE_UITEM	CPWriteUitem	QSVCWUITEM	70	Write contents of a user item SVT entry
RM_FUNC	CPSysResMgrFunc	QSVCRMFUNC	71	Special functions for the Resource Manager
-	-	-	72	Not used
SIG_INTERRUPT	CPSigInt	QSVCSIGINT	73	Signals are or are not to interrupt SVCs
SIG_MASK	CPSigMask	QSVCSIGMASK	74	Set, unset, or query the signal mask
SIG_RETURN	CPSigReturn	QSVCSIGRET	75	Return from a signal handler
SIG_SEND	CPSigSend	QSVCSIGSEND	76	Send a signal
SIG_STACK	CPSigStack	QSVCSIGSTK	77	Set up a signal stack
SIG_VEC	CPSigVec	QSVCSIGVEC	78	Control the action to be taken for a specified signal
-	-	-	79	Not used

SVC Handler Return Codes

This table summarizes the possible return codes generated by the SVC Handler. The values in this table are in hexadecimal. A more complete description of the possible return codes for each SVC is given in the specification of that SVC.

Table A-2 (Page 1 of 3). SVC Handler Return Codes

Return Code	Value	Description
QSVCgood	0	Request completed, no error
QSVCduplnam	80010001	A non-null SVT name is specified which is not unique
QSVCbadSVid	80010002	Specified SVid is invalid or or inaccessible to the caller. Possible reasons include: • It is not of the correct type • The specified SVid cannot be found in the SVT • The specified name is the name of an SVT entry that cannot be accessed by the requestor • If the calling task is a “system” task, the specified SVT entry is protected such that the caller cannot access it
QSVCnfSVT	80010003	No free SVT entries
QSVCnfTCB	80010004	No free TCBs
QSVCnclaim	80010005	Semaphore is not claimed
QSVCbadprty	80010006	Requested task priority is invalid
QSVCbadprmpt	80010007	Requested task pre-emption status is not valid
QSVCbadCMOS	80010008	The CMOS real-time clock is apparently not working
QSVCQempty	80010009	The specified queue is empty
QSVCbadRQE	8001000A	The specified RQE cannot be found on the specified queue
QSVCtimedout	8001000B	The time-out has occurred
QSVCdeadSVid	8001000C	The specified SVT entry is being or has been removed from the system. This can be because either the specific SVT entry or the owning process is being deleted
QSVCbadname	8001000D	The supplied SVT name is not valid, for example, it contains characters that are not valid or is a null name when an SVT entry is specified as having a name
QSVCnobuffer	8001000E	Insufficient free space in the message buffer area

Table A-2 (Page 2 of 3). SVC Handler Return Codes

Return Code	Value	Description
QSVCnowrite	8001000F	The SVT entry exists and is valid, and the caller has “read” access but no “write” access to it
QSVCsmallbuff	80010010	A message has been received, but the caller's buffer was too small, and some of the message is lost
QSVCnoevent	80010011	None of the events specified in a MUX_WAIT SVC has occurred
QSVCsembusy	80010012	Semaphore cannot be claimed since it is already claimed
QSVCtaskstop	80010013	Task is already stopped
QSVCtaskgo	80010014	Task is already running
QSVCnotinit	80010015	Task cannot be started because the registers have not yet been set up
QSVCnotrace	80010016	SVC Handler tracing cannot be turned on since there is no trace buffer
QSVCnf387area	80010017	No free space in the 387 save area
QSVCbadregs	80010018	The register values supplied for a WRITE_REGS SVC are unacceptable in some manner
QSVCbadint	80010019	The specified interrupt level is not valid or is already taken
QSVCincrit	8001001A	The task has been placed in the “pending stopped” state instead of “stopped” because its critical code section count is non-zero
QSVCbadcnt	8001001B	The specified count or length is not valid
QSVCnoroom	8001001C	The buffer supplied in an SVC is not large enough for the data to be returned or no space to allocate SVC Handler data area
QSVCnfPL0	8001001D	No free PL0 stacks
QSVCbadreq	8001001E	SVC option or parameter is not valid
QSVCnfSLIB	8001001F	No free SLIBs
QSVCnoFPU	80010020	There is no floating point processor available
QSVCnotavail	80010021	The requested facility is not available (for example, the system has no real-time clock or no “beeper”)
QSVCbadstack	80010022	Possible reasons include: • The page or pages containing the stack of a PL0 task do not exist • The page or pages containing the stack of a PL0 task are not writeable • The page or pages containing the stack of a PL0 task are not supervisor mode pages. • The page or pages containing the stack do not exist, are not writeable, or are supervisor mode for a user mode task.
QSVCnoFPUregs	80010023	There are no valid FPU registers for this task
QSVCinterrupt	80010024	This SVC was interrupted by a signal and has not been completed
QSVCtracefull	80010025	The trace buffer is full
QSVCtraceoff	80010026	“User trace” not implemented, tracing turned off
QSVCcrit255	80010027	Critical code section count is 255
QSVCnotimers	80010028	No free timer blocks available
QSVCnotdead	80010029	this SVT entry cannot be removed from the system because it is either not marked as removable, or it is not marked as “dying”
QSVCbusySVT	8001002A	This SVT entry cannot be removed from the system because it is still busy in some manner
QSVCbadtimer	8001002B	The specified timer is not active (TIMER_CNCL)
QSVCduplid	8001002C	An SVid is duplicated in the parameter list of a MUX_WAIT SVC
QSVCbadid	8001002D	Possible reasons include: • Specified SLIB ID is not the offset of a SLIB • This SLIH was not set up by the caller • This is not a valid timer block id • This is not the ID of a timer block allocated to the calling task by an ALLOC_TIMER SVC
QSVCSLIH2	8001002E	The specified SVid is already being used for notification of interrupts at this interrupt level (CRT_SLIH SVC)

Table A-2 (Page 3 of 3). SVC Handler Return Codes

Return Code	Value	Description
QSVCdebug2	8001002F	The specified task is already being debugged (P_TRACE function 8).
QSVCnodebug	80010030	The specified task cannot be debugged (P_TRACE function 8), or the specified task is not being debugged by the caller (P_TRACE function 9).
QSVCunpriv	80010031	The caller has insufficient privilege to perform this operation
QSVCnoname	80010032	SVT entry is not marked as "named"
QSVCbadsignal	80010033	Signal number is not valid
QSVCbadaddr	80010034	Specified address is not valid, that is,the location does not exist or is not accessible to the calling task
QSVCnfSCB	80010035	There are no free SCBs
QSVCallspaces	80010036	the supplied SVT name is all spaces (not used on Intel systems)
QSVCpendstop	80010037	The specified task is already in the "pending stopped" state.
QSVCregsnf	80010038	No suitable stack frame, containing saved task registers, can be found (READ_REGS, WRITE_REGS SVCs, POWER/PowerPC only)
QSVCtoomanymsgs	80010039	This task is has sent too many messages that have not yet been received.

