intel

EXTENDED iRMX® II.3 OPERATING SYSTEM DOCUMENTATION

VOLUME 3 SYSTEM CALLS

Order Number: 461846-001

Intel Corporation
3065 Bowers Avenue
Santa Clara, California 95051

In locations outside the United States, obtain additional copies of Intel documentation by contacting your local Intel sales office. For your convenience, international sales office addresses are located directly before the reader reply card in the back of the manual.

The information in this document is subject to change without notice.

Intel Corporation makes no warranty of any kind with regard to this material, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. Intel Corporation assumes no responsibility for any errors that may appear in this document. Intel Corporation makes no commitment to update or to keep current the information contained in this document.

Intel Corporation assumes no responsibility for the use of any circuitry other than circuitry embodied in an Intel product. No other circuit patent licenses are implied.

Intel software products are copyrighted by and shall remain the property of Intel Corporation. Use, duplication or disclosure is subject to restrictions stated in Intel's software license, or as defined in ASPR 7 104.9 (a)(9).

No part of this document may be copied or reproduced in any form or by any means without prior written consent of Intel Corporation.

The following are trademarks of Intel Corporation and its affiliates and may be used only to identify Intel products:

Above	iLBX	iPSC	OpenNET
BITBUS	i_m	iRMX	ONCE
COMMputer	iMDDX	iSBC	Plug-A-Bubble
CREDIT	iMMX	iSBX	PROMPT
Data Pipeline	Insite	iSDM	Promware
Genius	int_el	iSSB	QUEST
i	int_elBOS	iSXM	QueX
i	Intelevision	Library Manager	Ripplemode
I^2ICE	$int_eligent$ Identifier	MCS	RMX/80
ICE	$int_eligent$ Programming	Megachassis	RUPI
iCEL	Intellec	MICROMAINFRAME	Seamless
iCS	Intellink	MULTIBUS	SLD
iDBP	iOSP	MULTICHANNEL	UPI
iDIS	iPDS	MULTIMODULE	VLSiCEL
	iPSB		

XENIX, MS-DOS, Multiplan, and Microsoft are trademarks of Microsoft Corporation. UNIX is a trademark of Bell Laboratories. Ethernet is a trademark of Xerox Corporation. Centronics is a trademark of Centronics Data Computer Corporation. Chassis Trak is a trademark of General Devices Company, Inc. VAX and VMS are trademarks of Digital Equipment Corporation. Smartmodem 1200 and Hayes are trademarks of Hayes Microcomputer Products, Inc. IBM is a registered trademark of International Business Machines. Soft-Scope is a registered trademark of Concurrent Sciences.

Intel®

VOLUME PREFACE

MANUALS IN THIS VOLUME

This volume (Volume 3, *Extended iRMX® II System Calls*) contains the following manuals, all of which document the iRMX II system calls. In each manual you will find the system calls listed with their syntax and descriptions. Note that since these are reference manuals, their format differs somewhat from the other iRMX II Operating System manuals.

Extended iRMX® II Nucleus System Calls Reference Manual
Extended iRMX® II Basic I/O System Calls Reference Manual
Extended iRMX® II Extended I/O System Calls Reference Manual
Extended iRMX® II Application Loader System Calls Reference Manual
Extended iRMX® II Human Interface System Calls Reference Manual
Extended iRMX® II UDI System Calls Reference Manual

The *Extended iRMX® II Nucleus System Calls Reference Manual* describes the use of all Nucleus system calls.

The *Extended iRMX® II Basic I/O System Calls Reference Manual* describes the use of all BIOS system calls.

The *Extended iRMX® II Extended I/O System Calls Reference Manual* describes the use of all EIOS system calls.

The *Extended iRMX® II Application Loader System Calls Reference Manual* describes the use of all loader system calls.

The *Extended iRMX® II Human Interface System Calls Reference Manual* describes the use of all Human Interface system calls.

The *Extended iRMX® II UDI System Calls Reference Manual* describes the use of all UDI system calls.

VOLUME CONTENTS

Manuals are listed in the order they appear in the volumes. For a synopsis of each manual, refer to the *Introduction to the Extended iRMX\ II Operating System*.

VOLUME 1: *Extended iRMX® II Introduction, Installation, and Operating Instructions*

Introduction to the Extended iRMX II Operating System
Extended iRMX II Hardware and Software Installation Guide
Operator's Guide to the Extended iRMX II Human Interface
Master Index

VOLUME 2: *Extended iRMX® II Operating System User Guides*

Extended iRMX® II Nucleus User's Guide
Extended iRMX® II Basic I/O System User's Guide
Extended iRMX® II Extended I/O System User's Guide
Extended iRMX® II Human Interface User's Guide
Extended iRMX® II Application Loader User's Guide
Extended iRMX® II Universal Development Interface User's Guide
Device Drivers User's Guide

VOLUME 3: *Extended iRMX® II System Calls*

Extended iRMX® II Nucleus System Calls Reference Manual
Extended iRMX® II Basic I/O System Calls Reference Manual
Extended iRMX® II Extended I/O System Calls Reference Manual
Extended iRMX® II Application Loader System Calls Reference Manual
Extended iRMX® II Human Interface System Calls Reference Manual
Extended iRMX® II UDI System Calls Reference Manual

VOLUME 4: *Extended iRMX® II Operating System Utilities*

Extended iRMX® II Bootstrap Loader Reference Manual
Extended iRMX® II System Debugger Reference Manual
Extended iRMX® II Disk Verification Utility Reference Manual
Extended iRMX® II Programming Techniques Reference Manual
Guide to the Extended iRMX® II Interactive Configuration Utility

VOLUME 5: *Extended iRMX® II Interactive Configuration Utility Reference*

Extended iRMX® II Interactive Configuration Utility Reference Manual

REV.	REVISION HISTORY	DATE
-001	Original Issue.	01/88

intel

EXTENDED iRMX®II NUCLEUS SYSTEM CALLS REFERENCE MANUAL

Intel Corporation
3065 Bowers Avenue
Santa Clara, California 95051

PREFACE

INTRODUCTION

This manual documents the system calls of the Nucleus, the innermost layer of the Extended iRMX® II Operating System. The information provided in this manual is intended as a reference to the system calls and provides detailed descriptions of each call.

READER LEVEL

This manual is intended for programmers who are familiar with the concepts and terminology introduced in the *Extended iRMX II Nucleus User's Guide* and with the PL/M-286 programming language.

MANUAL ORGANIZATION

This manual presents logical groupings of Nucleus System calls. The individual calls within each group are in alphabetical order for easy reference. The following list shows how the system calls are grouped:

- Calls for jobs
- Calls for mailboxes
- Calls for semaphores
- Calls for segments and memory pools
- Calls for descriptors
- Calls for all objects
- Calls for exception handlers
- Calls for exception handlers
- Calls for interrupt handlers, tasks, and levels
- Calls for composite objects
- Calls for extension objects
- Calls for deletion control
- Calls for operating system extensions
- Calls for regions
- Calls for MULTIBUS® II systems

This manual uses the following conventions:

- System call names appear as headings on the outside upper corner of each page. The first appearance of each system call name is printed in ink; subsequent appearances are in black ink.
- Throughout this manual, most system calls are shown using a generic shorthand (such as ACCEPT$CONTROL instead of RQ$ACCEPT$CONTROL). This convention is used to make the names easier to understand. Only the calls that are iRMX II versions of iRMX I system calls are spelled out completely (such as RQE$CREATE$JOB). When you use the system calls in your programs, you must specify the actual PL/M-286 external-procedure names.

  You can also invoke the system calls from assembly language, but you must obey the PL/M-286 calling sequences when doing so. For more information on these calling sequences refer to the *Extended iRMX II Programming Techniques Reference Manual.*

Intel®

CONTENTS

Intel®

EXTENDED iRMX® II NUCLEUS SYSTEM CALLS

INTRODUCTION

This manual presents the iRMX® II Nucleus system calls in functional groups and provides a detailed description of each one.

The calling sequence for each call is the same as for the PL/M-286 interface. The information for each system call is organized in the following order:

- A brief sketch of the effects of the call.
- The PL/M-286 calling sequence for the system call.
- Definitions of the input parameters, if any.
- Definitions of the output parameters, if any.
- A detailed description of the effects of the call.
- An example of how the system call can be used.
- The condition codes that can result from using the call, with a description of the possible causes of each condition.

Throughout this manual, PL/M-286 data types such as BYTE, WORD, POINTER and SELECTOR are used. In addition, the iRMX II data types TOKEN and STRING are used. A TOKEN is a 16-bit value that uniquely identifies an iRMX II object. A STRING is a sequence of consecutive bytes in which the first byte specifies the number of bytes that follow it in the string. When these terms are used as data types, they are always capitalized.

Because TOKEN is not a PL/M-286 data type, you must declare it to be literally a SELECTOR every place you use it. The word "token" in lowercase refers to a value that the iRMX II Operating System returns to a TOKEN (the data type) when it creates the object.

The examples used in this manual assume the reader is familiar with PL/M. In these examples, the appropriate DECLARE and INCLUDE statements are made first. The reader should note the use of an INCLUDE statement that declares all of the system calls included in the iRMX II Operating System.

Following this introduction is a system call dictionary in which the calls are grouped according to type. The dictionary includes short descriptions and page numbers of the complete descriptions that follow.

The CREATE$JOB system call creates a job with a single task. The memory pool assigned with this system call is limited in size to 1M byte.

```
job = RQ$CREATE$JOB (directory$size, param$obj, pool$min, pool$max,
      max$objects, max$tasks, max$priority, except$handler,
      job$flags, task$priority, start$address, data$seg, stack$ptr,
      stack$size, task$flags, except$ptr);
```

Input Parameters

directory$size

A WORD specifying the maximum allowable number of entries a job can have in its object directory. The value zero indicates that no object directory is desired. The maximum value for this parameter is 0FF0H.

param$obj

A TOKEN indicating the presence or absence of a parameter object. See the *Extended iRMX II Nucleus User's Guide* for an explanation of parameter objects.

- If a valid selector, it must contain a token for the new job's parameter object.
- If set to SELECTOR$OF(NIL), it indicates that the new job has no parameter object.

pool$min

A WORD that specifies the minimum allowable size of the new job's pool, in 16-byte paragraphs. The pool$min parameter is also the initial size of the new job's pool. Pool$min should be at least two paragraphs (20H). If the stack$ptr parameter has a base value of SELECTOR$OF(NIL), pool$min should be at least two paragraphs plus the value of stack$size in 16-byte paragraphs.

pool$max

A WORD that indicates the maximum allowable size of the new job's memory in 16-byte paragraphs. If pool$max is smaller than pool$min, an E$PARAM error is returned.

max$objects

A WORD that specifies the maximum number of objects that the created job can own.

- If not 0FFFFH, contains the maximum number of objects, created by tasks in the new job, that can exist at one time.
- If 0FFFFH, indicates that there is no limit to the number of objects that tasks in the new job can create.

max$tasks — A WORD that specifies the maximum number of tasks that can exist simultaneously in the new job.

- If not 0FFFFH, it contains the maximum number of tasks that can exist simultaneously in the new job.
- If 0FFFFH, it indicates that there is no limit to the number of tasks that tasks in the new job can create.
- It cannot be zero. A value of 0H will produce the E$LIMIT exception.

max$priority — A BYTE that sets an upper limit on the priority of the tasks created in the new job.

- If not zero, it contains the maximum allowable priority of tasks in the new job. If max$priority exceeds the maximum priority of the parent job, an E$LIMIT error is returned.
- If zero, it indicates that the new job is to inherit the maximum priority attribute of its parent job.

except$handler — A POINTER to a structure of the following form:

```
STRUCTURE(
    EXCEPTION$HANDLER$PTR      POINTER,
    EXCEPTION$MODE             BYTE);
```

If exception$handler$ptr is not NIL, then it is a POINTER to the first instruction of the new job's own exception handler. If exception$handler$ptr is NIL, the new job's exception handler is the system default exception handler. In both cases, the exception handler for the new task becomes the default exception handler for the job.

The exception$mode indicates when control is to be passed to the exception handler. It is encoded as follows:

Value	When Control Passes To Exception Handler
0	Never
1	On programmer errors only
2	On environmental conditions only
3	On all exceptional conditions

job$flags

A WORD containing information that the Nucleus needs to create and maintain the job. The bits (where bit 15 is the high-order bit) have the following meanings:

Bits	Meaning
15-2	Reserved bits that should be set to zero.
1	If 0, then whenever a task in the new job or any of its descendant jobs makes a Nucleus system call, the Nucleus will check the parameters for validity. If 1, the Nucleus will not check the parameters of Nucleus system calls made by tasks in the new job. However, if any ancestor of the new job has been created with this bit set to 0, there will be parameter checking for the new job.
0	Reserved bit that should be set to zero.

task$priority

A BYTE that controls task priority as follows:

- If not zero, it contains the priority of the new job's initial task. If the task$priority parameter is greater (numerically smaller) than the new job's maximum priority attribute, an E$PARAM error is returned.
- If zero, it indicates that the new job's initial task is to have a priority equal to the new job's maximum priority attribute.

start$address

A POINTER to the first instruction of the new job's initial task (the task created with the job).

data$seg

A TOKEN that specifies which data segment the new job's initial task is to use.

- If a valid selector, it is the base selector of the data segment of the new job's initial task.
- If SELECTOR$OF(NIL), it indicates that the new job's initial task assigns its own data segment. Refer to the *Guide to the Extended iRMX II Interactive Configuration Utility* and the *Extended iRMX II Interactive Configuration Utility* Reference manual for more information about data segment allocation.

stack$ptr

A POINTER that specifies the location of the stack for the new job's initial task.

- If the pointer is valid, it points to the base of the user-provided stack of the new job's initial task.

- If the pointer is set to NIL, it indicates that the Nucleus should allocate a stack for the new job's initial task. The length of the allocated segment is equal to the value of the stack$size parameter.

stack$size — A WORD containing the size, in bytes, of the stack of the new job's initial task. The stack size must be at least 16 bytes and should be at least 300 (decimal) bytes if the new task is going to make Nucleus system calls. Refer to the *Extended iRMX II Programming Techniques* manual for further information on estimating stack sizes.

task$flags — A WORD containing information that the Nucleus needs to create and maintain the job's initial task. The bits (where bit 15 is the high order bit) have the following meanings:

Bits	Meaning
15-1	Reserved bits which should be set to zero.
0	If one, the initial task contains floating-point instructions. These instructions require the Numeric Processor Extension (NPX) component for execution. If zero, the initial task does not contain floating-point instructions.

Output Parameters

job — A TOKEN to which the Operating System will return a token for the new job.

except$ptr — A POINTER to a WORD to which the iRMX II Operating System will return the condition code generated by this system call.

Description

The CREATE$JOB system call creates a job with an initial task and returns a token for the job. The new job's parent is the calling task's job. The new job counts as one against the parent job's object limit. The new task counts as one against the new job's object and task limits. The new job's resources come from the parent job, as described in the *Extended iRMX II Nucleus User's Guide*. In particular, the max$task and max$objects values are deducted from the creating job's maximum task and maximum objects attributes, respectively.

CREATE$JOB

This system call is included for compatibility with iRMX I systems. When you use it, your memory pools are limited to 1M byte in size. To allocate larger memory pools, use the RQE$CREATE$JOB system call.

Example

```
/*****************************************************************
* This example illustrates how the CREATE$JOB system call can be *
*  used.                                                         *
*****************************************************************/

DECLARE TOKEN                   LITERALLY 'SELECTOR';
/* NUCLUS.EXT declares all system calls */
$INCLUDE(/rmx286/inc/NUCLUS.EXT)
INITIALTASK: PROCEDURE EXTERNAL;
END INITIALTASK;

    DECLARE job$token               TOKEN;
    DECLARE directory$size          WORD;
    DECLARE param$obj               TOKEN;
    DECLARE pool$min                WORD;
    DECLARE pool$max                WORD;
    DECLARE max$objects             WORD;
    DECLARE max$tasks               WORD;
    DECLARE max$priority            BYTE;
    DECLARE except$handler          POINTER;
    DECLARE job$flags               WORD;
    DECLARE task$priority           BYTE;
    DECLARE start$address           POINTER;
    DECLARE data$seg                TOKEN;
    DECLARE stack$pointer           POINTER;
    DECLARE stack$size              WORD;
    DECLARE task$flags              WORD;
    DECLARE status                  WORD;

SAMPLEPROCEDURE:
 PROCEDURE;
 directory$size = 10;           /* max 10 entries in object directory */
 param$obj = SELECTOR$OF(NIL); /*  new job has no parameter object   */
 pool$min = 01FFH;          /* min 01FFH, max 0FFFFH 16-byte     */
 pool$max = 0FFFFH;         /*  paragraphs in job pool           */
 max$objects = 0FFFFH;      /*  no limit to number of objects    */
 max$tasks = 10;            /* 10 tasks can exist simultaneously */
 max$priority = 0;          /* inherit max priority of parent    */
 except$handler = NIL;      /* use system default except handler */
 job$flags = 0;             /* parameter validation is on        */
 task$priority = 0;         /* set initial task to max priority  */
    start$address = @INITIALTASK;
                                /* points to first instruction of
                                   initial task                  */
```

```
    data$seg = SELECTOR$OF(NIL);  /* initial task sets up own data
                                     segment                    */
    stack$pointer = NIL;          /* Nucleus allocates stack    */
    stack$size = 512;             /* 512 bytes in stack of initial task */
    task$flags = 0;               /* no floating-point instructions */

    •
    •   Typical PL/M-286 Statements
    •

/*****************************************************************
 *  The calling task creates a job with an initial task labeled  *
 *  INITIALTASK.                                                 *
*****************************************************************/

    job$token = RQ$CREATE$JOB     (directory$size,
                                   param$obj,
                                   pool$min,
                                   pool$max,
                                   max$objects,
                                   max$tasks,
                                   max$priority,
                                   except$handler,
                                   job$flags,
                                   task$priority,
                                   start$address,
                                   data$seg,
                                   stack$pointer,
                                   stack$size,
                                   task$flags,
                                   @status);

•
•   Typical PL/M-286 Statements
•

END SAMPLEPROCEDURE;
```

Condition Codes

E$OK	0000H	No exceptional conditions.
EBADADDR	800FH	At least one of the except$handler, data$seg, or stack$ptr parameters is invalid. Either a selector does not refer to a valid segment, or an offset is outside the segment boundaries.
E$CONTEXT	0005H	The job containing the calling task is in the process of being deleted.

E$EXIST	0006H	The param$obj parameter is not SELECTOR$OF(NIL) and is not a token for an existing object.
E$LIMIT	0004H	At least one of the following is true: • max$objects is larger than the unused portion of the object allotment in the calling task's job. • max$tasks is larger than the unused portion of the task allotment in the calling task's job. • max$priority is greater (numerically smaller) than the maximum allowable task priority in the calling task's job. • directory$size is larger than 0FF0H. • The initial task would exceed the object limit in the new job. That is, the max$objects parameter is set to zero. • The initial task would exceed the task limit in the new job. The max$tasks parameter is set to zero.
E$MEM	0002H	At least one of the following is true: • The memory available to the new job is not sufficient to create a job descriptor (an internal data structure) and the object directory. • The memory available to the new job is not sufficient to satisfy the pool$min parameter. • The memory available to the new job is not sufficient to create the task as specified.
E$PARAM	8004H	At least one of the following is true: • pool$min is less than 16 + (number of paragraphs needed for the initial task and a system-allocated stack) + 5 (if the task uses the NPX component). • pool$min is greater than pool$max. • task$priority is unequal to zero and greater (numerically smaller) than max$priority. • stack$size is less than 16. • the exception handler mode is not valid.

E$SLOT	000CH	There isn't enough room in the GDT for the new job and task descriptors.

The RQE$CREATE$JOB system call creates a job with a single task. It provides the same services and has the same syntax as the CREATE$JOB system call, except that it can allocate memory pools of up to 16M bytes in size.

```
job = RQE$CREATE$JOB (directory$size, param$obj, pool$min,
      pool$max, max$objects, max$tasks, max$priority,
      except$handler, job$flags, task$priority, start$address,
      data$seg, stack$ptr, stack$size, task$flags, except$ptr);
```

Input Parameters

directory$size

A WORD specifying the maximum allowable number of entries a job can have in its object directory. The value zero is permitted, for the case where no object directory is desired. The maximum value for this parameter is 0FF0H.

param$obj

A TOKEN indicating the presence or absence of a parameter object. See the *Extended iRMX II Nucleus User's Guide* for an explanation of parameter objects.

- If a valid selector, it must contain a token for the new job's parameter object.
- If set to SELECTOR$OF(NIL), it indicates that the new job has no parameter object.

pool$min

A DWORD that specifies the minimum allowable size of the new job's pool, in 16-byte paragraphs. The pool$min parameter is also the initial size of the new job's pool. Pool$min should be at least two paragraphs (20H bytes) and no more than 0FFFFFH. If the stack$ptr parameter has a base value of SELECTOR$OF(NIL), pool$min should be at least two paragraphs plus the value of stack$size in 16 byte paragraphs.

pool$max

A DWORD that indicates the maximum allowable size of the new job's memory in 16-byte paragraphs. If pool$max is smaller than pool$min, an E$PARAM error is returned.

max$objects

A WORD that specifies the maximum number of objects that the created job can own.

- If not 0FFFFH, contains the maximum number of objects, created by tasks in the new job, that can exist at one time.

- If 0FFFFH, indicates that there is no limit to the number of objects that tasks in the new job can create.

max$tasks — A WORD that specifies the maximum number of tasks that can exist simultaneously in the new job.

- If not 0FFFFH, it contains the maximum number of tasks that can exist simultaneously in the new job.
- If 0FFFFH, it indicates that there is no limit to the number of tasks that tasks in the new job can create.
- It cannot be zero. A value of 0H will produce the E$LIMIT exception.

max$priority — A BYTE that sets an upper limit on the priority of the tasks created in the new job.

- If not zero, it contains the maximum allowable priority of tasks in the new job. If max$priority exceeds the maximum priority of the parent job, an E$LIMIT error is returned.
- If zero, it indicates that the new job is to inherit the maximum priority attribute of its parent job.

except$handler — A POINTER to a structure of the following form:

```
STRUCTURE(
EXCEPTION$HANDLER$PTR      POINTER,
EXCEPTION$MODE             BYTE);
```

If exception$handler$ptr is not NIL, then it is a POINTER to the first instruction of the new job's own exception handler. If exception$handler$ptr is NIL, the new job's exception handler is the system default exception handler. In both cases, the exception handler for the new task becomes the default exception handler for the job.

The exception$mode indicates when control is to be passed to the exception handler. It is encoded as follows:

Value	When Control Passes To Exception Handler
0	Never
1	On programmer errors only
2	On environmental conditions only
3	On all exceptional conditions

job$flags — A WORD containing information that the Nucleus needs to create and maintain the job. The bits (where bit 15 is the high-order bit) have the following meanings:

Bits	Meaning
15-2	Reserved bits that should be set to zero.
1	If 0, then whenever a task in the new job or any of its descendant jobs makes a Nucleus system call, the Nucleus will check the parameters for validity. If 1, the Nucleus will not check the parameters of Nucleus system calls made by tasks in the new job. However, if any ancestor of the new job has been created with this bit set to 0, there will be parameter checking for the new job.
0	Reserved bit that should be set to zero.

task$priority — A BYTE that controls task priority as follows:

- If not zero, it contains the priority of the new job's initial task. If the task$priority parameter is greater (numerically smaller) than the new job's maximum priority attribute, an E$PARAM error is returned.
- If zero, it indicates that the new job's initial task is to have a priority equal to the new job's maximum priority attribute.

start$address — A POINTER to the first instruction of the new job's initial task (the task created with the job).

data$seg — A TOKEN that specifies which data segment the new job's initial task is to use.

- If a valid selector, it is the base selector of the data segment of the new job's initial task.
- If SELECTOR$OF(NIL), it indicates that the new job's initial task assigns its own data segment. Refer to the *Guide to the Extended iRMX II Interactive Configuration Utility* and the *Extended iRMX II Interactive Configuration Utility Reference* manual for more information about data segment allocation.

stack$ptr — A POINTER that specifies the location of the stack for the new job's initial task.

- If the pointer is valid, it points to the base of the user-provided stack of the new job's initial task.

- If the pointer is set to NIL, it indicates that the Nucleus should allocate a stack for the new job's initial task. The length of the allocated segment is equal to the value of the stack$size parameter.

stack$size — A WORD containing the size, in bytes, of the stack of the new job's initial task. This size must be at least 16 (decimal) bytes. The Nucleus increases specified values that are not multiples of 16 up to the next higher multiple of 16.

The stack size should be at least 300 (decimal) bytes if the new task is going to make Nucleus system calls. Refer to the *Extended iRMX II Programming Techniques* manual for further information on estimating stack sizes.

task$flags — A WORD containing information that the Nucleus needs to create and maintain the job's initial task. The bits (where bit 15 is the high order bit) have the following meanings:

Bits	Meaning
15-1	Reserved bits which should be set to zero.
0	If one, the initial task contains floating-point instructions. These instructions require the Numeric Processor Extension (NPX) component for execution. If zero, the initial task does not contain floating-point instructions.

Output Parameters

job — A TOKEN to which the Operating System will return a token for the new job.

except$ptr — A POINTER to a WORD to which the iRMX II Operating System will return the condition code generated by this system call.

Description

The RQE$CREATE$JOB system call creates a job with an initial task and returns a token for the job. The new job's parent is the calling task's job. The new job counts as one against the parent job's object limit. The new task counts as one against the new job's object and task limits. The new job's resources come from the parent job, as described in the *Extended iRMX II Nucleus User's Guide*. In particular, the max$task and max$objects values are deducted from the creating job's maximum task and maximum objects attributes, respectively.

RQE$CREATE$JOB

This system call is an extension of the CREATE$JOB system that supports the full memory-addressing capabilities of the iRMX II Operating System. When you use it, you can assign memory pools of up to 16M bytes in size.

Example

```
/*****************************************************************
 *  This example illustrates how the RQE$CREATE$JOB system call   *
 *  can be used.                                                  *
*****************************************************************/

DECLARE TOKEN                    LITERALLY 'SELECTOR';

/* NUCLUS.EXT declares all system calls */

$INCLUDE(/rmx286/inc/NUCLUS.EXT)
INITIALTASK: PROCEDURE EXTERNAL;
END INITIALTASK;

    DECLARE job$token              TOKEN;
    DECLARE directory$size         WORD;
    DECLARE param$obj              TOKEN;
    DECLARE pool$min               DWORD;
    DECLARE pool$max               DWORD;
    DECLARE max$objects            WORD;
    DECLARE max$tasks              WORD;
    DECLARE max$priority           BYTE;
    DECLARE except$handler         POINTER;
    DECLARE job$flags              WORD;
    DECLARE task$priority          BYTE;
    DECLARE start$address          POINTER;
    DECLARE data$seg               TOKEN;
    DECLARE stack$pointer          POINTER;
    DECLARE stack$size             WORD;
    DECLARE task$flags             WORD;
    DECLARE status                 WORD;

SAMPLEPROCEDURE:
 PROCEDURE;
  directory$size = 10;           /* max 10 entries in object directory */
  param$obj = SELECTOR$OF(NIL); /*  new job has no parameter object */
  pool$min = 01FFH;              /* min 01FFH, max 0FFFFFH 16-byte */
  pool$max = 0FFFFFH;             /*  paragraphs in job pool      */
  max$objects = 0FFFFH;          /*  no limit to number of objects */
  max$tasks = 10;                /* 10 tasks can exist simultaneously  */
  max$priority = 0;              /* inherit max priority of parent     */
  except$handler = NIL;          /* use system default except handler  */
  job$flags = 0;                 /* parameter validation is on         */
  task$priority = 0;             /* set initial task to max priority   */
```

```
   start$address = @INITIALTASK;
                                /* points to first instruction of
                                   initial task                  */
   data$seg = SELECTOR$OF(NIL); /* initial task sets up own data
                                    segment                      */
   stack$pointer = NIL;         /* Nucleus allocates stack      */
   stack$size = 512;            /* 512 bytes in stack of initial task */
   task$flags = 0;              /* no floating-point instructions */

•
•   Typical PL/M-286 Statements
•

/*****************************************************************
 *  The calling task creates a job with an initial task labeled  *
 *  INITIALTASK.                                                 *
*****************************************************************/

    job$token = RQE$CREATE$JOB     (directory$size,
                                   param$obj,
                                   pool$min,
                                   pool$max,
                                   max$objects,
                                   max$tasks,
                                   max$priority,
                                   except$handler,
                                   job$flags,
                                   task$priority,
                                   start$address,
                                   data$seg,
                                   stack$pointer,
                                   stack$size,
                                   task$flags,
                                   @status);

•
•   Typical PL/M-286 Statements
•

END SAMPLEPROCEDURE;
```

Condition Codes

E$OK	0000H	No exceptional conditions.
EBADADDR	800FH	At least one of the following parameters is invalid: except$handler, data$seg, or stack$ptr. Either a selector does not refer to a valid segment, or an offset is outside the segment boundaries.
E$CONTEXT	0005H	The job containing the calling task is in the process of being deleted.
E$EXIST	0006H	The param$obj parameter is not SELECTOR$OF(NIL) and is not a token for an existing object.
E$LIMIT	0004H	At least one of the following is true: • max$objects is larger than the unused portion of the object allotment in the calling task's job. • max$tasks is larger than the unused portion of the task allotment in the calling task's job. • max$priority is greater (numerically smaller) than the maximum allowable task priority in the calling task's job. • directory$size is larger than 0FF0H. • The initial task would exceed the object limit in the new job. That is, the max$objects parameter is set to zero. • The initial task would exceed the task limit in the new job. The max$tasks parameter is set to zero.
E$MEM	0002H	At least one of the following is true: • The memory available to the new job is not sufficient to create a job descriptor (an internal data structure) and the object directory. • The memory available to the new job is not sufficient to satisfy the pool$min parameter. • The memory available to the new job is not sufficient to create the task as specified.

E$PARAM	8004H	At least one of the following is true: • pool$min is less than 16 + (number of paragraphs needed for the initial task and a system-allocated stack) + 5 (if the task uses the NPX component). • pool$min is greater than pool$max. • task$priority is unequal to zero and greater (numerically smaller) than max$priority. • stack$size is less than 16. • the exception handler mode is not valid.
E$SLOT	000CH	There isn't enough room in the GDT for the new job and task descriptors.