SVC Handler Task Fault Codes

This table summarizes the possible task fault error codes generated by the SVC Handler for software detected program faults. These values are in the TCB field *TCBerrcode* when the task stop code (in the TCB field *TCBstop*) is set to *QSVCsofterr*. The values are in hexadecimal. A more complete description of the possible faults for each SVC is given in the specification of that SVC.

Table A-3. SVC Handler Generated Task Fault Codes

Fault Code	Value	Description
QSVCinvalid	80	This might be due to one of the following : • Invalid SVC function code. • The calling task does not have the privilege required to make this SVC. • The calling code is insufficiently privileged. • This SVC is part of a facility that has been omitted from the SVC Handler by a compile time option.
QSVCparlist	81	A parameter list for an SVC, used by the SVC Handler, is not valid in some manner, for example it is outside the segment limits or has insufficient access.
QSVCinvSVid	82	A request has been made to, or a request has been made quoting, an SVid to which the requestor has no access.
QSVCunitint	83	This task has executed an INT instruction that uses an un-initialized interrupt entry.
QSVCno387	84	This task has executed a floating point instruction when there is no floating point processor available.
QSVCmsgcount	85	This task sent too many messages that have not yet been received. **Note:** This fault is not generated in the current versions of CP/Q. Instead, the sending task receives the return code *QSVCtoomanymsgs*.
QSVCnotimpl	FF	An SVC has been made that is currently not implemented.

SVC Handler Task Stopped Codes

This table summarizes the task stopped codes, as held in the TCB field *TCBstop*. These values are returned to a task making a QRY_DISP SVC.

Table A-4. Task Stopped Codes

Code Name	**Value**	**Description**
QSVCnostart	0	The task is in the stopped state as a result of having been created but never started.
QSVCstopped	1	The task is stopped as a result of a STOPTASK SVC from another task.
QSVCsigterm	2	This task has been terminated as the result of a signal being received. The signal number is in the TCB field *TCBerrcode*; these are shown in the signal name table in the section "CP/Q Signal Facilities" in the SVC Handler manual.
QSVCsofterr	3	The task is stopped as a result of an error detected by the SVC Handler. The reason for the error is indicated by the value in the TCB field *TCBerrcode*; these are detailed in the section "SVC Handler Task Fault Codes" on page A-5 and also in Table A-3 on page A-5.
QSVCsoftflt	4	The task is stopped as a result of a fault report from another task. The TCB field *TCBerrcode* holds the error code, which was supplied by the task that issued the FALTTASK SVC.
QSVCtaskhalt	5	The task has halted. The TCB field *TCBerrcode* holds the error or return code which was supplied by the task in the TASKHALT SVC.
QSVCpageerr	0x4*x*	The task is stopped by the SVC Handler as a result of a fault detected by the page fault handler. The fault type is specified by the (hex) digit *x* and is as specified by the page fault handler; the error codes are defined in the Memory Manager manual. The TCB field *TCBerrcode* holds the page fault linear address.
QSVCharderr	0x8*x*	The task is stopped by the SVC Handler as a result of a processor detected fault; the fault type is specified by the (hex) digit *x*, as listed below. The TCB field *TCBerrcode* holds the error code returned by the processor, or 0 for those faults that do not have an error code. The following values are all the faults that are theoretically possible on an Intel processor; in practice, some of them will never occur. **0x80** Divide error **0x81** Single step interrupt, or data trap **0x82** NMI **0x83** Break-point interrupt (INT 3 instruction) **0x84** Overflow interrupt (INTO instruction) **0x85** Array bound error (BOUND instruction) **0x86** Opcode not valid **0x87** Processor extension not available **0x88** Double exception **0x89** Processor extension segment overrun **0x8A** Task state segment not valid **0x8B** Segment not present **0x8C** Stack segment overrun **0x8D** General protection exception **0x8E** Page fault **0x90** Floating point error **0x91** Alignment error **Note:** Normally page faults (error code 0x8E) are reported as a message with stop code *QSVCpageerr* = 0x4*x*.

Messages Generated by the SVC Handler

Messages generated by the SVC handler have the sender SVid field set to 0xFFFFFFFF, and the message type set to the value in the first data DWORD of the message (as described below). They are all “short” messages; the reason for sending the message is specified by a message code in the first double word of data. The contents of the next 2-4 double words varies, depending on the message code in the first data double word, as follows:

QINFnucleus = 2
This is an information message from the SVC Handler, and it can be for one of the following reasons held in the second double word:

QSVCfreeSVT = 20
This is a message to the Resource Manager in response to a DEL_ENT SVC that was rejected because the SVT entry in question was “busy” in some manner. This message signifies that the “busy” has been cleared, and the DEL_ENT should be re-tried (it is not guaranteed that the SVT is no longer busy - it may need to be re-tried several times).

The third data double word contains the SVid of the SVT entry concerned.

QSVCexterminate = 21
This is a message to the Resource Manager when a signal has been issued which results in the target task being terminated. This message is a request for the Resource Manager to terminate the task. Note that when this message is sent, the task referred to is already stopped.

The third data double word contains the SVid of the SVT entry of the task concerned, and the fourth double word holds the signal number.

QSVCweneedRQEs = 22
This is a message to the Resource Manager, sent when the system is running short of RQEs. The Resource Manager should request that more be created.

QSVCinfstop = 32
The SVC Handler generates a message of this type when a task is stopped in response to a STOPTASK SVC if the task was placed in the “pending stopped” state because it had a non-zero critical code section count. The message is sent to the issuer of the STOPTASK SVC.

QINFerror = 3
This is a message to indicate that a task has halted, or has (or has been) faulted. Whenever a task faults or stops for any reason other than a STOPTASK SVC (for example a fault is detected by the processor or by the SVC Handler software), the SVC Handler generates a message to the fault handler of the task.

The second data double word contains the SVid of the task concerned. The third data word contains the type or nature of the task halt or fault, as follows. In general, each of these has an associated return or error code, which is in the fourth data word of the message.

QSVCsigterm = 2

The task has been stopped by the SVC Handler because of a signal has been sent to the task. The signal number, which will be one of the values shown in The signal name table in the section “CP/Q Signal Facilities” in the SVC Handler manual is in bytes 12-15 of the message, and also in The TCB field *TCBerrcode*.

QSVCsofterr = 3

The task has been stopped by the SVC Handler because of an error detected by the SVC Handler software. The error type, which in this case is as defined in the section “SVC Handler Detected Faults” in the SVC Handler manual, is in the fourth data double word of the message, and also in the TCB field *TCBerrcode*.

QSVCsoftflt = 4

The task has been stopped by the SVC Handler as the result of a fault report from another task. The fourth data double word of this message and the TCB field *TCBerrcode*, hold the error code, which was supplied by the task that issued the FALTTASK SVC.

QSVCtaskhalt = 5

The task has halted. The fourth data double word of this message and the TCB field *TCBerrcode*, hold the error code, which was supplied by the task in the TASKHALT SVC.