The DELETE$JOB system call deletes a job.

```
CALL RQ$DELETE$JOB (job, except$ptr);
```

Input Parameter

job	A TOKEN for the job to be deleted. A value of SELECTOR$OF(NIL) specifies the calling task's job.

Output Parameter

except$ptr	A POINTER to a WORD to which the iRMX II Operating System will return the condition code generated by this system call.

Description

The DELETE$JOB system call deletes the specified job, all of the job's tasks, and all objects created by the tasks. Exceptions are that jobs and extension objects (see the *Extended iRMX II Nucleus User's Guide*) created by tasks in the target job must be deleted prior to the call to DELETE$JOB. Information concerning the descendants of a job can be obtained by invoking the OFFSPRING system call.

During the deletion of any interrupt tasks owned by the job, the interrupt levels associated with those tasks are reset. The levels that do not have interrupt tasks associated with them will not be reset during an RQ$DELETE$JOB call.

During deletion, all resources that the target job had borrowed from its parent are returned.

Deleting a job causes a credit of one toward the object total of the parent job. Also, the maximum tasks and maximum objects attributes of the deleted job are credited to the current tasks and current objects attributes, respectively, of the parent job.

Example

```
/*****************************************************************
 * This example illustrates how the DELETE$JOB system call can be   *
 *  used to delete the calling task's job.                          *
*****************************************************************/

DECLARE TOKEN                   LITERALLY 'SELECTOR';
```

```
/* NUCLUS.EXT declares all system calls */
$INCLUDE(/rmx286/inc/NUCLUS.EXT)

    DECLARE calling$tasks$job     TOKEN;
    DECLARE status                WORD;

SAMPLEPROCEDURE:
    PROCEDURE;

  calling$task$job = SELECTOR$OF(NIL);  /* Set job to task's job. */

        •
        •     Typical PL/M-286 Statements
        •

/*****************************************************************
 * If you set the job parameter to SELECTOR$OF(NIL), the DELETE$JOB *
 *  system call will delete the calling task's job.               *
*****************************************************************/

    CALL RQ$DELETE$JOB (calling$tasks$job, @status);

END SAMPLEPROCEDURE;
```

Condition Codes

E$OK	0000H	No exceptional conditions.
E$CONTEXT	0005H	At least one of the following is true: • There are undeleted jobs or extension objects (see the *Extended iRMX II Nucleus User's Guide*) which have been created by tasks in the target job. • The deleting task has access to data guarded by a region contained in the job to be deleted. (Refer to the *Extended iRMX II Nucleus User's Guide* for information concerning regions.)
E$EXIST	0006H	The job parameter is not a token for an existing object.
ENOTCONFIGURED	0008H	This system call is not part of the present configuration.
E$TYPE	8002H	The job parameter is a token for an object that is not a job.

The OFFSPRING system call returns a token for each child (job) of a job.

```
token$list = RQ$OFFSPRING (job, except$ptr);
```

Input Parameter

job — A TOKEN for the job whose offspring are desired. A value of SELECTOR$OF(NIL) specifies the calling task's job.

Output Parameter

token$list — A TOKEN that indicates the children of the specified job.

- If a valid selector, the TOKEN contains a token for a segment. The first word in the segment contains the number of words in the remainder of the segment. Subsequent words contain the tokens for jobs that are the immediate children of the specified job.
- If SELECTOR$OF(NIL), the specified job has no children.

except$ptr — A POINTER to a WORD to which the iRMX II Operating System will return the condition code generated by this system call.

Description

The OFFSPRING system call returns the token for a segment. The segment contains a token for each child of the specified job. By repeated use of this call, tokens can be obtained for all descendants of a job; this information is needed by a task which is attempting to delete a job that has child jobs.

Example

```
/*************************************************************************
 *  This example illustrates how the OFFSPRING system call can be used  *
 *  to return a token for each child of a job.                           *
 *************************************************************************/

DECLARE TOKEN      LITERALLY 'SELECTOR';

/*  NUCLUS.EXT declares all system calls */
$INCLUDE(/rmx286/inc/NUCLUS.EXT)
```

```
    DECLARE token$list              TOKEN;
    DECLARE calling$tasks$job       TOKEN;
    DECLARE status                  WORD;

SAMPLEPROCEDURE:
    PROCEDURE;

        •
        •     Typical PL/M-286 Statements
        •

/*************************************************************************
 *  In this example, the calling task invokes the system call OFFSPRING *
 *  to obtain a token for a segment.  This segment contains the tokens  *
 *  for jobs that are immediate children of the calling task's job.     *
 *************************************************************************/

    calling$tasks$job = SELECTOR$OF(NIL);

    token$list = RQ$OFFSPRING       (calling$tasks$job,
                                     @status);
        •
        •     Typical PL/M-286 Statements
        •

END SAMPLEPROCEDURE;
```

Condition Codes

E$OK	0000H	No exceptional conditions.
E$EXIST	0006H	The job parameter is not a token for an existing object.
E$LIMIT	0004H	The calling task's job has already reached its object limit.
E$MEM	0002H	The memory available to the specified job is not sufficient to complete this call.
ENOTCONFIGURED	0008H	This system call is not part of the present configuration.

OFFSPRING

E$SLOT	000CH	There isn't enough room in the GDT for another descriptor.
E$TYPE	8002H	The job parameter contains a token for an object that is not a job.

The RQE$OFFSPRING system call performs the same function as the RQ$OFFSPRING system call. However, RQE$OFFSPRING returns the list of child job tokens in a structure that you supply, rather than in a segment.

```
CALL RQE$OFFSPRING (job, list$ptr, except$ptr);
```

Input Parameter

job — A TOKEN for the job whose offspring are desired. A value of SELECTOR$OF(NIL) specifies the calling task's job.

Output Parameters

list$ptr — A POINTER to a STRUCTURE in which the system call returns tokens for the children of the specified job. The format of this data structure is as follows:

```
DECLARE offspring  STRUCTURE (
                              max$num WORD,
                              actual  WORD,
                              children(*) TOKEN);
```

The fields of this structure are as follows:

max$num — This is actually an input field. Before invoking the system call, you must set this field to indicate the maximum number of child job tokens the system call can return in this structure. That is, this field must specify the number of slots for children tokens in this structure. The value in this field must be greater than zero.

actual — The system call fills in this field to indicate the number of tokens it returned in this structure. This number will never be larger than the max$num value.

children(*) — The system call fills in these fields with the tokens for the immediate children of the specified job. The number of tokens in this list is indicated in the actual field.

except$ptr — A POINTER to a WORD to which the iRMX II Operating System will return the condition code generated by this system call.

Description

The RQE$OFFSPRING system call returns a structure that contains a token for each child of the specified job. By repeated use of this call, tokens can be obtained for all descendants of a job. This information is needed by a task that is attempting to delete a job that has child jobs.

This system call returns exactly the same information as RQ$OFFSPRING. The only difference between the two system calls is that RQ$OFFSPRING creates an iRMX II segment to contain the information about the offspring tokens; RQE$OFFSPRING returns the token information in a structure that you supply. Using structures instead of iRMX II segments minimizes the number of iRMX II objects (and thus the number of GDT entries). It also means that the memory for the list is allocated when the task starts running, not dynamically when needed. This minimizes the chance of the system call failing because of a lack of memory.

The offspring structure that you supply has two fields in addition to the slots in which the system call returns the tokens. The first, max$num, is an input parameter that you fill in to indicate the amount of room in the structure for offspring tokens. The second field, actual, is filled in by the system call when it returns the tokens. The actual field is set to indicate the number of tokens actually returned. If there are more tokens to be returned than slots in the structure, the system call returns only enough to fill up the structure (that is, max$num).

Example

```
/*************************************************************************
 *  This example illustrates how the RQE$OFFSPRING system call can be   *
 *  used to return a token for each child of a job.                      *
 *************************************************************************/

DECLARE TOKEN      LITERALLY 'SELECTOR';

/*  NUCLUS.EXT declares all system calls */
$INCLUDE(/rmx286/inc/NUCLUS.EXT)

    DECLARE token$list            STRUCTURE (
            max$num               WORD,
            actual                WORD,
            children(20)          TOKEN);
    DECLARE calling$tasks$job     TOKEN;
    DECLARE status                WORD;
```

```
SAMPLEPROCEDURE:
    PROCEDURE;

        •
        •     Typical PL/M-286 Statements
        •

/*************************************************************************
 *  In this example, the calling task invokes the system call           *
 *  RQE$OFFSPRING to obtain a list of up to 20 tokens for the jobs that *
 *  are the immediate children of the calling task's job.               *
 *************************************************************************/

    calling$tasks$job = SELECTOR$OF(NIL);
    token$list.max$num = 20;

    CALL RQE$OFFSPRING     (calling$tasks$job, @token$list
                                @status);
        •
        •     Typical PL/M-286 Statements
        •

END SAMPLEPROCEDURE;
```

Condition Codes

E$OK	0000H	No exceptional conditions.
E$EXIST	0006H	The job parameter is not a token for an existing object.
ENOTCONFIGURED	0008H	This system call is not part of the present configuration.
E$TYPE	8002H	The job parameter contains a token for an object that is not a job.

The CREATE$TASK system call creates a task.

```
task = RQ$CREATE$TASK (priority, start$address, data$seg, stack$ptr,
       stack$size, task$flags, except$ptr);
```

Input Parameters

priority — A BYTE that specifies the priority of the new task.

- If not zero, it contains the priority of the new task. The priority parameter must not exceed the maximum allowable priority of the calling task's job. If it does, an E$PARAM error is returned.
- If zero, it indicates that the new task's priority is to equal the maximum allowable priority of the calling task's job.

start$address — A POINTER to the first instruction of the new task.

data$seg — A TOKEN that specifies the new task's data segment.

- If a valid selector, the TOKEN contains the base address of the new task's data segment.
- If set to SELECTOR$OF(NIL), the TOKEN indicates that the new task assigns its own data segment. Refer to *Guide To The Extended iRMX II Interactive Configuration Utility* and *Extended iRMX II Interactive Configuration Utility Reference Manual* for further information on data segment allocation.

stack$ptr — A POINTER that specifies the location of the stack for the new task.

- If this is a valid pointer, the Nucleus uses the sum of the offset portion and the stack$size parameter (declared during the call to CREATE$TASK) as the value of the SP register (the stack pointer).
- If the pointer is set to NIL, the Nucleus allocates a stack to the new task. The length of the stack is equal to the value of the stack$size parameter.

stack$size — A WORD containing the size, in bytes, of the new task's stack segment.

The stack size must be at least 16 bytes and should be at least 300 bytes if the new task is going to make Nucleus system calls. Refer to the *Extended iRMX II Programming Techniques* manual for further information on assigning stack sizes.

task$flags — A WORD containing information that the Nucleus needs to create and maintain the task. The bits (where bit 15 is the high-order bit) have the following meanings:

Bits	Meaning
15-1	Reserved bits which should be set to zero
0	If one, the task contains floating-point instructions. These instructions require the NPX component for execution If zero, the task does not contain floating-point instructions

Output Parameters

task — A TOKEN to which the Operating System will return a token for the new task.

except$ptr — A POINTER to a WORD to which the iRMX II Operating System will return the condition code generated by this system call.

Description

The CREATE$TASK system call creates a task and returns a token for it. The new task counts as one against the object and task limits of the calling task's job. Attributes of the new task are initialized upon creation as follows:

- priority: as specified in the call.
- execution state: ready.
- suspension depth: 0.
- containing job: the job that contains the calling task.
- exception handler: the exception handler of the containing job.
- exception mode: the exception mode of the containing job.

Example

```
/*************************************************************************
 *  This example illustrates how the CREATE$TASK system call can be      *
 *  used.                                                                *
 *************************************************************************/

DECLARE TOKEN       LITERALLY 'SELECTOR';

/* NUCLUS.EXT declares all system calls */
$INCLUDE(/rmx286/inc/NUCLUS.EXT)

TASKCODE: PROCEDURE EXTERNAL;
END TASKCODE;

    DECLARE task$token          TOKEN;
    DECLARE priority$level$210 LITERALLY '210';
    DECLARE start$address       POINTER;
    DECLARE data$seg            TOKEN;
    DECLARE stack$pointer       POINTER;
    DECLARE stack$size$512      LITERALLY '512';
                                /* new task's stack size is 512 bytes */
    DECLARE task$flags          WORD;
    DECLARE status              WORD;

SAMPLEPROCEDURE:
    PROCEDURE;

    start$address = @TASKCODE;   /* first instruction of the new task */
    data$seg = SELECTOR$OF(NIL); /* task sets up own data segment */
    stack$pointer = NIL;         /* automatic stack allocation  */
    task$flags = 0;       /* designates no floating-point instructions */

        •
        •     Typical PL/M-286 Statements
        •

/*************************************************************************
 *  The task (whose code is labeled TASKCODE) is created when the        *
 *  calling task invokes the CREATE$TASK system call.                    *

*************************************************************************/

    task$token = RQ$CREATE$TASK   (priority$level$66,
                                   start$address,
                                   data$seg,
                                   stack$pointer,
                                   stack$size$512,
                                   task$flags,
                                   @status);
```

```
•
•     Typical PL/M-286 Statements
•

END SAMPLEPROCEDURE;
```

Condition Codes

E$OK	0000H	No exceptional conditions.
EBADADDR	800FH	The data$seg, or stack$ptr is invalid. Either a selector does not refer to a valid segment, or an offset is outside the segment boundaries.
E$LIMIT	0004H	The calling task's job has already reached its object limit or task limit.
E$MEM	0002H	The memory available to the calling task's job is not sufficient to create a task as specified (task descriptor, stack, and possibly NPX area).
ENOTCONFIGURED	0008H	This system call is not part of the present configuration.
E$PARAM	8004H	At least one of the following is true: • The stack$size parameter is less than 16. • The priority parameter is nonzero and greater (numerically smaller) than the maximum allowable priority for tasks in the calling task's job.
E$SLOT	000CH	There isn't enough room in the GDT for another descriptor.

The DELETE$TASK system call deletes a task.

```
CALL RQ$DELETE$TASK (task, except$ptr);
```

Input Parameter

task — A TOKEN that identifies the task to be deleted.

- If a valid selector, the TOKEN must contain a token for the task to be deleted.
- If SELECTOR$OF(NIL), this parameter indicates that the calling task should be deleted.

Output Parameter

except$ptr — A POINTER to a WORD to which the iRMX II Operating System will return the condition code generated by this system call.

Description

The DELETE$TASK system call deletes the specified task from the system and from any queues in which the task was waiting. DELETE$TASK allows any task currently within a region to exit the region before being deleted. Deleting the task counts as a credit of one toward the object total of the containing job. It also counts as a credit of one toward the containing job's task total.

You cannot successfully delete an interrupt task by invoking this system call. Any attempt to do so results in an E$CONTEXT exceptional condition. To delete an interrupt task, invoke the RESET$INTERRUPT system call.

Example

```
/*****************************************************************
 *  This example illustrates how the DELETE$TASK system call can be *
 *  used.                                                          *
 *                                                                 *
*****************************************************************/

DECLARE TOKEN       LITERALLY 'SELECTOR';
```

```
/*  NUCLUS.EXT declares all system calls */
$INCLUDE(/rmx286/inc/NUCLUS.EXT)

TASKCODE: PROCEDURE EXTERNAL;
END TASKCODE;

   DECLARE task$token           TOKEN;
   DECLARE priority$level$210   LITERALLY '210';
   DECLARE start$address        POINTER;
   DECLARE data$seg             TOKEN;
   DECLARE stack$pointer        POINTER;
   DECLARE stack$size$512       LITERALLY '512';  /* new task's stack
                                                     size is 512 bytes */
    DECLARE task$flags           WORD;
    DECLARE status               WORD;

SAMPLEPROCEDURE:  PROCEDURE;

    start$address = @TASKCODE;     /* points to first instruction of
                                      the new task */
    data$seg = SELECTOR$OF(NIL);   /* task sets up own data segment */
    stack$pointer = NIL;           /* automatic stack allocation  */
    task$flags = 0;                /* indicates no floating-point
                                      instructions */

        •
        •     Typical PL/M-286 Statements
        •

/*****************************************************************
 *  In order to delete a task, a task must know the token for that  *
 *  task.  In this example, the needed token is known because the   *
 *  calling task creates the new task (The task's code is labeled   *
 *  TASKCODE).                                                      *
*****************************************************************/

    task$token = RQ$CREATE$TASK  (priority$level$210,
                                  start$address,
                                  data$seg,
                                  stack$pointer,
                                  stack$size$512,
                                  task$flags,
                                  @status);

        •
        •     Typical PL/M-286 Statements
        •
```

```
/*************************************************************************
 *  The calling task has created a task (whose code is labeled      *
 *  TASKCODE) which is not an interrupt task.  When this task is no *
 *  longer needed, it may be deleted by any task that knows its     *
 *  token.                                                          *
 *************************************************************************/

   CALL RQ$DELETE$TASK (task$token, @status);

        •
        •     Typical PL/M-286 Statements
        •

END SAMPLEPROCEDURE;
```

Condition Codes

E$OK	0000H	No exceptional conditions.
E$CONTEXT	0005H	The task parameter is a token for an interrupt task.
E$EXIST	0006H	One of the following conditions has occurred: • The task parameter is not a token for an existing object. • The task parameter represents a task whose job is being deleted. • More than one task is trying to delete a task which is in a region.
ENOTCONFIGURED	0008H	This system call is not part of the present configuration.
E$TYPE	8002H	The task parameter is a token for an object which is not a task.

The GET$PRIORITY system call returns the priority of a task.

```
priority = RQ$GET$PRIORITY (task, except$ptr);
```

Input Parameter

task — A TOKEN that specifies the task whose priority is being requested.

- If a valid selector, the TOKEN must contain a token for the task whose priority is being requested.
- If SELECTOR$OF(NIL), the calling task is asking for its own priority.

Output Parameters

priority — A BYTE in which the system call returns the priority of the task indicated by the task parameter.

except$ptr — A POINTER to a WORD to which the iRMX II Operating System will return the condition code generated by this system call.

Description

The GET$PRIORITY system call returns the iRMX II priority of the specified task.

Example

```
/*************************************************************************
 *  This example illustrates how the GET$PRIORITY system call can be     *
 *  used.                                                                *
 *************************************************************************/

DECLARE TOKEN      LITERALLY 'SELECTOR';

/*  NUCLUS.EXT declares all system calls */
$INCLUDE(/rmx286/inc/NUCLUS.EXT)

    DECLARE priority                BYTE;
    DECLARE calling$task            TOKEN;
    DECLARE status                  WORD;
```

GET$PRIORITY

```
SAMPLEPROCEDURE:
    PROCEDURE;

        •
        •     Typical PL/M-286 Statements
        •

/*************************************************************************
 *  The GET$PRIORITY system call returns the priority of the calling    *
 *  task.                                                                *
 *************************************************************************/

    calling$task = SELECTOR$OF(NIL);   /* Select calling task. */

    priority = RQ$GET$PRIORITY    (calling$task,
                                   @status);

        •
        •     Typical PL/M-286 Statements
        •

END SAMPLEPROCEDURE;
```

Condition Codes

E$OK	0000H	No exceptional conditions.
E$EXIST	0006H	The task parameter is not a token for an existing object.
ENOTCONFIGURED	0008H	This system call is not part of the present configuration.
E$TYPE	8002H	The task parameter is a token for an object that is not a task.

The GET$TASK$TOKENS system call returns the token requested by the calling task.

```
object = RQ$GET$TASK$TOKENS (selection, except$ptr);
```

Input Parameter

selection — A BYTE that tells the iRMX II Operating System what information is desired. It is encoded as follows:

Value	Object for which a Token is Requested
0	The calling task.
1	The calling task's job.
2	The parameter object of the calling task's job.
3	The root job.

Output Parameters

object — A TOKEN to which the iRMX II Operating System will return the requested token.

except$ptr — A POINTER to a WORD to which the iRMX II Operating System will return the condition code generated by this system call.

Description

The GET$TASK$TOKENS system call returns a token for either the calling task, the calling task's job, the parameter object of the calling task's job, or the root job, depending on the encoded request.

Example

```
/*************************************************************************
 *  This example illustrates how the GET$TASK$TOKENS system call can be *
 *  used to return the TOKEN requested by the calling task.             *
 *************************************************************************/

DECLARE TOKEN       LITERALLY 'SELECTOR';
```

GET$TASK$TOKENS

```
/*  NUCLUS.EXT declares all system calls */
$INCLUDE(/rmx286/inc/NUCLUS.EXT)

    DECLARE task$token                  TOKEN;
    DECLARE calling$task                LITERALLY '0';
    DECLARE status                      WORD;

SAMPLEPROCEDURE:
    PROCEDURE;

        •
        •     Typical PL/M-286 Statements
        •

/*************************************************************************
 *  If you set the selection parameter to zero, the GET$TASK$TOKENS     *
 *  system call will return a token for the calling task.               *
 *************************************************************************/

    task$token = RQ$GET$TASK$TOKENS     (calling$task,
                                         @status);
        •
        •     Typical PL/M-286 Statements
        •

END SAMPLEPROCEDURE;
```

Condition Codes

E$OK	0000H	No exceptional conditions.
E$PARAM	8004H	The selection parameter is greater than 3.

The RESUME$TASK system call decreases by one the suspension depth of a task.

```
CALL RQ$RESUME$TASK (task, except$ptr);
```

Input Parameter

task	A TOKEN for the task whose suspension depth is to be decremented.

Output Parameter

except$ptr	A POINTER to a WORD to which the iRMX II Operating System will return the condition code generated by this system call.

Description

The RESUME$TASK system call decreases by one the suspension depth of the specified non-interrupt task. The task should be in either the suspended or asleep-suspended state, so its suspension depth should be at least one. If the suspension depth is still positive after being decremented, the state of the task is not changed. If the depth becomes zero, and the task is in the suspended state, then it is placed in the ready state. If the depth becomes zero, and the task is in the asleep-suspended state, then it is placed in the asleep state.

Example

```
/*************************************************************************
 *  This example illustrates how the RESUME$TASK system call can be      *
 *  used to decrease by one the suspension depth of a task.              *
 *************************************************************************/

DECLARE TOKEN       LITERALLY 'SELECTOR';

/*  NUCLUS.EXT declares all system calls */
$INCLUDE(/rmx286/inc/NUCLUS.EXT)

TASKCODE: PROCEDURE EXTERNAL;
END TASKCODE;

    DECLARE task$token              TOKEN;
    DECLARE priority$level$200      LITERALLY '200';
    DECLARE start$address           POINTER;
```

RESUME$TASK

```
    DECLARE data$seg              TOKEN;
    DECLARE stack$pointer         POINTER;
    DECLARE stack$size$512        LITERALLY '512'; /* new task's stack
                                                      size is 512 bytes */
    DECLARE task$flags            WORD;
    DECLARE status                WORD;

SAMPLEPROCEDURE:
    PROCEDURE;

    start$address = @TASKCODE;   /* first instruction of the new task */
    data$seg = SELECTOR$OF(NIL); /* task sets up own data seg         */
    stack$pointer = NIL;          /* automatic stack allocation        */
    task$flags = 0;               /* indicates no floating-point
                                     instructions                      */
        •
        •     Typical PL/M-286 Statements
        •

/************************************************************************
 *  In this example the calling task creates a non-interrupt task and   *
 *  suspends that task before invoking the RESUME$TASK system call.     *
 ************************************************************************/

    task$token = RQ$CREATE$TASK   (priority$level$200,
                                   start$address,
                                   data$seg,
                                   stack$pointer,
                                   stack$size$512,
                                   task$flags,
                                   @status);
        •
        •     Typical PL/M-286 Statements
        •
/************************************************************************
 *  After creating the task, the calling task invokes SUSPEND$TASK.     *
 *  This system call increases by one the suspension depth of the new   *
 *  task (whose code is labeled TASKCODE).                              *
 ************************************************************************/

    CALL RQ$SUSPEND$TASK          (task$token,
                                   @status);
        •
        •     Typical PL/M-286 Statements
        •
```

```
/*************************************************************************
 *  Using the token for the suspended task (whose code is labeled       *
 *  TASKCODE), the calling task invokes RESUME$TASK to decrease by the *
 *  one the suspension depth of the suspended task.                      *
 *************************************************************************/
   CALL RQ$RESUME$TASK           (task$token,
                                  @status);

        •
        •     Typical PL/M-286 Statements
        •

END SAMPLEPROCEDURE;
```

Condition Codes

E$OK	0000H	No exceptional conditions.
E$CONTEXT	0005H	The task indicated by the task parameter is an interrupt task.
E$EXIST	0006H	The task parameter is not a token for an existing object.
E$STATE	0007H	The task indicated by the task parameter was not suspended when the call was made.
E$TYPE	8002H	The task parameter is a token for an object that is not a task.

The SET$PRIORITY system call changes the priority of a task.

CAUTION

Tasks can become blocked for long periods of time, and real-time performance of the iRMX II Operating System can be degraded when a task uses this system call to lower its own priority.

```
CALL RQ$SET$PRIORITY (task, priority, except$ptr);
```

Input Parameters

task	A TOKEN for the task whose priority is to be changed. Setting this parameter to SELECTOR$OF(NIL) selects the invoking task.
priority	A BYTE containing the task's new priority. A zero value specifies the maximum priority of the specified task's containing job.

Output Parameter

except$ptr	A POINTER to a WORD to which the iRMX II Operating System will return the condition code generated by this system call.

Description

The SET$PRIORITY system call allows the priority of a non-interrupt task to be altered dynamically. If the priority parameter is set to zero, the task's new priority is its containing job's maximum priority. Otherwise, the priority parameter contains the new priority of the specified task. The new priority, if explicitly specified, must not exceed its containing job's maximum priority.

Example

```
/*************************************************************************
 *  This example illustrates how the SET$PRIORITY system call can be     *
 *  used to change the priority of a task.                               *
 *************************************************************************/

DECLARE TOKEN       LITERALLY 'SELECTOR';
```

```
/*  NUCLUS.EXT declares all system calls */
$INCLUDE(/rmx286/inc/NUCLUS.EXT)

 TASKCODE: PROCEDURE EXTERNAL;
 END TASKCODE;

    DECLARE task$token                  TOKEN;
    DECLARE priority$level$66           LITERALLY '66';
    DECLARE priority$level$210          LITERALLY '210';
    DECLARE start$address               POINTER;
    DECLARE data$seg                    TOKEN;
    DECLARE stack$pointer               POINTER;
    DECLARE stack$size$512              LITERALLY '512'; /* new task's
                                                            stack size is
                                                            512 bytes */
    DECLARE task$flags                  WORD;
    DECLARE status                      WORD;
    DECLARE job$token                   TOKEN;

SAMPLEPROCEDURE:
    PROCEDURE;
    start$address = @TASKCODE;       /* pointer to first instruction of
                                        interrupt task */
    data$seg = SELECTOR$OF(NIL);      /* task sets up own data segment */
    stack$pointer = NIL;              /* automatic stack allocation  */
    task$flags = 0;                   /* designates no floating-point
                                        instructions */
       •
       •     Typical PL/M-286 Statements
       •

/*************************************************************************
 *  In this example, the calling task creates a task whose priority is  *
 *  to be changed.  The new task initially has a priority level 66.     *
 *************************************************************************/

    task$token = RQ$CREATE$TASK   (priority$level$66,
                                   start$address,
                                   data$seg,
                                   stack$pointer,
                                   stack$size$512,
                                   task$flags,
                                   @status);

/*************************************************************************
 *  The calling task in this example does not need to invoke the        *
 *  CATALOG$OBJECT system call to ensure the successful use of the      *
 *  SET$PRIORITY system call.  To allow other tasks access to the new   *
 *  task, however, requires that the task's object token be cataloged.  *
 *************************************************************************/
```

```
    CALL RQ$CATALOG$OBJECT          (job$token,
                                    task$token,
                                    @(9, 'TASKCODE'),
                                    @status);
        •
        •     Typical PL/M-286 Statements
        •

/************************************************************************
 *  The new task (whose code is labeled TASKCODE) is not an interrupt  *
 *  task, so its priority may be changed dynamically by invoking the    *
 *  SET$PRIORITY system call.                                           *
 ************************************************************************/

    CALL RQ$SET$PRIORITY            (task$token,
                                    priority$level$210,
                                    @status);
        •
        •     Typical PL/M-286 Statements
        •

/************************************************************************
 *  Once the need for the higher priority is no longer present, the     *
 *  priority of the new task can be changed back to its original        *
 *  priority by invoking SET$PRIORITY a second time.                    *
 ************************************************************************/

    CALL RQ$SET$PRIORITY            (task$token,
                                    priority$level$66,
                                    @status);
    •
    •    Typical PL/M-286 Statements
    •

END SAMPLEPROCEDURE;
```

Condition Codes

E$OK	0000H	No exceptional conditions.
E$CONTEXT	0005H	The specified task is an interrupt task. You cannot set the priority of an interrupt task dynamically.
E$EXIST	0006H	The task parameter is not a token for an existing object.

E$LIMIT	0004H	The priority parameter contains a priority value that is higher than the maximum priority of the specified task's containing job.
ENOTCONFIGURED	0008H	This system call is not part of the present configuration.
E$TYPE	8002H	The task parameter is a token for an object that is not a task.

The SLEEP system call puts the calling task to sleep.

```
CALL RQ$SLEEP (time$limit, except$ptr);
```

Input Parameter

time$limit — A WORD indicating the conditions in which the calling task is to be put to sleep.

- If not zero and not 0FFFFH, causes the calling task to go to sleep for that many clock intervals, after which it will be awakened. The length of a clock interval is configurable. Refer to the *Extended iRMX II Interactive Configuration Utility Reference Manual* for further information.
- If zero, causes the calling task to be placed on the list of ready tasks, immediately behind all tasks of the same priority. If there are no such tasks, there is no effect and the calling task continues to run.
- If 0FFFFH, an error is returned.

Output Parameter

except$ptr — A POINTER to a WORD to which the iRMX II Operating System will return the condition code generated by this system call.

Description

The SLEEP system call has two uses. One use places the calling task in the asleep state for a specific amount of time. The other use allows the calling task to defer to the other ready tasks with the same priority. When a task defers in this way it is placed on the list of ready tasks, immediately behind those other tasks of equal priority.