QSVCpageerr = 0x4*x*

The task has been stopped by the SVC Handler because of a fault detected by the page fault handler; the error codes are defined in the Memory Manager manual. In this case, the return code in the TCB field *TCBerrcode*, holds the page fault linear address.

See the section “Task Fault/Halt Messages” in the SVC Handler manual for details of the meaning of the possible values.

QSVCharderr = 0x8*x*

The task has been stopped by the SVC Handler because of a fault detected by the processor. The fault type (that is, the interrupt number by which the fault was reported) is *n*-0x80, where *n* is the value in these bytes of the message and the TCB field *TCBstop*. The fourth data double word of this message, and the TCB field *TCBerrcode*, hold the error code reported by the processor, if any, or 0 for those faults that have no error code.

See the section “Task Fault/Halt Messages” in the SVC Handler manual for details of the meaning of the possible values.

QINFIHattn = 4

A hardware interrupt has occurred on a hardware level for which the receiver of the message is a Second Level Interrupt Handler (SLIH). The second data double word contains the hardware interrupt level (in the range 0-15 on a PS/2), and the third contains the ID of the SLIB for this SLIH, as returned by the SVC handler in the reply to the CRT_SLIH SVC that set up the SLIH.

QINFtimer = 5

The occurrence of a timer event is being notified by a message. The second double word contains the ID of the timer control block for the event that has occurred, and the third contains the current time in ticks since midnight. The fourth word holds the user identifier specified when the timer was set.

QINFpageflt = 8

This message type is generated only if the Memory Manager page fault handler is not installed, and is to notify a page fault in a task. The second data double word contains the SVid of the task concerned, the third data double word contains 0x8E (i.e. hardware error type 14), the fourth double word contains the error code reported by the processor, and the fifth data word contains the page fault linear address, as reported by the processor in the CR2 register.

Appendix B. Memory Manager return codes

Memory Manager user return codes

If a request to the memory manager is successfully completed, the memory manager returns a return code of QMsuccess(0x00000000). All other results return a return code in the form 0x8002nnnn, where the hi-order 8002 indicates a Memory Manager return code, and the low order 16 bits indicate the cause of the error. If an error is detected, one of the following return codes may be returned:

QMsuccess-(0x0000)	The call was completed successfully.
QMbad_offset-(0x0001)	The offset passed is not that of a valid memory object.
QMbad_type-(0x0002)	Error in type field. The Caller did not have sufficient privilege for request, bits combinations set were not valid, reserved bits set, or required bits not supplied.
QMbad_PID-(0x0003)	Process ID provided is not valid.
QMbad_size-(0x0004)	Size provided was not valid - 0 or greater than 2GB-1, or size not contained in a single memory object.
QMbad_addr-(0x0005)	Address and/or length not on page boundary or real memory range beyond 64MB (CPCreateRange call).
QMbad_location-(0x0006)	Location provided was not valid, not within specified object.
QMbad_sparse-(0x0007)	Object not sparse.
QMbad_range-(0x0008)	Range in use.
QMbad_fixlist-(0x0009)	Caller does not have write access to fixlist.
QMbad_fixhandle-(0x000A)	Fixhandle provided is not a valid fixhandle.
QMno_block-(0x0010)	A block of free linear address space of sufficient size is not available in the class requested.
QMno_access-(0x0011)	User does not have requested access to object.
QMno_page-(0x0012)	No free real storage page frame of the desired type available.
QMnot_owned-(0x0013)	Caller does not own referenced object.
QMno_range-(0x0014)	Attempting to create new range (number of ranges already at maximum).
QMno_PFD-(0x0015)	Attempting to create new range; insufficient PFDs available.
QMreal_alloc-(0x0016)	Storage requested by specific real address already in use.
QMproc_limit-(0x0018)	Some process memory limit would have been exceeded by request. object's fix or swap count at maximum.
QMsys_limit-(0x0019)	Some system limit would have been exceeded by request such as available free linear address space or object's fix or swap count at maximum.
QMobj_fixed-(0x001A)	Object's fix count was not zero on call requiring a zero fix count.
QMobj_accessed-(0x001B)	Object's verify count or memory lock count was not zero on call requiring a zero count.
QMaccess_info-(0x001D)	Requester has access to object, information return.
QMnot_implemented-(0x0020)	Function requested has not been implemented.

QMwrap_count-(0x0021)	Memory lock, fix, share, or verify count at it's maximum value, call cannot be processed.
QMprocess_dead-(0x0022)	A function was requested that creates a new memory object while the process in which the object was to be created is in the final stage of process removal. Such requests cannot be honored.
QMunfixNP-(0x0027)	Attempting to un-fix object, but page found marked not present.
QMfix_sparse-(0x0028)	Attempting to fix area within a sparse object that is not committed.
QMuser_fix_error-(0x0029)	Attempting to un-fix object, but object not fixed.
QMuser_verify_error-(0x002A)	Attempting to free verify but object has no verify count
QMuser_swap_error-(0x002B)	Attempting to release no-swap, but object not marked non-swappable.
QMuser_commit_error-(0x002C)	Attempting to commit page already committed or decommit page not previously committed.
QMrestricted_function-(0x0030)	This function requires IOPL and/or Supervisor privilege, and the caller did not have the requisite privilege.
QMinvalid_function-(0x0031)	The requested function or subfunction does not exist.
QMfix_FNP-(0x0032)	Attempting to fix area which contain a Force Not Present page.
QMalias_sparse-(0x0033)	Attempting to alias an area within a sparse object that is not committed.
QMalias_FNP-(0x0034)	Attempting to alias an area that contains a force not present page.
QMbusy_fork-(0x0035)	The request referred to a process which was in the process of doing a fork, and the request could not be honored at this time. The request can be tried again later.
QMcommit_FNP-(0x0037)	Attempting to commit a page marked Force Not Present.

A more complete description of each error code that a call might return can be found in the detailed description for the call, as well as a more precise definition of the error condition in the context of that particular call.

In addition, internal errors might be detected inside by the Memory manager. These return codes are covered in “Memory Manager internal error codes” on page B-3.

Page Fault Handler Fault Handling

In a system such as CP/Q, certain faults (due to user error) are detected by the page fault handler. In CP/Q, the page fault handler, when detecting such an error signal to the SVC handler by a non-zero return code that a fault occurred. The nature of the fault is in the TCBstop code and the offset of the memory reference causing the error is in the TCBerrcode. The stop codes are as follows::

Page protection fault-(0x40) The task was trying to reference the offset contained in the TCBerrcode and either did not have the necessary privilege or access to the underlying page.

Not allocated fault-(0x41) The task was trying to reference the offset contained in the TCBerrcode but no memory object exists at that offset.

Sparse fault-(0x42) The task was trying to reference the offset contained in the TCBerrcode, which is within a sparse memory object, but, the specific offset refers to a page which had not been previously committed.

Forced Not Present fault-(0x43) The task was trying to reference the offset contained in the TCBerrcode but the underlying page frame had been marked Force Not Present.

No real pages-(0x44) There were no free page frames available to satisfy this page fault. TCBerrcode contains QMno_page.