Example

```
/*************************************************************************
 *  This example illustrates how the SLEEP system call can be used.      *
 *************************************************************************/

DECLARE TOKEN      LITERALLY 'SELECTOR';

/*  NUCLUS.EXT declares all system calls */
$INCLUDE(/rmx286/inc/NUCLUS.EXT)

    DECLARE time$limit            WORD;
    DECLARE status                WORD;

SAMPLEPROCEDURE:
    PROCEDURE;

    time$limit = 100;             /* sleep for 100 clock ticks */

        •
        •     Typical PL/M-286 Statements
        •

/*************************************************************************
 *  The calling task puts itself in the asleep state for 100 clock       *
 *  ticks by invoking the SLEEP system call.                             *
 *************************************************************************/

    CALL RQ$SLEEP                 (time$limit,  /*  10ms is the default */
                                   @status);    /*   100 = 1 second     */

        •
        •     Typical PL/M-286 Statements
        •

END SAMPLEPROCEDURE;
```

Condition Codes

E$OK	0000H	No exceptional conditions.
ENOTCONFIGURED	0008H	This system call is not part of the present configuration.
E$PARAM	8004H	The time$limit parameter contains the invalid value 0FFFFH.

The SUSPEND$TASK system call increases by one the suspension depth of a task.

```
CALL RQ$SUSPEND$TASK (task, except$ptr);
```

Input Parameter

task — A TOKEN specifying the task whose suspension depth is to be incremented.

- if a valid selector, contains a token for the task whose suspension depth is to be incremented.
- if SELECTOR$OF(NIL), indicates that the calling task is suspending itself.

Output Parameter

except$ptr — A POINTER to a WORD to which the iRMX II Operating System will return the condition code generated by this system call.

Description

The SUSPEND$TASK system call increases by one the suspension depth of the specified task. If the task is already in either the suspended or asleep-suspended state, its state is not changed. If the task is in the ready or running state, it enters the suspended state. If the task is in the asleep state, it enters the asleep-suspended state.

SUSPEND$TASK cannot be used to suspend interrupt tasks.

Example

```
/*************************************************************************
 *  This example illustrates how the SUSPEND$TASK system call can be    *
 *  used to increase the suspension depth of a non-interrupt task.      *
 *************************************************************************/

DECLARE TOKEN       LITERALLY 'SELECTOR';

/*  NUCLUS.EXT declares all system calls */
$INCLUDE(/rmx286/inc/NUCLUS.EXT)
```

```
TASKCODE: PROCEDURE EXTERNAL;
END TASKCODE;

    DECLARE task$token            TOKEN;
    DECLARE priority$level$200    LITERALLY '200';
    DECLARE start$address         POINTER;
    DECLARE data$seg              TOKEN;
    DECLARE stack$pointer         POINTER;
    DECLARE stack$size$512        LITERALLY '512'; /* new task's stack
                                                      size is 512 bytes */
    DECLARE task$flags            WORD;
    DECLARE status                WORD;

SAMPLEPROCEDURE:
    PROCEDURE;
    start$address = @TASKCODE;   /* first instruction of the new task */
    data$seg = SELECTOR$OF(NIL);  /* task sets up own data seg */
    stack$pointer = NIL;          /* automatic stack allocation  */
    task$flags = 0;               /* designates no floating-point
                                     instructions */

        •
        •     Typical PL/M-286 Statements
        •

 ***************************************************************************
 *  In order to suspend a task, a task must know the token for that      *
 *  task.  In this example, the needed token is known because the        *
 *  calling task creates the new task (whose code is labeled TASKCODE).  *
 ***************************************************************************/

    task$token = RQ$CREATE$TASK   (priority$level$200,
                                   start$address,
                                   data$seg,
                                   stack$pointer,
                                   stack$size$512,
                                   task$flags,
                                   @status);

        •
        •     Typical PL/M-286 Statements
        •

/***************************************************************************
 *  After creating the task, the calling task invokes SUSPEND$TASK.      *
 *  This system call increases by one the suspension depth of the new    *
 *  task (whose code is labeled TASKCODE).                                *
 ***************************************************************************/

    CALL RQ$SUSPEND$TASK (task$token, @status);
```

```
        •
        •     Typical PL/M-286 Statements
        •
END SAMPLEPROCEDURE;
```

Condition Codes

E$OK	0000H	No exceptional conditions.
E$CONTEXT	0005H	The specified task is an interrupt task. You cannot suspend interrupt tasks.
E$EXIST	0006H	The task parameter is not a token for an existing object.
E$LIMIT	0004H	The suspension depth for the specified task is already at the maximum of 255.
E$TYPE	8002H	The task parameter is a token for an object that is not a task.

The CREATE$MAILBOX system call creates a mailbox.

```
mailbox = RQ$CREATE$MAILBOX (mailbox$flags, except$ptr);
```

Input Parameters

mailbox$flags — A WORD containing information about the new mailbox. The bits (where bit 15 is the high-order bit) have the following meanings:

Bits	Meaning
15-6	Reserved bits which should be set to zero.
5	A bit that determines the type of messages that this mailbox can handle, as follows:

Value	Message Scheme
0	This mailbox passes iRMX II objects. The SEND$MESSAGE and RECEIVE$MESSAGE system calls can be used to send and receive objects.
1	This mailbox passes up to 128 bytes of data. The SEND$DATA and RECEIVE$DATA system calls can be used to send and receive the data.

Bits	Meaning
4-1	A value that, when multiplied by four, specifies the number of messages that can be queued on the high performance object queue. Eight is the minimum size for the high performance queue; that is, specifying a value less than eight in these bits results in a high performance queue that holds eight objects. These four bits have meaning only when the mailbox is set up to pass object tokens (not data). When the mailbox is set up to pass data, the Operating System ignores these bits and automatically sets up a queue that is three messages long, each message is 128 bytes in length.

0	A bit that determines the queuing scheme for the task queue of the new mailbox, as follows:

Value	Queuing Scheme
0	First-in/first-out
1	Priority based

Output Parameters

mailbox	A TOKEN to which the Operating System will return a token for the new mailbox.
except$ptr	A POINTER to a WORD to which the iRMX II Operating System will return the condition code generated by this system call.

Description

This system call creates a mailbox, an exchange that tasks can use to exchange messages. There are two kinds of mailboxes that you can create, depending on the kind of messages your tasks wish to exchange. Bit 5 in the mailbox$flags parameter specifies the kind of mailbox to create.

If you set bit 5 to 0, the mailbox is set up to pass iRMX II objects between tasks. That is, if your message is a block of data, you must set it up as an iRMX II segment first. You can pass other kinds of iRMX II objects as messages too. To send the message, use the token for that object as input to the SEND$MESSAGE system call. The RECEIVE$MESSAGE system call can be used to receive object tokens from a mailbox.

If you set bit 5 to 1, the mailbox is set up to pass data. Instead of creating an iRMX II object for your message, you can use the SEND$DATA system call to pass up to 80H bytes of data from a user-supplied buffer. RECEIVE$DATA can be used to receive a message from a mailbox and place it into another user-supplied buffer. Passing data instead of objects can be important to systems whose Global Descriptor Table (GDT) is almost full, because each object you create requires an entry in the GDT. Of course, you can pass only 80H bytes of data per message. But the data can be in the form of a pointer, which can point to an area larger than 128 bytes.

Each mailbox you create can be used in only one way. That is, a mailbox set up to pass objects can pass only objects, not data. A mailbox set up to pass data cannot pass objects.

When you set up a mailbox to pass objects, you can also specify the size of a high-performance queue that is associated with the mailbox. This queue is a block of memory that stores objects waiting to be sent or received. It is permanently assigned to the mailbox, even if no objects are queued there. If the queue overflows, the Nucleus temporarily allocates another 200-object queue.

Setting the size of the high-performance queue involves a tradeoff between memory and performance. Setting a size that is too large wastes memory, because the unused portion of the queue is unavailable for other uses. But setting a size that is too small forces the Nucleus to create a temporary queue (and creating and deleting objects are relatively slow operations). You should set up a high-performance queue large enough to contain all the objects queued during normal operations, and let the overflow queue handle large overflows or unusual circumstances.

If you create a mailbox that passes data, you don't specify the size of the message queue. The Operating System automatically sets up the queue to an appropriate size of 400 decimal bytes.

Example

```
/*****************************************************************
 *  This example illustrates how the CREATE$MAILBOX system call  *
 *   can be used.                                                *
*****************************************************************/

DECLARE TOKEN                   LITERALLY 'SELECTOR';

/* NUCLUS.EXT declares all system calls */

$INCLUDE(/rmx286/inc/NUCLUS.EXT)
    DECLARE mbx$token             TOKEN;
    DECLARE mbx$flags             WORD;
    DECLARE status                WORD;
SAMPLEPROCEDURE:
    PROCEDURE;
    mbx$flags = 0;              /* designates a high performance
                                   object queue of eight objects;
                                   designates a first-in/first-out
                                   task queue. */

    •
    •     Typical PL/M-286 Statements
    •

/*****************************************************************
 * The token mbx$token is returned when the calling task invokes  *
 * the CREATE$MAILBOX system call.                                *
*****************************************************************/
```

```
    mbx$token = RQ$CREATE$MAILBOX (mbx$flags,
                                   @status);

    •
    •     Typical PL/M-286 Statements
    •

END SAMPLEPROCEDURE;
```

Condition Codes

E$OK	0000H	No exceptional conditions.
E$LIMIT	0004H	The calling task's job has already reached its object limit.
E$MEM	0002H	The memory available to the calling task's job is not sufficient to create a mailbox.
ENOTCONFIGURED	0008H	This system call is not part of the present configuration.
E$SLOT	000CH	There isn't enough room in the GDT for the new job and task descriptors

The DELETE$MAILBOX system call deletes a mailbox.

```
CALL RQ$DELETE$MAILBOX (mailbox, except$ptr);
```

Input Parameter

mailbox	A TOKEN for the mailbox to be deleted.

Output Parameters

except$ptr	A POINTER to a WORD to which the iRMX II Operating System will return the condition code generated by this system call.

Description

The DELETE$MAILBOX system call deletes the specified mailbox. If any tasks are queued at the mailbox at the moment of deletion, they are awakened with an E$EXIST exceptional condition. If there is a queue of object tokens data messages at the moment of deletion, the queue is discarded. Deleting the mailbox counts as a credit of one toward the object total of the containing job.

Example

```
/*************************************************************************
 *  This example illustrates how the DELETE$MAILBOX system call can *
 * be used.                                                              *

*************************************************************************/

DECLARE TOKEN                   LITERALLY 'SELECTOR';

/*  NUCLUS.EXT declares all system calls */
$INCLUDE(/rmx286/inc/NUCLUS.EXT)

    DECLARE mbx$token           TOKEN;
    DECLARE mbx$flags           WORD;
    DECLARE status              WORD;

SAMPLEPROCEDURE:
    PROCEDURE;
```

```
    mbx$flags = 0;          /* designates eight objects to be queued
                               on the high performance object
                               queue;  designates a first-in/
                               first-out task queue;  designates a
                               mailbox that passes tokens. */
        •
        •  Typical PL/M-286 Statements
        •

/*************************************************************************
 *  In order to delete a mailbox, a task must know the token for    *
 * that mailbox.  In this example, the needed token is known        *
 * because the calling task creates the mailbox.                    *

*************************************************************************/

    mbx$token = RQ$CREATE$MAILBOX (mbx$flags, @status);

        •
        •  Typical PL/M-286 Statements
        •

/*************************************************************************
 *  When the mailbox is no longer needed, it may be deleted by      *
 *  any task that knows the token for the mailbox.                  *

*************************************************************************/

    CALL RQ$DELETE$MAILBOX (mbx$token, @status);

        •
        •  Typical PL/M-286 Statements
        •

END SAMPLEPROCEDURE;
```

Condition Codes

E$OK	0000H	No exceptional conditions.
E$EXIST	0006H	Either the mailbox parameter is not a token for an existing object or it represents a mailbox whose job is in the process of being deleted.
ENOTCONFIGURED	0008H	This system call is not part of the present configuration.
E$TYPE	8002H	The mailbox parameter is a token for an object which is not a mailbox.

The RECEIVE$DATA system call delivers the calling task to a mailbox, where it can wait for a message to be returned.

```
actual = RQ$RECEIVE$DATA (mailbox, message$ptr, time$limit,
                          except$ptr);
```

Input Parameters

mailbox	A TOKEN for the mailbox from which the calling task expects to receive a message.
time$limit	A WORD that indicates how long the calling task is willing to wait. • If zero, indicates that the calling task is not willing to wait. • If 0FFFFH, indicates that the task will wait as long as is necessary. • If between 0 and 0FFFFH, indicates the number of clock intervals that the task is willing to wait. The length of a clock interval is configurable. Refer to the *Extended iRMX II Interactive Configuration Utility Reference Manual* for further information.

Output Parameters

actual	A WORD in which the Operating System returns the number of bytes actually received.
message$ptr	A POINTER to the start of a user-supplied buffer. The system call places the message into this buffer. The maximum message length is 128 bytes, so this buffer should be large enough to contain messages of that length.
except$ptr	A POINTER to a WORD to which the iRMX II Operating System will return the condition code generated by this system call.

RECEIVE$DATA

Description

The RECEIVE$DATA system call receives messages from mailboxes that have been set up to pass data (rather than tokens). It causes the calling task either to receive the data message or to wait for the data in the task queue of the specified mailbox. If the message queue at the mailbox is not empty, then the calling task immediately receives the message at the head of the queue and remains ready. Otherwise, the calling task goes into the task queue of the mailbox and goes to sleep, unless the task is not willing to wait. In the latter case, or if the task's waiting period elapses without a data message arriving, the task is awakened with an E$TIME exceptional condition.

When you create a mailbox with CREATE$MAILBOX, you can specify whether the mailbox will be used to pass object tokens or data. RECEIVE$DATA functions only with those mailboxes that have been set up to pass data. RECEIVE$DATA returns the message data (up to a maximum of 128 bytes) in a user-specified memory buffer. The system call also returns the length of the actual message received.

Example

```
/*************************************************************************
 *  This example illustrates how the RECEIVE$DATA system call can be     *
 *  used to receive a message segment.                                   *
*************************************************************************/

DECLARE TOKEN       LITERALLY 'SELECTOR';

/*  NUCLUS.EXT declares all system calls */
$INCLUDE(/rmx286/inc/NUCLUS.EXT)
    DECLARE mbx$token              TOKEN;
    DECLARE calling$tasks$job      TOKEN;
    DECLARE wait$forever           LITERALLY '0FFFFH';
    DECLARE message(80H)           BYTE;
    DECLARE status                 WORD;
    DECLARE actual                 WORD;

SAMPLEPROCEDURE:
    PROCEDURE;

        •
        •     Typical PL/M-286 Statements
        •

/*************************************************************************
 *  In this example, the calling task looks up the token for the mailbox*
 *  prior to invoking the RECEIVE$DATA system call.                      *
 *************************************************************************/

    calling$tasks$job = SELECTOR$OF(NIL);
```

```
    mbx$token = RQ$LOOKUP$OBJECT  (calling$tasks$job,
                                   @(3,'MBX'),
                                   wait$forever,
                                   @status);
        •
        •     Typical PL/M-286 Statements
        •

/*************************************************************************
 *  Knowing the token for the mailbox, the calling task can wait for a  *
 *  message from this mailbox by invoking the RECEIVE$DATA system       *
 *  call.                                                               *
 *************************************************************************/
    actual = RQ$RECEIVE$DATA      (mbx$token,
                                   @message,
                                   wait$forever,
                                   @status);
        •
        •     Typical PL/M-286 Statements
        •
END SAMPLEPROCEDURE;
```

Condition Codes

E$OK	0000H	No exceptional conditions.
E$EXIST	0006H	The mailbox parameter is not a token for an existing object.
ENOTCONFIGURED	0008H	This system call is not part of the present configuration.
E$TIME	0001H	One of the following is true: • The calling task was not willing to wait and there was no message available. • The task waited in the task queue and its designated waiting period elapsed before the task got the desired message.
E$TYPE	8002H	One of the following is true: • The mailbox parameter contains a token for an object that is not a mailbox. • The mailbox has been set up to pass tokens, not data.