Internal Error-(0x4F) The page fault handler detected an internal error. TCBerrcode contains an error code. These are described in "Memory Manager internal error codes."

Memory Manager internal error codes

During the processing of a Memory Manager request, internal error conditions may be detected. Depending on the nature of the error, several events could occur. The return code is passed back to the requestor, if it is judged that the fault is likely isolated that particular request. Callers receiving an internal error type of return code should terminate as gracefully and quickly as possible. If the damage is severe, but potentially limited to a single task then that task is terminated with the return code via a CPFaultTask call. Catastrophic errors may necessitate terminating the system. An error code is used to indicate the specific internal failure found. All such error codes have the form 0x8002fnnn, where the 'f' indicates an internal error and the 'nnn' indicates the exact cause of the failure. The possible codes are listed below:

Memory manager internal error codes - internal errors detected in the Virtual Memory Manager.

QMnoPCB-(0xf001)	There was no PCB pointer (in the TCB) for the creation of an object. This is a system error.
QMnoOCB-(0xf002)	There were no free OCBs available to satisfy this request. This is a system error.
QMlock_zero-(0xf004)	Memory manager's internal lock was zero on exiting a call that set the lock. This is a system error.
QMnoPFD-(0xf005)	There were no free PFDs available to satisfy this request. This is a system error.
QMbad_rel_chain-(0xf020)	OCB related chain corrupted.
QMbad_alias_chain-(0xf021)	OCB alias chain corrupted.
QMbadOCBtype-(0xf022)	OCB type bits inconsistent.
QMbadOCBchain-(0xf023)	OCB class chain corrupted.

Memory manager initialization internal error codes - internal errors detected during the initialization of the Memory Manager.

QMinit_small-(0xf040)	IPLed system's real memory smaller than MEMSize at system build time (in the TCB) for the creation of an object. This is probably a user error. System should be rebuilt with smaller MEMSize, or more memory is needed on IPLed machine, or IPL code or machine is defective.
QMinit_high-(0xf041)	Create range call for high memory failed. This is a system or IPL code error.

QMinit_low-(0xf042)	Create range call for low memory failed. This is a system or IPL code error.
QMinit_rom-(0xf043)	Offset range for 128KB ROM unavailable. This is a system or IPL code error.
QMinit_pfdmap-(0xf044)	Could not get pfd mapping space. This is a system or IPL code error.

Page Manager error codes - internal errors detected in the Page Manager.

QMPsparse_present-(0xf101) Found sparse page present on allocation.
QMPalloc_present-(0xf102) Found page present on initial allocation.
QMbadPTErealaddr-(0xf030) Could not find PFD for a PTE real address.
QMbadPFDfixcount-(0xf031) Mismatch between PFD and OCB fix count.
QMbadPTErealaddr-(0xf030) Could not find PFD for a PTE real address (Intel only).
QMbadPFDfixcount-(0xf031) Mismatch between PFD and OCB fix count (Intel only).

Page Fault Handler error codes - internal errors detected in the Page Fault Handler. These errors occur asynchronously, that is, not related directly to a memory manager call, and they cause the current task to be faulted by the SVC Handler stopping the task.

Note: Faulting a single task might not resolve the problem, or the faulted task may not be the direct cause of the problem. This may quickly bring the system down. This is thought preferable to immediately crashing the entire system, as removing one or more tasks might allow recovery. If it is determined that an error is absolutely fatal, then the system is terminated directly.

The TCB code is 0x4F, and the TCBerrcode indicates the particular fault internal error code as follows&colon,

QMnoPCB-(0xf001) There was no PCB that matched the CR3 value (page directory address).
QMbad_rel_chain-(0xf020) related OCB chain is corrupted.
QMbadPTErealaddr-(0xf030) Could not find the PFD for a PTE real address.

Appendix C. Resource Manager return codes

Resource Manager user return codes

If a request to the Resource Manager is successfully completed, the Resource Manager returns a return code of *QRMgood*(0x00000000). All other results return a return code in the form 0x8003nnnn, where the hi-order 8003 indicates a Resource Manager return code, and the low order 16 bits indicate the cause of the error. If an error is detected one of the following return codes might be returned:

QRMgood-(0x0000)	The call was completed successfully
QRMnopcb-(0x0001)	No free Process Control Blocks (PCBs) in the system.
QRMnosvt-(0x0002)	No free Supervisor Table Entries (SVTs) in the system.
QRMnotcb-(0x0003)	No free Task Control Blocks (TCBs) in the system.
QRMnotimer-(0x0004)	No free Timer control blocks in the system.
QRMpcbmax-(0x0007)	Process has maximum PCB allocation.
QRMsvtmax-(0x0008)	Process has maximum SVT allocation.
QRMtcbmax-(0x0009)	Process has maximum TCB allocation.
QRMtimermax-(0x000A)	Process has maximum Timer block allocation.
QRMforkchild-(0x000B)	Created child task returning from fork process call.
QRMbadname-(0x000D)	Specified name is a duplicate or not valid.
QRMbadproc-(0x000E)	Specified process is not valid.
QRMbadsvt-(0x000F)	Specified SVid is not valid or is not the SVid of a task.
QRMinvalid-(0x0010)	Process does not support this request, or process is not a resource provider.
QRMtwice-(0x0011)	Deletion already pending for this process.
QRMprocremv-(0x0012)	Request denied, process removal is in progress for the target process.
QRMnoloader-(0x0013)	The Resource Manager could not resolve the task name "QLOADER" in searching for the CP/Q Loader task.
QRMprocabort-(0x0014)	Process has failed.
QRMbadlength-(0x0015)	Parameter or buffer length given on input to the call is not valid.
QRMbadprocacc-(0x0016)	Accessing process is not valid.
QRMnotfound-(0x0017)	Notification request not found.
QRMprocnoname-(0x0018)	Process is not named.
QRMillegal-(0x0019)	Process inaccessible to requestor.
QRMbadopt-(0x001A)	Incorrect or incompatible options.
QRMinv_func-(0x001B)	Function is not valid.
QRMfailure-(0x001C)	A resource provider returned failure for the process removal. The process is not be removed.
QRMbusy-(0x001E)	Task is faulter or is being debugged.
QRMnonrb-(0x001F)	No free Notification Request Blocks (NRBs) in the system.
QRMnomsg-(0x0021)	Resource Manager internal messaging failure.
QRMglobalnrb-(0x0022)	A global tracking request for the same SVid (to be notified) already exists.

QRMforcedremv-(0x0023)	Function completed successfully, but resource created was forced to be removable.
QRMnofunc-(0x0024)	No kernel extension function codes are currently available in the system.
QRMnotglobal-(0x0025)	The entry offset to the user-defined kernel extension is not located in global memory.
QRMbadtimer-(0x0026)	Specified timer block is not valid.
QRMbadukern-(0x0027)	Specified function code specified when creating a user-defined kernel extension is not valid.
QRMdupfunc-(0x0028)	The function code specified is already in by another task.
QRMnowacc-(0x0029)	The Memory Manager has informed the Resource Manager that write access to the output buffer specified is not allowed.
QRMnosupvstk-(0x002A)	There are no supervisor stacks available in the system.
QRMprclocked-(0x002B)	Request denied, the target process is locked (most likely under test) and cannot be removed at this moment. Try again later.
QRMprocfork-(0x002C)	Request was denied. The target process is currently undergoing a process fork operation. Try again later.
QRMprocexec-(0x002D)	Request denied. The target process is currently undergoing a process exec operation. Try again later.
QRMprocproc-(0x002E)	Request denied. The target process is currently undergoing a process creation operation. Try again later.
QRMproccreat-(0x002F)	Request was denied. The target process is currently being created. Try again later.
QRMpending-(0x0030)	Request accepted; however, the target process could not be removed immediately because of the existence of child processes. Process removal for the target process is pending and will be removed after all child processes are removed.
QRMinvalidfunc	Function code number is not valid.
QRMnotimp-(0x00FF)	Request not implemeted.

A more complete description of each error code that a call may return can be found in the detailed description for the call, as well as a more precise definition of the error condition in the context of that particular call.

In addition, internal errors might be detected inside by the Resource Manager. These return codes are described in “Resource Manager internal error codes” on page C-3.

Resource Manager fault codes

Fault codes are generated by the Resource Manager for software detected program faults. These values are in the TCB field *TCBerrcode* when the task stop code (in the TCB field *TCBstop*) is set to *QSVCsofterr* All such error codes have the form 0x8003008n, where the ‘8’ indicates a task fault and the ‘n’ indicates the exact cause of the failure. The possible codes are listed below:

QRMgenlerr-(0x0080)	Fatal error, no cause.
QRMxnopriv-(0x0081)	The caller attempted to create a process with unlimited resources and did not have “SYSTEM” privilege.

Resource Manager internal error codes

During Resource Manager initialization or during the processing of a Resource Manager request, internal error conditions may be detected. These catastrophic errors can necessitate terminating the system. An error code indicates the specific internal failure found. All such error codes have the form 0x800300Cn, where the ‘C’ indicates an internal error and the ‘n’ indicates the exact cause of the failure. The possible codes are listed below:

Resource Manager initialization internal error codes - internal errors detected during the initialization of the Resource Manager.

QRMiniterr2-(0x00C2)	There were too few Process Control Blocks (PCBs) built into the system.
QRMiniterr3-(0x00C3)	There were too few Supervisor Task Entries (SVTs) built into the system.
QRMiniterr4-(0x00C4)	There were too few Timer blocks built into the system.
QRMiniterr5-(0x00C5)	There were too few Task Control Blocks (TCBs) built into the system.

Resource Manager internal error codes - internal errors detected in the Resource Manager.

QRMiniterr1-(0x00C1)	The SVC Handler returned an error when the Resource Manager attempted to claim or release the SDA semaphore.
QRMiniterr6-(0x00C6)	Unrecognized message from SVC Handler to Resource Manager.
QRMiniterr10-(0x00CA)	The Resource Manager has detected an invalid or corrupt process chain.
QRMiniterr11-(0x00CB)	The Resource Manager could not allocate a preallocated resource.
QRMiniterr12-(0x00CC)	The Resource Manager has detected that the RMDA Kernel Extension Control Block has been corrupted.
QRMiniterr13-(0x00CD)	The Resource Manager received an unexpected return code from the SVC Handler when deleting SVT entries.

Glossary and Abbreviations

A

ASCIIZ. An ASCIIZ string consists of a sequence of ASCII characters, terminated by a NUL character (0x00).

B

BIOS. Basic Input/Output System. This is the device driver and interrupt handler code provided with an IBM compatible PC, in a ROM or EEPROM on the system board and certain device adapter boards. BIOS includes code to test the system on power on, to provide interrupt handlers for standard devices, and to IPL an operating system.

C

Common. Common Memory Objects. This is a “class” of memory object defined by the Memory Manager, which can be shared by different processes in a controlled manner. If a common memory object is visible in two (or more) different processes, it has the same linear address offset in each of the processes.

See also Global and Private.

COW. Copy On Write. This is a mechanism whereby programs running in different address spaces may share a data area, but have private copies of any parts (for example, pages) that are written to. This is achieved by placing read-only page references to the data in each address space; when the program attempts to write to a location, a page fault occurs. The page fault handler recognizes this situation, and creates a private writable copy of just the affected data page, leaving the other pages as read-only access to the shared data.

Safe and reliable operation of copy on write requires the supervisor mode read-only page protection facility that is present in 486 and later processors, but omitted from a 386. For this reason, CP/Q does not provide the COW facility when the system is running on a 386 (if COW mode is requested, it is implemented as copy mode, that is a physical copy is made).

CPL. Current Privilege Level. The Intel *x*86 processors have the concept of privilege levels. There are 4, numbered from 0 (most privileged) to 3 (least privileged). If a program is running at, say, privilege level 2, it is allowed to access items specified as being at privilege level 2 or 3, but not those at privilege level 0 or 1. The privilege level associated with the code currently running is called the CPL, and is defined by the two low order bits of the CS register.

The Intel specification denotes CPL 3 as “user” mode, and CPL 0-2 as “supervisor” mode. CP/Q supports only CPL 3 (for “user” programs) and CPL 0, for the system kernel, device drivers and certain other privileged modules.

creator process. The creator process, defined in the Resource Manager creation parameter lists, is the process to which the newly created object is to be attached. If a new process is being created, it is a child of the creator process.

creator task. The creator task is the caller of the create function.

D

Descriptor. The Intel processors provide memory protection facilities at the memory segment level. Each physical segment has an associated *descriptor*, which defines the segment type, access, base address and size. Other descriptors identify control blocks defined by the Intel *x*86 architecture, which determine the action taken by the processor when interrupts occur (both hardware and as a result of INT instructions), or provide and control access to code modules within a system. Descriptors are held in tables, namely the GDT, the LDT and the IDT. Consult a specification of the Intel processors for further information.

DMA. Direct Memory Access. This is the name given to the access to system memory by an I/O controller, as requested by the controller. This is used by I/O controllers, for example disk controllers, to read and/or write directly from or to system memory without requiring any intervention by the main processor; this drastically reduces the system overhead of I/O, and can produce equally dramatic increases in I/O performance.

DPL. Descriptor Privilege Level. The Intel *x*86 processors have the concept of privilege levels. There are 4, numbered from 0 (most privileged) to 3 (least privileged). If a program is running at, say, current privilege level 2, it is allowed to access items specified as being at privilege level 2 or 3, but not those at privilege level 0 or 1. Associated with any descriptor is a value, called the DPL, which specifies the maximum privilege level for code that may access the item (such as a segment) defined by that descriptor.

DRMM. Direct Real Memory Map. The linear address range 0-*n*K (where *n*K is the size of the physical memory of the machine) is reserved, and is available in all address spaces in supervisor mode as a 1-to-1 mapping of linear address to real memory, that is virtual = real.