The RECEIVE$MESSAGE system call delivers the calling task to a mailbox, where it can wait for an object token to be returned.

```
object = RQ$RECEIVE$MESSAGE (mailbox, time$limit, response$ptr,
                             except$ptr);
```

Input Parameters

mailbox — A TOKEN for the mailbox at which the calling task expects to receive an object token.

time$limit — A WORD that indicates how long the calling task is willing to wait.

- If zero, indicates that the calling task is not willing to wait.
- If 0FFFFH, indicates that the task will wait as long as is necessary.
- If between 0 and 0FFFFH, indicates the number of clock intervals that the task is willing to wait. The length of a clock interval is configurable. Refer to the *Extended iRMX II Interactive Configuration Utility Reference Manual* for further information.

Output Parameters

object — A TOKEN for the object being received.

response$ptr — A POINTER to a TOKEN in which the system returns a value. The returned pointer:

- if a valid pointer, points to a token for the exchange to which the receiving task is to send a response.
- if NIL, indicates that no response is expected by the sending task.

except$ptr — A POINTER to a WORD to which the iRMX II Operating System will return the condition code generated by this system call.

Description

The RECEIVE$MESSAGE system call causes the calling task either to get the token for an object or to wait for the token in the task queue of the specified mailbox. This mailbox must be set up to pass objects. If the object queue at the mailbox is not empty, then the calling task immediately gets the token at the head of the queue and remains ready. Otherwise, the calling task goes into the task queue of the mailbox and goes to sleep, unless the task is not willing to wait. In the latter case, or if the task's waiting period elapses without a token arriving, the task is awakened with an E$TIME exceptional condition.

When you create a mailbox with CREATE$MAILBOX, you can specify whether the mailbox will be used to pass object tokens or data. RECEIVE$MESSAGE functions only with those mailboxes that have been set up to pass objects.

It is possible that the token returned by RECEIVE$MESSAGE is a token for an object that has already been deleted. To verify that the token is valid, the receiving task can invoke the GET$TYPE system call. However, tasks can avoid this situation by adhering to proper programming practices.

One such practice is for the sending task to request a response from the receiving task and not delete the object until it gets a response. When the receiving task finishes with the object, it sends a response, the nature of which must be determined by the writers of the two tasks, to the response mailbox. When the sending task gets this response, it can then delete the original object, if it so desires.

Example

```
/*************************************************************************
 *  This example illustrates how the RECEIVE$MESSAGE system call can be *
 *  used to receive a message segment.                                   *
*************************************************************************/

DECLARE TOKEN       LITERALLY 'SELECTOR';

/*  NUCLUS.EXT declares all system calls */
$INCLUDE(/rmx286/inc/NUCLUS.EXT)
    DECLARE mbx$token             TOKEN;
    DECLARE calling$tasks$job     TOKEN;
    DECLARE wait$forever          LITERALLY '0FFFFH';
    DECLARE seg$token             TOKEN;
    DECLARE response              TOKEN;
    DECLARE status                WORD;
SAMPLEPROCEDURE:
    PROCEDURE;
```

```
        •
        •     Typical PL/M-286 Statements
        •

/************************************************************************
 *  In this example the calling task looks up the token for the mailbox *
 *  prior to invoking the RECEIVE$MESSAGE system call.                  *
 ************************************************************************/
   calling$tasks$job = SELECTOR$OF(NIL);
   mbx$token = RQ$LOOKUP$OBJECT  (calling$tasks$job,
                                  @(3,'MBX'),
                                  wait$forever,
                                  @status);

        •
        •     Typical PL/M-286 Statements
        •

/************************************************************************
 *  Knowing the token for the mailbox, the calling task can wait for a  *
 *  message from this mailbox by invoking the RECEIVE$MESSAGE system    *
 *  call.                                                               *
 ************************************************************************/

   seg$token = RQ$RECEIVE$MESSAGE     (mbx$token,
                                       wait$forever,
                                       @response,
                                       @status);

        •
        •     Typical PL/M-286 Statements
        •

END SAMPLEPROCEDURE;
```

Condition Codes

E$OK	0000H	No exceptional conditions.
E$EXIST	0006H	The mailbox parameter is not a token for an existing object.
ENOTCONFIGURED	0008H	This system call is not part of the present configuration.

E$TIME	0001H	One of the following is true: • The calling task was not willing to wait and there was not a token available. • The task waited in the task queue and its designated waiting period elapsed before the task got the desired token.
E$TYPE	8002H	One of the following is true: • The mailbox parameter contains a token for an object that is not a mailbox. • The mailbox was set up to pass data messages, not objects.

The SEND$DATA system call sends a message of up to 80H bytes to a mailbox.

```
CALL RQ$SEND$DATA (mailbox, message$ptr, actual$length, except$ptr);
```

Input Parameters

mailbox	A TOKEN for the mailbox to which the message is to be sent. This mailbox must be one that was created to pass data, not objects.
message$ptr	A POINTER to a memory buffer containing the message.
actual$length	A WORD specifying the length of the message. Any value between 0 and 0FFFFH can be specified. However, because messages are limited to 80H bytes, any value over 80H causes only 80H bytes to be sent.

Output Parameter

except$ptr	A POINTER to a WORD to which the iRMX II Operating System will return the condition code generated by this system call.

Description

The SEND$DATA system call sends messages to mailboxes that have been set up to pass data. It sends to a specified mailbox a maximum of 80H bytes from a user-specified buffer. The number of bytes actually sent is also specified in the SEND$DATA call. If there are tasks in the task queue at that mailbox, the task at the head of the queue is awakened and is given the data. Otherwise, the message data is placed at the tail of the mailbox's message queue.

When you create a mailbox with CREATE$MAILBOX, you can specify whether the mailbox will be used to pass object tokens or data. SEND$DATA functions only with those mailboxes that have been set up to pass data.

Example

```
/*************************************************************************
 *  This example illustrates how the SEND$DATA system call can be       *
 *  used to send data to a mailbox.                                      *
 *************************************************************************/

DECLARE TOKEN       LITERALLY 'SELECTOR';
```

```
/*  NUCLUS.EXT declares all system calls */
$INCLUDE(/rmx286/inc/NUCLUS.EXT)

    DECLARE msg$ptr       POINTER;
    DECLARE size          WORD;
    DECLARE mbx$token     TOKEN;
    DECLARE mbx$flags     WORD;
    DECLARE status        WORD;
    DECLARE job$token     TOKEN;

SAMPLEPROCEDURE:
    PROCEDURE;
    mbx$flags = 20H;                /* designates a data mailbox */

    job$token = SELECTOR$OF(NIL);  /* indicates objects to be cataloged
                                      into the object directory of the
                                      calling task's job */
        •
        •     Typical PL/M-286 Statements
        •

/*************************************************************************
 *  The calling task creates a mailbox and catalogs the mailbox token.  *
 *  The calling task then sends message data to the mailbox.            *
 *************************************************************************/

    mbx$token = RQ$CREATE$MAILBOX  (mbx$flags,
                                    @status);

/*************************************************************************
 *  It is not mandatory for the calling task to catalog the mailbox     *
 *  token in order to send a message.  It is necessary, however, to     *
 *  catalog (or in someway communicate) the mailbox token if another    *
 *  task is to receive the message.                                     *
 *************************************************************************/

    CALL RQ$CATALOG$OBJECT            (job$token,
                                       mbx$token,
                                       @(3, 'MBX'),
                                       @status);
        •
        •     Typical PL/M-286 Statements
        •

/*************************************************************************
 *  The calling task invokes the SEND$DATA system call to send a        *
 *  message to the specified mailbox.                                   *
 *************************************************************************/
```

```
    CALL RQ$SEND$DATA                (mbx$token,
                                      @('this is a message '),
                                      18,
                                      @status);
        •
        •     Typical PL/M-286 Statements
        •

END SAMPLEPROCEDURE;
```

Condition Codes

E$OK	0000H	No exceptional conditions.
EBADADDR	800FH	The pointer to the message is invalid. Either the selector does not refer to a valid segment, or the offset is outside the segment boundaries.
E$EXIST	0006H	The mailbox token is not a token for an existing object.
E$MEM	0002H	The data message queue is full and the system does not have enough memory to create another.
ENOTCONFIGURED	0008H	This system call is not part of the present configuration.
E$TYPE	8002H	Either one of the following is true: • The mailbox parameter is a token for an object that is not a mailbox. • The specified mailbox was set up to pass tokens, not data.

The SEND$MESSAGE system call sends an object token to a mailbox.

```
CALL RQ$SEND$MESSAGE (mailbox, object, response, except$ptr);
```

Input Parameters

mailbox	A TOKEN for the mailbox to which an object token is to be sent. This mailbox must be one that was set up to pass objects, not pointers.
object	A TOKEN containing an object token which is to be sent.
response	A TOKEN for a mailbox or semaphore at which the sending task will wait for a response. • If a valid selector, contains a token for the desired response mailbox or semaphore. • If SELECTOR$OF(NIL), indicates that no response is requested.

Output Parameter

except$ptr	A POINTER to a WORD to which the iRMX II Operating System will return the condition code generated by this system call.

Description

The SEND$MESSAGE system call sends messages to mailboxes that have been set up to pass object tokens. It sends a token for an iRMX II object to the specified mailbox. If there are tasks in the task queue at that mailbox, the task at the head of the queue is awakened and is given the token. Otherwise, the object token is placed at the tail of the object queue of the mailbox. The sending task has the option of specifying a mailbox or semaphore at which it will wait for a response from the task that receives the object. The nature of the response must be agreed upon by the writers of the two tasks.

When you create a mailbox with CREATE$MAILBOX, you can specify whether the mailbox will be used to pass object tokens or pointers. SEND$MESSAGE functions only with those mailboxes that have been set up to pass object tokens.

Example

```
/*************************************************************************
 *  This example illustrates how the SEND$MESSAGE system call can be     *
 *  used to send a segment token to a mailbox.                           *
 *************************************************************************/

DECLARE TOKEN      LITERALLY 'SELECTOR';

/*  NUCLUS.EXT declares all system calls */
$INCLUDE(/rmx286/inc/NUCLUS.EXT)

    DECLARE seg$token     TOKEN;
    DECLARE size          WORD;
    DECLARE mbx$token     TOKEN;
    DECLARE mbx$flags     WORD;
    DECLARE no$response   LITERALLY '0';
    DECLARE status        WORD;
    DECLARE job$token     TOKEN;

SAMPLEPROCEDURE:
    PROCEDURE;

    size = 64;                      /* designates new segment to contain 64
                                       bytes */
    mbx$flags = 0;                  /* designates four objects to be queued
                                       on the high performance object
                                       queue;  designates a first-in/
                                       first-out task queue */
    job$token = SELECTOR$OF(NIL);   /* indicates objects to be cataloged
                                       into the object directory of the
                                       calling task's job */
       •
       •     Typical PL/M-286 Statements
       •

/*************************************************************************
 *  The calling task creates a segment and a mailbox and catalogs the    *
 *  mailbox token.  The calling task then uses the tokens for both       *
 *  objects to send a message.                                           *
 *************************************************************************/

    seg$token = RQ$CREATE$SEGMENT  (size,
                                    @status);
    mbx$token = RQ$CREATE$MAILBOX  (mbx$flags,
                                    @status);
```

```
/*************************************************************************
 *  It is not mandatory for the calling task to catalog the mailbox     *
 *  token in order to send a message.  It is necessary, however, to     *
 *  catalog (or in someway communicate) the mailbox token if another    *
 *  task is to receive the message.                                      *
 *************************************************************************/

   CALL RQ$CATALOG$OBJECT            (job$token,
                                      mbx$token,
                                      @(3, 'MBX'),
                                      @status);
      •
      •     Typical PL/M-286 Statements
      •

/*************************************************************************
 *  The calling task invokes the SEND$MESSAGE system call to send the   *
 *  token for the segment to the specified mailbox.                      *
 *************************************************************************/

   CALL RQ$SEND$MESSAGE              (mbx$token,
                                      seg$token,
                                      no$response,
                                      @status);
      •
      •     Typical PL/M-286 Statements
      •

END SAMPLEPROCEDURE;
```

Condition Codes

E$OK	0000H	No exceptional conditions.
E$EXIST	0006H	One or more of the input parameters is not a token for an existing object.
E$MEM	0002H	The high performance queue is full and the calling task's job does not contain sufficient memory to complete the call.
ENOTCONFIGURED	0008H	This system call is not part of the present configuration.

E$TYPE	8002H	At least one of the following is true: • The mailbox parameter is a token for an object that is not a mailbox. • The response parameter is a token for an object that is neither a mailbox nor a semaphore. • The specified mailbox was set up to pass data, not tokens.

The CREATE$SEMAPHORE system call creates a semaphore.

```
semaphore = RQ$CREATE$SEMAPHORE (initial$value, max$value,
                                 semaphore$flags, except$ptr);
```

Input Parameters

initial$value — A WORD containing the initial number of units to be in the custody of the new semaphore.

max$value — A WORD containing the maximum number of units over which the new semaphore is to have custody at any given time. If max$value is zero, an E$PARAM error is returned.

semaphore$flags — A WORD containing information about the new semaphore. The low-order bit determines the queuing scheme for the new semaphore's task queue:

Value	Queuing Scheme
0	First-in/first-out
1	Priority based

The remaining bits in semaphore$flags are reserved for future use and should be set to zero.

Output Parameters

semaphore — A TOKEN to which the Operating System will return a token for the new semaphore.

except$ptr — A POINTER to a WORD to which the iRMX II Operating System will return the condition code generated by this system call.

Description

The CREATE$SEMAPHORE system call creates a semaphore and returns a token for it. The created semaphore counts as one against the object limit of the calling task's job.

Example

```
/*************************************************************************
 *  This example illustrates how the CREATE$SEMAPHORE system call can   *
 *  be used.                                                            *
 *************************************************************************/

DECLARE TOKEN                   LITERALLY 'SELECTOR';

/* NUCLUS.EXT declares all system calls */
$INCLUDE(/rmx286/inc/NUCLUS.EXT)

    DECLARE sem$token           TOKEN;
    DECLARE init$value          WORD;
    DECLARE max$value           WORD;
    DECLARE sem$flags           WORD;
    DECLARE status              WORD;

SAMPLEPROCEDURE:
    PROCEDURE;

    init$value = 1;             /* the new semaphore has one initial
                                     unit */
    max$value = 10H;            /* the new semaphore can have a maximum
                                   of 16 units */
    sem$flags = 0;              /* designates a first-in/
                                   first-out task queue */
        •
        •  Typical PL/M-286 Statements
        •

/*************************************************************************
 *  The token sem$token is returned when the calling task invokes the   *
 *  CREATE$SEMAPHORE system call.                                       *
 *************************************************************************/

    sem$token = RQ$CREATE$SEMAPHORE  (init$value,
                                      max$value,
                                      sem$flags,
                                      @status);
        •
        •   Typical PL/M-286 Statements
        •

END SAMPLEPROCEDURE;
```

Condition Codes

E$OK	0000H	No exceptional conditions.
E$LIMIT	0004H	The calling task's job has already reached its object limit.
E$MEM	0002H	The memory available to the calling task's job is not sufficient to create a semaphore.
ENOTCONFIGURED	0008H	This system call is not part of the present configuration.
E$PARAM	8004H	At least one of the following is true: • The initial$value parameter is larger than the max$value parameter. • The max$value parameter is 0.
E$SLOT	000CH	There isn't enough room in the GDT for another descriptor.

The DELETE$SEMAPHORE system call deletes a semaphore.

```
CALL RQ$DELETE$SEMAPHORE (semaphore, except$ptr);
```

Input Parameter

semaphore	A TOKEN for the semaphore to be deleted.

Output Parameter

except$ptr	A POINTER to a WORD to which the iRMX II Operating System will return the condition code generated by this system call.

Description

The DELETE$SEMAPHORE system call deletes the specified semaphore. If there are tasks in the semaphore's queue at the moment of deletion, they are awakened with an E$EXIST exceptional condition. The deleted semaphore counts as a credit of one toward the object total of the containing job.

Example

```
/*************************************************************************
 *  This example illustrates how the DELETE$SEMAPHORE system call   *
 * can be used.                                                          *
 *                                                                       *
*************************************************************************/

DECLARE TOKEN      LITERALLY 'SELECTOR';

/*  NUCLUS.EXT declares all system calls */
$INCLUDE(/rmx286/inc/NUCLUS.EXT)

    DECLARE sem$token   TOKEN;
    DECLARE init$value  WORD;
    DECLARE max$value   WORD;
    DECLARE sem$flags WORD;
    DECLARE status  WORD;
```

```
SAMPLEPROCEDURE:
    PROCEDURE;

    init$value = 1; /* the new semaphore has one initial unit */
    max$value = 10H;  /* the new semaphore can have a maximum
                         of 16 units */
    sem$flags = 0;  /* designates a first-in
                       first-out task queue */
        •
        •     Typical PL/M-286 Statements
        •

/*************************************************************************
 *  In order to delete a semaphore, a task must know the token for  *
 *  that semaphore.  In this example, the needed token is known     *
 *  because the calling task creates the semaphore.                 *
 *                                                                  *
*************************************************************************/

    sem$token = RQ$CREATE$SEMAPHORE  (init$value, max$value,
                                      sem$flags, @status);
        •
        •     Typical PL/M-286 Statements
        •

/*************************************************************************
 *  When the semaphore is no longer needed, it may be deleted by    *
 *  any task that knows the token for the semaphore.                *
 *                                                                  *
*************************************************************************/

    CALL RQ$DELETE$SEMAPHORE         (sem$token, @status);
        •
        •     Typical PL/M-286 Statements
        •

END SAMPLEPROCEDURE;
```

Condition Codes

E$OK	0000H	No exceptional conditions.
E$EXIST	0006H	One of the following is true: • The semaphore parameter is not a token for an existing object • The semaphore parameter represents a semaphore whose job is being deleted.

ENOTCONFIGURED	0008H	This system call is not part of the present configuration.
E$TYPE	8002H	The semaphore parameter is a token for an object that is not a semaphore.

The RECEIVE$UNITS system call delivers the calling task to a semaphore, where it waits for units.

```
value = RQ$RECEIVE$UNITS (semaphore, units, time$limit, except$ptr);
```

Input Parameters

semaphore	A TOKEN for the semaphore from which the calling task wants to receive units.
units	A WORD containing the number of units that the calling task is requesting.
time$limit	A WORD that indicates how long the calling task is willing to wait. • If zero, the WORD indicates that the calling task is not willing to wait. • If 0FFFFH, the WORD indicates that the task will wait as long as is necessary. • If between 0 and 0FFFFH, the WORD indicates the number of clock intervals that the task is willing to wait. The length of a clock interval is configurable. Refer to the *Extended iRMX II Interactive Configuration Utility Reference Manual* for further information.

Output Parameters

value	A WORD containing the number of units remaining in the semaphore after the calling task's request is satisfied.
except$ptr	A POINTER to a WORD to which the iRMX II Operating System will return the condition code generated by this system call.

RECEIVE$UNITS

Description

The RECEIVE$UNITS system call causes the calling task either to get the units that it is requesting or to wait for them in the semaphore's task queue. If the units are available and the task is at the front of the queue, the task receives the units and remains ready. Otherwise, the task is placed in the semaphore's task queue and goes to sleep, unless the task is not willing to wait. In the latter case, or if the task's waiting period elapses before the requested units are available, the task is awakened with an E$TIME exceptional condition.

Example

```
/*************************************************************************
 *  This example illustrates how the RECEIVE$UNITS system call can be   *
 *  used to receive a unit.                                              *
 *************************************************************************/

DECLARE TOKEN      LITERALLY 'SELECTOR';

/*  NUCLUS.EXT declares all system calls */
$INCLUDE(/rmx286/inc/NUCLUS.EXT)
    DECLARE sem$token             TOKEN;
    DECLARE calling$tasks$job     TOKEN;
    DECLARE wait$forever          LITERALLY '0FFFFH';
    DECLARE seg$token             TOKEN;
    DECLARE units$remaining       WORD;
    DECLARE units$requested       WORD;
    DECLARE status                WORD;
SAMPLEPROCEDURE:
    PROCEDURE;

        •
        •     Typical PL/M-286 Statements
        •

/*************************************************************************
 *  In this example, the calling task looks up the token for the        *
 *  semaphore prior to invoking the RECEIVE$UNITS system call.          *
 *************************************************************************/
    calling$tasks$job = SELECTOR$OF(NIL);
    sem$token = RQ$LOOKUP$OBJECT      (calling$tasks$job,
                                       @(5,'SEMA4'),
                                       wait$forever,
                                       @status);

        •
        •     Typical PL/M-286 Statements
        •
```

```
/*************************************************************************
 *  Knowing the token for the semaphore, the calling task can wait for  *
 *  units at this semaphore by invoking the RECEIVE$UNITS system call.  *
 *************************************************************************/

   units$requested = 4;

   units$remaining = RQ$RECEIVE$UNITS (sem$token,
                                       units$requested,
                                       wait$forever,
                                       @status);
       •
       •     Typical PL/M-286 Statements
       •

END SAMPLEPROCEDURE;
```

Condition Codes

E$OK	0000H	No exceptional conditions.
E$EXIST	0006H	The semaphore parameter is not a token for an existing object.
E$LIMIT	0004H	The units parameter is greater than the maximum value specified for the semaphore when it was created.
ENOTCONFIGURED	0008H	This system call is not part of the present configuration.
E$TIME	0001H	One of the following is true: • The calling task was not willing to wait and the requested units were not available. • The task waited in the task queue and its designated waiting period elapsed before the requested units were available.
E$TYPE	8002H	The semaphore parameter is a token for an object that is not a semaphore.

The SEND$UNITS system call sends units to a semaphore.

```
CALL RQ$SEND$UNITS (semaphore, units, except$ptr);
```

Input Parameters

semaphore	A TOKEN for the semaphore to which the units are to be sent.
units	A WORD containing the number of units to be sent.

Output Parameter

except$ptr	A POINTER to a WORD to which the iRMX II Operating System will return the condition code generated by this system call.

Description

The SEND$UNITS system call sends the specified number of units to the specified semaphore. If the transmission would cause the semaphore to exceed its maximum allowable supply, then an E$LIMIT exceptional condition occurs. Otherwise, the transmission is successful and the Nucleus attempts to satisfy the requests of the tasks in the semaphore's task queue, beginning at the head of the queue.

Example

```
/*************************************************************************
 *  This example illustrates how the SEND$UNITS system call can be used *
 *  to send units to a semaphore.                                        *
 *************************************************************************/

DECLARE TOKEN      LITERALLY 'SELECTOR';

/*  NUCLUS.EXT declares all system calls */
$INCLUDE(/rmx286/inc/NUCLUS.EXT)

    DECLARE sem$token            TOKEN;
    DECLARE init$value           WORD;
    DECLARE max$value            WORD;
    DECLARE sem$flags            WORD;
    DECLARE three$units$sent     LITERALLY '3';
    DECLARE status               WORD;
    DECLARE job$token            TOKEN;
```

```
SAMPLEPROCEDURE:
    PROCEDURE;
    init$value = 1;                 /* the new semaphore has one initial
                                       unit */
    max$value = 10H;                /* the new semaphore can have a maximum
                                       of 16 units */
    sem$flags = 0;                  /* designates a first-in/first-out
                                       task queue */
    job$token = SELECTOR$OF(NIL); /* indicates objects to be cataloged
                                       into the object directory of the
                                       calling task's job */
        •
        •     Typical PL/M-286 Statements
        •
/*************************************************************************
 *  The calling task creates a semaphore and catalogs the semaphore     *
 *  token.  The calling task then uses the token to send a unit.        *
 *************************************************************************/

    sem$token = RQ$CREATE$SEMAPHORE  (init$value,
                                      max$value,
                                      sem$flags,
                                      @status);
        •
        •     Typical PL/M-286 Statements
        •

 /************************************************************************
 *  It is not mandatory to catalog the semaphore token in order to send *
 *  units.  It is necessary, however, to catalog (or in someway         *
 *  communicate) the semaphore token if another task is to receive the  *
 *  units.                                                              *
 ************************************************************************/

    CALL RQ$CATALOG$OBJECT          (job$token,
                                     sem$token,
                                     @(5, 'SEMA4'),
                                     @status);
        •
        •     Typical PL/M-286 Statements
        •

/*************************************************************************
 *  The calling task invokes the SEND$UNITS system call to send the     *
 *  units to the semaphore just created (sem$token.)                    *
*************************************************************************/

    CALL RQ$SEND$UNITS              (sem$token,
                                     three$units$sent,
                                     @status);
```

```
•
•     Typical PL/M-286 Statements
•

END SAMPLEPROCEDURE;
```

Condition Codes

E$OK	0000H	No exceptional conditions.
E$EXIST	0006H	The semaphore parameter is not a token for an existing object.
E$LIMIT	0004H	The number of units that the calling task is trying to send would cause the semaphore's supply of units to exceed its maximum allowable supply.
ENOTCONFIGURED	0008H	This system call is not part of the present configuration.
E$TYPE	8002H	The semaphore parameter is a token for an object that is not a semaphore.

The CREATE$SEGMENT system call creates a segment.

```
segment = RQ$CREATE$SEGMENT (size, except$ptr);
```

Input Parameter

size — A WORD that specifies the size of the requested segment.

- If not zero, it contains the size, in bytes, of the requested segment.
- If zero Or 0FFFFH, It indicates that the size of the request is 65536 (64K) bytes.

Output Parameters

segment — A TOKEN to which the Operating System will return a token for the new segment.

except$ptr — A POINTER to a WORD to which the iRMX II Operating System will return the condition code generated by this system call.

Description

The CREATE$SEGMENT system call creates a segment and returns the token for it. The memory for the segment is taken from the free portion of the memory pool of the calling task's job, unless borrowing from the parent job is both necessary and possible. The new segment counts as one against the object limit of the calling task's job.

To gain access into the segment, you should base an array or structure on the SELECTOR that is returned as the token for the segment.

When setting up the descriptor for the new segment, the Nucleus assigns the segment as a data segment, with read/write access, at privilege level 0.

Example

```
MAINPROC: DO;
/*************************************************************************
 *  This example illustrates how the CREATE$SEGMENT system call can be  *
 *  used.                                                                *
 *************************************************************************/

DECLARE TOKEN                          LITERALLY 'SELECTOR';

/* NUCLUS.EXT declares all system calls */
$INCLUDE(/rmx286/inc/NUCLUS.EXT)

    DECLARE seg$token                  TOKEN;
    DECLARE seg$size                   WORD;
    DECLARE status                     WORD;

SAMPLEPROCEDURE:
    PROCEDURE;

    seg$size = 100H;                  /* the size of the requested segment
                                          is 256 bytes */
            Typical PL/M-286 Statements

/*************************************************************************
 *  The token seg$token is returned when the calling task invokes the   *
 *  CREATE$SEGMENT system call.                                          *
 *************************************************************************/

    seg$token = RQ$CREATE$SEGMENT (seg$size, @status);

     Typical PL/M-286 Statements

END SAMPLEPROCEDURE;
END MAINPROC;
```

Condition Codes

E$OK	0000H	No exceptional conditions.
E$LIMIT	0004H	The calling task's job has already reached its object limit.
E$MEM	0002H	The memory available to the calling task's job is not sufficient to create a segment of the specified size.
ENOTCONFIGURED	0008H	This system call is not part of the present configuration.
E$SLOT	000CH	There isn't enough room in the GDT for another descriptor.

The DELETE$SEGMENT system call deletes a segment or a descriptor.

```
CALL RQ$DELETE$SEGMENT (segment, except$ptr);
```

Input Parameter

segment — A TOKEN for the segment or descriptor to be deleted.

Output Parameter

except$ptr — A POINTER to a WORD to which the iRMX II Operating System will return the condition code generated by this system call.

Description

The DELETE$SEGMENT system call deletes iRMX II segments created via CREATE$SEGMENT and descriptors created via RQE$CREATE$DESCRIPTOR. When deleting iRMX II segments, this system call returns the specified segment to the memory pool from which it was allocated. The deleted segment counts as a credit of one toward the object total of the containing job.

When deleting descriptors, this system call does not return any memory to the memory pool. It simply clears the descriptor slot in the Global Descriptor Table (GDT) and returns that slot to the memory manager for reassignment.

Example

```
/*****************************************************************
 *  This example illustrates how the DELETE$SEGMENT system call     *
 *  can be used.                                                    *
 *****************************************************************/

DECLARE TOKEN      LITERALLY 'SELECTOR';

/*  NUCLUS.EXT declares all system calls */
$INCLUDE(/rmx286/inc/NUCLUS.EXT)

    DECLARE seg$token TOKEN;
    DECLARE size      WORD;
    DECLARE status    WORD;
```

```
SAMPLEPROCEDURE:
    PROCEDURE;
    size = 64;          /* designates new segment to contain
                           64 bytes */
        •
        •     Typical PL/M-286 Statements
        •

/*************************************************************************
 *  In order to delete a segment, a task must know the token for    *
 *  that segment.  In this example, the needed token is known       *
 *  because the    calling task creates the segment.                *

*************************************************************************/

    seg$token = RQ$CREATE$SEGMENT  (size, @status);

        •
        •     Typical PL/M-286 Statements
        •

/*************************************************************************
 *  When the segment is no longer needed, it may be deleted by any  *
 *  task that knows the token for the segment.                      *
 *                                                                  *
*************************************************************************/

    CALL RQ$DELETE$SEGMENT         (seg$token, @status);

        •
        •     Typical PL/M-286 Statements
        •

END SAMPLEPROCEDURE;
```

Condition Codes

E$OK	0000H	No exceptional conditions.
E$EXIST	0006H	At least one of the following is true: • The segment parameter is not a token for an existing object. • The segment parameter represents a segment or descriptor whose job is being deleted.