The probe for the system debugger (NAP) uses the DRMM (it runs in virtual = real address space).

F

FLIH. First Level Interrupt Handler. A code module that is entered by the processor when a hardware interrupt occurs. In a CP/Q system, this term is used more particularly for the FLIHs within the SVC Handler, that generate interrupt notifications for second level interrupt handlers (SLIHs). There can be only one FLIH for any particular hardware interrupt level.

G

GDT. Global Descriptor Table. A table of descriptors, architected by Intel as part of the specification of the *x*86 processors. There is only one active GDT in a system, which is set up during initial system load, and cannot be easily changed once the system is running. This table is used to define the memory segments or code modules accessible to all tasks in the system. Consult a specification of the Intel processors for further information.

Global. Global Memory Objects. This is a “class” of memory object defined by the Memory Manager. Global memory objects are visible in *all* processes, at the same linear address offset in each process.

See also Common and Private.

I

ICB. Interrupt Control Block. A type of control block used by the CP/Q system kernel, to control the use of hardware interrupt levels. The format is defined in the SVC Handler manual.

IDT. Interrupt Descriptor Table. A table of descriptors, architected by Intel as part of the specification of the *x*86 processors. This table is used to determine the action taken by the processor when hardware interrupts occur, or INT instructions are executed. Consult a specification of the Intel processors for further information.

Interrupt Handler. A term for the code or task that is executed when an external interrupt is recognized, upon processor detected fault, or as a result of the software INT instruction.

The particular interrupt handler for a given interrupt is defined by a descriptor in the IDT.

IOPL. Input/Output Privilege Level. The minimum (that is, numerically largest) processor privilege level at which I/O may be performed. The use of instructions to perform I/O to devices is restricted by the *x*86 processors to code for which CPL ≤ IOPL. This means that many programs running in the system simply are not permitted to perform I/O. The IOPL is held in two bits within the FLAGS register, but only privileged code can successfully alter the IOPL. In CP/Q systems, IOPL is set to 0. Instructions that are considered I/O type instructions include the IN and OUT instructions as well as the ability to change the setting of the Interrupt Enable bit in the flags register.

K

Kernel. The CP/Q Kernel is the SVC Handler, Memory Manager and Resource Manager together with their associated data areas (the System Data Area).

L

LDT. Local Descriptor Table. A table of descriptors, architected by Intel as part of the specification of the *x*86 processor. There may be more than one LDT in a system; each task may have no LDT, its own private LDT, or may share an LDT with a number of tasks. This table is used to define the memory segments or code modules local to those tasks that have this table as their LDT. Items in an LDT are not accessible to tasks having a different LDT, unless special provision is made by to create a segment alias. Consult a specification of the Intel processors for further information.

CP/Q does not use LDTs, nor does it support their use.

N

NAP. The probe program required to enable the system level debugger SLEEP to load and debug a remote CP/Q system.

NDA. Nucleus Data Area. This is the fixed area at the start of the System Data Area that holds various items of information, such as the time and date, pointer to the current task, the system GDT, the system TSS, and so on, and also holds the pointers and chain headers for the other control blocks held in other parts of the System Data Area.

NMI. Non-Maskable Interrupt. Almost all processors provide facilities for the processor to be *interrupted* by external events, such as I/O operations completing. Such external interrupts can be disabled (or *masked*) by software running on the processor; for example it is common practice, while handling one interrupt, to prevent other interrupts from occurring.

The Intel *x*86 processors also have a special interrupt which is completely independent of the above mentioned external interrupt mechanism (it is raised by asserting a signal on a special pin of the processor), and which cannot be disabled within the processor. This is called a Non-Maskable Interrupt. An NMI can occur at any time, including while handling an I/O interrupt with interrupts disabled.

The NMI is used in IBM compatible systems for reporting memory parity failures. It can also be used by a debugger to cause an entry to the debugger whenever an NMI is raised (by use of an external push button switch). On some IBM PCs, NMI *can* be masked, through special I/O logic on the system board that is external to the processor.

O

OCB. Object Control Block. A type of control block used by the Memory Manager to record information about every memory “object” in the system.

P

PCB. Process Control Block. A type of control block used by the CP/Q Resource Manager to hold information relevant to a particular process. The format is defined in the Resource Manager manual.

PFD. Page Frame Descriptor. A type of control block used by the Memory Manager to maintain a chain of every allocatable page of physical memory in the system.

Private. Private Memory Objects. This is a “class” of memory object defined by the Memory Manager. Private memory objects are private to the address space of the process that owns them, that is, such objects are not visible in other processes.

See also Common and Global.

process. In CP/Q, a **process** is something that can acquire system resources, such as memory. Everything in the system (that is, all memory, tasks, SVT entries, and so on) belongs to some process. Each process is associated with an address space; conversely, all tasks within a given process share the same address space. Accounting is in general performed at the process level. Processes are arranged in a tree structured hierarchy, with the special process SYSTEM at the top of the tree. Apart from process SYSTEM, each process in the system also “belongs” to the process immediately higher in the tree (its “parent”).

A process is not a code module that can be run on the processor; the process probably contains one or more *tasks* which can be dispatched. The code of these tasks is also not part of the process; however, the memory occupied by that code is assigned to the process.

R

Resource Provider. A resource provider (RP) is any system extension or sub-system, such as a screen manager or file system, which provides a service (or “resource”) to client processes. Providing a service may involve the acquisition of memory or other system resources either directly by the RP or on behalf of the client (using the notion of effective process). The resource tracking interface to the Resource Manager allows the RP to respond to the removal of clients by recovering resources allocated to such clients or otherwise adjusting its internal state.

RPL. Requested Privilege Level. The Intel *x*86 processors have the concept of privilege levels. There are 4, numbered from 0 (most privileged) to 3 (least privileged). If a program is running at, say, privilege level 2, it is allowed to access items specified as being at privilege level 2 or 3, but not those at privilege level 0 or 1. Access to descriptors is performed by having appropriate selectors in segment registers; the privilege level associated with the attempted access is specified by the two low order bits of the segment registers, and is called the RPL.

RQE. Request Queue Element. A type of control block used by the CP/Q system kernel to form message (or request) queues. The format is defined in the SVC Handler manual.

S

SCB. Signal Control Block. A type of control block used by the SVC Handler to record the requested action for specific signals for a task.

SDA. System Data Area. This an area of memory that holds the control blocks, pointers and control variables used by the system kernel code to control the running of the CP/Q system. The SDA consists of a fixed header or anchor area, called the Nucleus Data Area or NDA, and other areas containing control blocks.

The format of the NDA is predefined, and is known at compile time. The areas containing the control blocks may be of variable size (initially set at system build time, but possibly changed when the system is running), and is at offsets that are set at system build time - that is, they are unknown when the system kernel code is compiled. The NDA contains pointers to the other variable areas, and also such items as the pointers to the start of the free block chains.

SLIB. Second Level Interrupt Block. A type of control block used by the CP/Q system kernel for forming lists of second level interrupt handlers. The format is defined in the SVC Handler manual.

SLIH. Second Level Interrupt Handler. A code module (currently a task in a CP/Q system) that is entered as a result of some action taken by a first level interrupt handler (FLIH). There may be more than one SLIH corresponding to a given hardware interrupt level.

SLEEP/R. System Loader and Error Environment Process/Remote A special program designed and written specifically to build and test CP/Q systems. It uses a probe program, called NAP, to remotely debug a system. This program has its own manual, to which the reader is referred for further information.

SVC. SuperVisor Call. This is the mechanism whereby a task requests services of the "system" or of other elements in the system, or performs services for other elements in the system. In a CP/Q system on an Intel processor, an SVC is made by an INT 0x7A instruction (for a privilege level 3 program), or an indirect call for privilege level 0 programs. In a CP/Q system on a RIOS processor, an SVC is made by an SVC instruction.

The CP/Q code module that is entered when an SVC is made is called the SVC Handler.

SVid. The SVid of a CP/Q entity is the name used to specify this entity to the SVC Handler. It consists of an SVT index (that is, the number of the SVT entry within the SVT) and an incarnation number.

SVT. SuperVisor Table. An array of elements, representing system objects known to the supervisor and containing sundry information concerning object type (task, semaphore, other), access permission, and various pointers to associated control blocks and queues.

T

Task. In CP/Q, a dispatchable program or code module is termed a **task**. The system kernel keeps details of each task in an associated Task Control Block and SVT entry.

The Intel processors have architected into them the notion of a task. The processor provides protection between tasks, and also provides facilities to swap tasks, both voluntarily by jump or call instructions, and involuntarily, on interrupts for example. The CP/Q system uses the protection mechanisms, but for performance reasons does not perform hardware task switches, except in special cases such as system abends, or other traps to SLEEP/R.

TCB. Task Control Block. A type of control block used by the CP/Q system kernel, to hold all the information relevant to a particular task. The format is defined in the SVC Handler manual.

TSS. Task State Segment. A type of processor control segment architected by Intel in the design of the *x*86 processors. In particular, it includes the register save area for a task.

In a CP/Q system, there is one TSS used by all tasks (the task change is performed by software within this TSS), and there are certain other TSS's set up by SLEEP which are used when starting CP/Q and also when the system traps back to the debugger.

X

XPT. Auxiliary Page Tables. Are used by the Memory Manager to manage virtual memory. For each active segment that has any virtual memory allocate, there is a pointer to an XPT Directory block.

Index

Artwork Definitions

id	File	Page	References
TXTQ	CPQSET	i	

LERS Definitions

id	File	Page	References
SAMEPG	LSLAPL1 SCRIPT	i	2-1, 3-1, 4-1, 5-1, 6-1, 7-1, 8-1, 10-1, 11-1
NEWPG	LSLAPL1 SCRIPT	i	2-2, 2-6, 2-7, 3-1, 3-3, 3-5, 4-1, 4-6, 4-11, 4-15, 4-19, 4-24, 5-2, 5-4, 6-2, 6-4, 6-5, 6-8, 7-2, 7-5, 7-6, 7-8, 7-10, 8-2, 8-4, 8-5, 8-7, 8-9, 8-10, 8-20, 8-21, 8-23, 9-1, 9-5, 10-1, 10-4, 10-5, 10-6, 10-7, 10-8, 10-12, 11-1, 11-3, 11-4, 11-5, 11-11

Table Definitions

id	File	Page	References
SUM1	RSLSUMS	A-1	A-1, A-1
SUM2	RSLSUMS	A-1	A-1
RET1	RSLSUMS	A-3	A-3, A-3
RET2	RSLSUMS	A-3	A-3
FLT1	RSLSUMS	A-5	A-5, A-5
FLT2	RSLSUMS	A-5	A-5
STP1	RSLSUMS	A-6	A-6, A-6
STP2	RSLSUMS	A-6	A-6

Figures

id	File	Page	References	
PMFIG	RSLSVC3	4-3	4-1	4-2
RMFIG	RSLSVC3	4-7	4-2	4-7
SMFIG	RSLSVC3	4-12	4-3	4-12
SSFIG	RSLSVC3	4-21	4-4	4-20
SRFIG	RSLSVC3	4-25	4-5	4-25
DRFIG	RSLPTRCE	8-11	8-1	8-11, 8-11
IFPRFIG	RSLPTRCE	8-12	8-2	8-11, 8-13
IREGFIG	RSLPTRCE	8-14	8-3	8-14, 8-14
IQTFIG	RSLPTRCE	8-17	8-4	8-16

CPUIFIG	RSLPTRCE	8-17	8-5 8-18
REGCOD	RSLSVC7	8-24	8-6 8-23
TBLOCK	RSLSVC10	10-2	10-1 10-2
QOUTBUF	NSLRMQRY	22-7	22-1
QRESBUF	NSLRMQRY	22-10	22-2
QPSBUF	NSLRMQRY	22-14	22-3
STKCAL	NSLRMUIN	24-44	24-1 24-43
STKINT	NSLRMUIN	24-45	24-2 24-44

Headings

id	File	Page	References
SVHNOTE	LSLAPL1 SCRIPT	1-1	Chapter 1, Section Notes
DISPRET	RSLSVC1	2-7	CPYield (DISP_RET) - Return to Dispatcher 10-6
RCVMSG	RSLSVC3	4-6	CPRecvMsg (RECV_MESG) - Receive a Message
TIMTIK	RSLSVC10	10-12	CPTimerTick (TIMER_TICK) - Setup a Timer Event 10-8
FUNCSTR	RSLPMM	13-1	Chapter 13, Allocation Functions
FUNCEND	RSLPMM	20-1	Chapter 20, Miscellaneous Functions
CRACC	NSLRMCRE	24-1	Creation access checks
ECPROC	NSLRMCRE	24-8	CPCreateProc
ECTASK	NSLRMCRE	24-27	CPCreateTask
CREUIN	NSLRMCRE	24-43	Implementation Notes 24-42, 24-47, 24-47
FKPROC	NSLRMCRE	24-49	CPFork
TRKREQ	NSLRMTRK	26-6	CPTrkRequest
TRKREQA	NSLRMTRK	26-8	CPTrkRequestAll
SVCFUNC	RSLSUMS	A-1	SVC Functions
SVCFALT	RSLSUMS	A-5	SVC Handler Task Fault Codes A-6
RETSS	RSLMRETC	B-1	Appendix B, Memory Manager return codes
PFER	RSLMRETC	B-2	Page Fault Handler Fault Handling
OOPS	RSLMRETC	B-3	Memory Manager internal error codes B-2, B-3
RMOOPS	NSLRMRET	C-3	Resource Manager internal error codes C-2

Index Entries

id	File	Page	References
SVCC1	RSLSVC1	2-1	(1) System Calls (2) specific functions 2-3, 2-6, 2-7, 3-1, 3-2, 3-3, 3-5, 4-1, 4-2, 4-6, 4-11, 4-15, 4-19, 4-24, 5-1, 5-3, 5-4, 6-1, 6-3, 6-4, 6-5, 6-8, 6-8, 7-1, 7-3, 7-3, 7-3, 7-5, 7-6, 7-8, 7-10, 8-1, 8-3, 8-4, 8-5, 8-7, 8-9, 8-10, 8-20, 8-21, 8-23, 9-2, 9-5, 10-1, 10-2, 10-4, 10-5, 10-6, 10-6, 10-7, 10-8, 10-12, 11-1, 11-2, 11-3, 11-4, 11-5, 11-11
SVCI1	RSLSVC1	2-1	(1) SVC (2) specific functions 2-3, 2-6, 2-7, 3-1, 3-2, 3-3, 3-5, 4-1, 4-2, 4-6, 4-11, 4-15, 4-19, 4-24, 5-1, 5-3, 5-4, 6-1, 6-3, 6-4, 6-5, 6-8, 7-1, 7-3, 7-5, 7-6, 7-8, 7-10, 8-1, 8-3, 8-4, 8-5, 8-7, 8-9, 8-10, 8-20, 8-21, 9-2, 9-5, 10-1, 10-2, 10-4, 10-5, 10-6, 10-6, 10-7, 10-8, 10-12, 11-1, 11-2, 11-3, 11-5, 11-11
ALLOCF	RSLPMM	13-1	(1) Allocation Functions (2) CPAllocBase 13-5, 13-9
MCALLS	RSLPMM	13-1	(1) Memory Manager Calls (2) CPAllocBase 13-5, 13-9, 14-1, 14-3, 15-1, 15-5, 15-8, 16-1, 16-3, 17-1, 17-3, 18-1, 18-3, 19-1, 19-5, 19-9, 19-11, 20-1, 20-3, 20-6, 20-8, 20-9, 20-11, 20-15, 20-18, 20-20, 20-22
DEALOC	RSLPMM	14-1	(1) Deallocation Functions 14-1, 14-3
SHAREF	RSLPMM	15-1	(1) Shared Object Functions 15-1, 15-5, 15-8
CHANGEF	RSLPMM	16-1	(1) Change Object Functions 16-1, 16-3
SPARSEF	RSLPMM	17-1	(1) Sparse Object Functions 17-1, 17-3
VERF	RSLPMM	18-1	(1) Memory Verification Functions 18-1, 18-3
FIXF	RSLPMM	19-1	(1) Memory Fixing Functions 19-1, 19-5, 19-9, 19-11
GETREAL	RSLPMM	19-1	(1) Obtaining Real Addresses 19-1, 19-5, 20-18
MISCF	RSLPMM	20-1	(1) Miscellaneous Functions 20-1, 20-3, 20-6, 20-8, 20-9, 20-11, 20-15, 20-18, 20-20, 20-22

Revisions

id	File	Page	References
A1ED27	NSLREVS	i	27th Edition
A1ED28	NSLREVS	i	28th Edition
S3V23	NSLREVS	i	
MMED13	NSLREVS	i	
RMED13	NSLREVS	i	
FS21	NSLREVS	i	
SMED6	NSLREVS	i	
LDED19	NSLREVS	i	
RMED12	?		

? ? 25-4, 25-4, 25-4, 25-5

Tables

id	File	Page	References
SVHFC	RSLSUMS		
		A-5	A-3
			A-6

Processing Options

Runtime values:

Document fileid	LSLAPL1 SCRIPT
Document type	USERDOC
Document style	IBMXAGD
Profile	EDFPRF40
Service Level	0032
SCRIPT/VS Release	4.0.0
Date	98.06.17
Time	14:33:59
Device	PSA
Number of Passes	3
Index	YES
SYSVAR G	INLINE
SYSVAR X	YES

Formatting values used:

Annotation	NO
Cross reference listing	YES
Cross reference head prefix only	NO
Dialog	LABEL
Duplex	YES
DVCF conditions file	(none)
DVCF value 1	(none)
DVCF value 2	(none)
DVCF value 3	(none)
DVCF value 4	(none)
DVCF value 5	(none)
DVCF value 6	(none)
DVCF value 7	(none)
DVCF value 8	(none)
DVCF value 9	(none)
Explode	NO
Figure list on new page	NO
Figure/table number separation	YES
Folio-by-chapter	YES
Head 0 body text	(none)
Head 1 body text	Chapter
Head 1 appendix text	Appendix
Hyphenation	NO
Justification	NO
Language	ENGL
Keyboard	395
Layout	OFF
Leader dots	YES
Master index	(none)
Partial TOC (maximum level)	4
Partial TOC (new page after)	INLINE
Print example id's	NO
Print cross reference page numbers	YES
Process value	(none)
Punctuation move characters	.,
Read cross-reference file	(none)
Running heading/footing rule	NONE
Show index entries	NO
Table of Contents (maximum level)	4
Table list on new page	YES
Title page (draft) alignment	RIGHT
Write cross-reference file	(none)

Imbed Trace

Page 0	CPQSET
Page 0	NSLREVS
Page 2-1	RSLSVC1
Page 3-1	RSLSVC2
Page 4-1	RSLSVC3
Page 5-1	RSLSVC4
Page 6-1	RSLSVC5
Page 7-1	RSLSVC6
Page 8-1	RSLSVC7
Page 8-9	RSLPTRCE
Page 9-1	RSLSVC8
Page 9-4	RSLSVC9
Page 10-1	RSLSVC10
Page 11-1	RSLSVC11
Page 11-4	RSLSTRCE
Page 12-1	RSLPMM
Page 13-4	RSLMFALT
Page 13-8	RSLMFALT
Page 13-13	RSLMFALT
Page 14-2	RSLMFALT
Page 14-3	RSLMFALT
Page 15-4	RSLMFALT
Page 15-7	RSLMFALT
Page 15-10	RSLMFALT
Page 16-2	RSLMFALT
Page 16-4	RSLMFALT
Page 17-2	RSLMFALT
Page 17-4	RSLMFALT
Page 18-2	RSLMFALT
Page 18-4	RSLMFALT
Page 19-4	RSLMFALT
Page 19-8	RSLMFALT
Page 19-10	RSLMFALT
Page 19-12	RSLMFALT
Page 20-2	RSLMFALT
Page 20-5	RSLMFALT
Page 20-7	RSLMFALT
Page 20-8	RSLMFALT
Page 20-10	RSLMFALT
Page 20-14	RSLMFALT
Page 20-17	RSLMFALT
Page 20-19	RSLMFALT
Page 20-21	RSLMFALT
Page 20-23	RSLMFALT
Page 21-1	NSLRMQRY
Page 22-14	NSLRMSET
Page 23-8	NSLRMCRE
Page 24-43	NSLRMUIN
Page 24-52	NSLRMREM
Page 25-10	NSLRMTRK
Page A-1	RSLSUMS
Page A-9	RSLMRETC
Page B-4	NSLRMRET
Page C-4	RSLGLOSS