ENOTCONFIGURED	0008H	This system call is not part of the present configuration.
E$TYPE	8002H	The segment parameter is a token for an object that is not a segment or a descriptor.

The GET$POOL$ATTRIB system call returns information about the memory pool of the calling task's job. For compatibility with iRMX I systems, this system call can report pool sizes no larger than 1M byte.

```
CALL RQ$GET$POOL$ATTRIB (attrib$ptr, except$ptr);
```

Output Parameters

attrib$ptr — A POINTER to a data structure of the following form:

```
STRUCTURE(
          POOL$MAX       WORD,
          POOL$MIN       WORD,
          INITIAL$SIZE   WORD,
          ALLOCATED      WORD,
          AVAILABLE      WORD);
```

The system call fills in the fields of this structure so that after the call:

- POOL$MAX contains the maximum allowable size (in 16-byte paragraphs) of the memory pool of the calling task's job.
- POOL$MIN contains the minimum allowable size (in 16-byte paragraphs) of the memory pool of the calling task's job.
- INITIAL$SIZE contains the original value of the pool$min attribute.
- ALLOCATED contains the number of 16-byte paragraphs currently allocated from the memory pool of the calling task's job.
- AVAILABLE contains the number of 16-byte paragraphs currently available in the memory pool of the calling task's job. It does not include memory that could be borrowed from the parent job. The memory indicated in AVAILABLE may be fragmented and thus not allocatable as a single segment.

except$ptr — A POINTER to a WORD to which the Operating System will return the condition code generated by this system call.

Description

The GET$POOL$ATTRIB system call returns information regarding the memory pool of the calling task's job. The data returned comprises the allocated and available portions of the pool, as well as its initial, minimum, and maximum sizes.

This system call is available for compatibility with iRMX I systems. Because the elements of the attrib$ptr structure are all WORD values, this system call cannot return accurate size information about memory pools that are larger than 1M byte. If the memory pool is larger than 1M byte, this system call reports the size as 1M byte. To get accurate information concerning large (over 1M byte) memory pools, use the RQEGETPOOL$ATTRIB system call.

Example

```
/*************************************************************************
 *  This example illustrates how the GET$POOL$ATTRIB system call can     *
 *  be used to return information about the memory pool of the           *
 *  calling task's job.                                                  *
 *************************************************************************/

DECLARE TOKEN       LITERALLY 'SELECTOR';

/*  NUCLUS.EXT declares all system calls */
$INCLUDE(/rmx286/inc/NUCLUS.EXT)

    DECLARE mem$pool  STRUCTURE (
                                 mem$pool$max      WORD,
                                 mem$pool$min      WORD,
                                 mem$initial$size  WORD,
                                 mem$allocated     WORD,
                                 mem$available     WORD);
    DECLARE status    WORD;

SAMPLEPROCEDURE:
    PROCEDURE;
        •
        •     Typical PL/M-286 Statements
        •

/*************************************************************************
 *  The maximum and minimum size of the memory pool, the original value  *
 *  of the minimum pool size, and the allocated and available number of  *
 *  16-byte paragraphs in the memory pool of the calling task's job are  *
 *  all returned when the calling task invokes the GET$POOL$ATTRIB       *
 *  system call.                                                         *
 *************************************************************************/
```

```
    CALL RQ$GET$POOL$ATTRIB  (@mem$pool,
                              @status);
        •
        •     Typical PL/M-286 Statements
        •

END SAMPLEPROCEDURE;
```

Condition Codes

E$OK	0000H	No exceptional conditions.
EBADADDR	800FH	The attrib$ptr pointer is invalid. Either the selector does not refer to a valid segment, or the offset is outside the segment boundaries.
ENOTCONFIGURED	0008H	This system call is not part of the present configuration.

The RQEGETPOOL$ATTRIB system call returns information about any job's memory pool. It is similar to the GET$POOL$ATTRIB system call, except that it can report pool sizes larger than 1M byte, and it returns information about the parent job and the amount of memory borrowed.

```
CALL RQE$GET$POOL$ATTRIB (attrib$ptr, except$ptr);
```

Input/Output Parameter

attrib$ptr — A POINTER to a data structure of the following form:

```
STRUCTURE(
          TARGET$JOB     TOKEN,
          PARENT$JOB     TOKEN,
          POOL$MAX   DWORD,
          POOL$MIN   DWORD,
          INITIAL$SIZE   DWORD,
          ALLOCATED DWORD,
          AVAILABLE DWORD,
          BORROWED   DWORD);
```

This structure holds both input and output fields. You fill in the TARGET$JOB field to identify the job whose memory-pool information you want. The system call fills in the remaining fields to provide that information. The fields of this structure contain the following information:

- TARGET$JOB is a field that you fill in to specify the token for the job whose memory pool you want to examine. A value of SELECTOR$OF(NIL) indicates the calling task's job.
- PARENT$JOB is a field in which the system call returns a token for the parent job of the target job you specified.
- POOL$MAX is a DWORD that the system call fills in to specify the maximum allowable size (in 16-byte paragraphs) of the target job's memory pool.
- POOL$MIN is a DWORD that the system call fills in to specify the minimum allowable size (in 16-byte paragraphs) of the target job's memory pool.
- INITIAL$SIZE is a DWORD that the system call fills in to specify the original value of the pool$min attribute.

- ALLOCATED is a DWORD that the system call fills in to specify the number of 16-byte paragraphs currently allocated from the target job's memory pool.
- AVAILABLE is a DWORD that the system call fills in to specify the number of 16-byte paragraphs currently available in the target job's memory pool. It does not include memory that could be borrowed from the parent job. The memory indicated in AVAILABLE might be fragmented and thus not allocatable as a single segment.
- BORROWED is a DWORD that the system call fills in to specify the amount of memory (in 16-byte paragraphs) that the target job has borrowed.

except$ptr — A POINTER to a WORD to which the Operating System will return the condition code generated by this system call.

Description

The RQEGETPOOL$ATTRIB system call returns information regarding the memory pool of any job you specify. The data returned comprises the allocated and available portions of the pool; the initial, minimum, and maximum pool sizes; the amount of memory that the job has borrowed; and the identity of the job's parent job.

This system call is similar to the GET$POOL$ATTRIB system call, but it offers several enhancements. Unlike that system call, RQEGETPOOL$ATTRIB can return information about memory pools that are larger than 1M byte. It is not restricted to returning information about the calling task's job; it can return information about any job. And, it returns the amount of memory the job borrowed along with a token for the job's parent job.

Example

```
/*************************************************************************
 *  This example illustrates how the RQE$GET$POOL$ATTRIB system call     *
 *  can be used to return information about the memory pool of the       *
 *  calling task's job.                                                  *
 *************************************************************************/

DECLARE TOKEN       LITERALLY 'SELECTOR';

/*  NUCLUS.EXT declares all system calls */
$INCLUDE(/rmx286/inc/NUCLUS.EXT)
```

```
    DECLARE mem$pool  STRUCTURE (
                                 targ$job          TOKEN,
                                 parent$job        TOKEN,
                                 mem$pool$max      DWORD,
                                 mem$pool$min      DWORD,
                                 mem$initial$size  DWORD,
                                 mem$allocated     DWORD,
                                 mem$available     DWORD,
                                 mem$borrowed      DWORD);
    DECLARE status    WORD;

SAMPLEPROCEDURE:
    PROCEDURE;

    mem$pool.targ$job = SELECTOR$OF(NIL);  /* Set the calling task's job
                                              as the calling job. */

        •
        •     Typical PL/M-286 Statements
        •

/************************************************************************
 *  The parent job's token, the maximum and minimum size of the memory  *
 *  pool, the original value of mem$pool$min, and the amount            *
 *  of allocated, available, and borrowed memory in the memory pool of  *
 *  the calling task's job are all returned when the task invokes the   *
 *  RQE$GET$POOL$ATTRIB system call.                                    *
 ************************************************************************/

    CALL RQ$RQE$GET$POOL$ATTRIB  (@mem$pool,
                                  @status);
        •
        •     Typical PL/M-286 Statements
        •

END SAMPLEPROCEDURE;
```

Condition Codes

E$OK	0000H	No exceptional conditions.
EBADADDR	800FH	The attrib$ptr pointer is invalid. Either the selector does not refer to a valid segment, or the offset is outside the segment boundaries.

E$EXIST	0006H	The token for the target job is not a valid iRMX token.
ENOTCONFIGURED	0008H	This system call is not part of the present configuration.
E$TYPE	8002H	The token for the target job is not a job token.

The GET$SIZE system call returns the size, in bytes, of a segment.

```
size = RQ$GET$SIZE (segment, except$ptr);
```

Input Parameter

segment — A TOKEN for a segment whose size is desired.

Output Parameters

size — A WORD in which the system call returns the size of the segment, as follows.

- If not zero, it contains the size, in bytes, of the segment indicated by the segment parameter.
- If zero, the WORD indicates that the size of the segment is 65536 (64K) bytes.

except$ptr — A POINTER to a WORD to which the iRMX II Operating System will return the condition code generated by this system call.

Description

The GET$SIZE system call returns the size, in bytes, of an iRMX II segment.

Example

```
/*************************************************************************
 *  This example illustrates how the GET$SIZE system call can be used.  *
 *************************************************************************/

DECLARE TOKEN      LITERALLY 'SELECTOR';

/*  NUCLUS.EXT declares all system calls */
$INCLUDE(/rmx286/inc/NUCLUS.EXT)
    DECLARE mbx$token             TOKEN;
    DECLARE calling$task$job      TOKEN;
    DECLARE wait$forever          LITERALLY '0FFFFH';
```

```
    DECLARE seg$token TOKEN;
    DECLARE response  TOKEN;
    DECLARE size      WORD;
    DECLARE status    WORD;

SAMPLEPROCEDURE:
    PROCEDURE;

        •
        •     Typical PL/M-286 Statements
        •

/*************************************************************************
 *  In order to invoke the GET$SIZE system call, the calling task must  *
 *  know the token for the segment.  In this example, the calling task  *
 *  invokes the LOOKUP$OBJECT and RECEIVE$MESSAGE system calls to       *
 *  receive the token for a segment (seg$token).  The calling task      *
 *  invoked LOOKUP$OBJECT to receive the token for the mailbox named    *
 *  'MBX'. 'MBX' had been designated as the mailbox another task        *
 *  would use to send an object.                                        *
 *************************************************************************/

    calling$task$job = SELECTOR$OF(NIL);

    mbx$token = RQ$LOOKUP$OBJECT       (calling$task$job,
                                        @(3,'MBX'),
                                        wait$forever,
                                        @status);

        •
        •     Typical PL/M-286 Statements
        •

/*************************************************************************
 *  The RECEIVE$MESSAGE system call returns seg$token to the calling    *
 *  task.                                                               *
 *************************************************************************/

    seg$token = RQ$RECEIVE$MESSAGE     (mbx$token,
                                        wait$forever,
                                        @response,
                                        @status);

        •
        •     Typical PL/M-286 Statements
        •
/*************************************************************************
 *  The GET$SIZE system call returns the size of the segment pointed    *
 *  to by seg$token.                                                    *
 *************************************************************************/
```

```
    size = RQ$GET$SIZE                (seg$token, @status);

        •
        •     Typical PL/M-286 Statements
        •

END SAMPLEPROCEDURE;
```

Condition Codes

E$OK	0000H	No exceptional conditions.
E$EXIST	0006H	The segment parameter is not a token for an existing object.
ENOTCONFIGURED	0008H	This system call is not part of the present configuration.
E$TYPE	8002H	The segment parameter is a token for an object that is not a segment.

The SET$POOL$MIN system call sets a job's pool$min attribute.

```
CALL RQ$SET$POOL$MIN (new$min, except$ptr);
```

Input Parameter

new$min — A WORD indicating the pool$min attribute of the calling task's job.

- If 0FFFFH, indicates that the pool$min attribute of the calling task's job is to be set equal to that job's pool$max attribute.
- If less than 0FFFFH, contains the new value of the pool$min attribute of the calling task's job. This new value must not exceed that job's pool$max attribute.

Output Parameter

except$ptr — A POINTER to a WORD to which the iRMX II Operating System will return the condition code generated by this system call.

Description

The SET$POOL$MIN system call sets the pool$min attribute of the calling task's job. The new value must not exceed that job's pool$max attribute. When the pool$min attribute is made larger than the current pool size, the pool is not enlarged until the additional memory is needed.

Example

```
/*************************************************************************
 *  This example illustrates how the SET$POOL$MIN system call can be     *
 *  used.                                                                *
 *************************************************************************/

DECLARE TOKEN           LITERALLY 'SELECTOR';

/*  NUCLUS.EXT declares all system calls */
$INCLUDE(/rmx286/inc/NUCLUS.EXT)

    DECLARE new$min     WORD;
    DECLARE status      WORD;
```

```
SAMPLEPROCEDURE:
    PROCEDURE;

    new$min = 0FFFFH;          /* sets pool$min attribute of calling
                                  task's job equal to job's pool$max
                                  attribute */
       •
       •     Typical PL/M-286 Statements
       •

/*************************************************************************
 *  In this example the pool$min attribute of the calling task's job    *
 *  is to be set equal to that job's pool$max attribute.                *
 *************************************************************************/

    CALL RQ$SET$POOL$MIN           (new$min,
                                    @status);
       •
       •     Typical PL/M-286 Statements
       •

END SAMPLEPROCEDURE;
```

Condition Codes

E$OK	0000H	No exceptional conditions.
E$LIMIT	0004H	The new$min parameter is not 0FFFFH, but it is greater than the pool$max attribute of the calling task's job.
ENOTCONFIGURED	0008H	This system call is not part of the present configuration.

The RQ$CREATE$BUFFER$POOL system call establishes a buffer pool and returns a token for it.

```
buffer$pool = RQ$CREATE$BUFFER$POOL (maximum$buffs, flags, except$ptr);
```

Input Parameters

maximum$buffs — A WORD that indicates the maximum number of buffers that can exist in the buffer pool at one time. The maximum size of the buffer pool is controlled by this parameter.

flags — A WORD that defines the attributes of the buffer pool as follows:

Bit	Meaning
0	Reserved, should be set to zero.
1	Indicates if data chaining is supported. If set (1), then data chains are supported. If not set (0), then only contiguous buffers will be used.
2-15	Reserved, should be set to zero.

Output Parameters

buffer$pool — A TOKEN in which the system call returns a token for the newly-created buffer pool.

except$ptr — A POINTER to a WORD to which the Operating System will return the condition code generated for this system call.

Description

This system call sets up a buffer pool that will be associated with one or more ports. These buffer pools can be used without any ports associated with them. In such cases, they are general-purpose buffer managers. Once a buffer pool has been set up, tasks can request segments of memory from the buffer pool (via RQ$REQUEST$BUFFER) instead of creating the segments directly (via CREATE$SEGMENT) each time memory space is needed. When a task finishes with a buffer, it can release the buffer back to the buffer pool (via RQ$RELEASE$BUFFER) for later use by other tasks.

When a buffer pool is created, it contains no memory segments. Therefore, you must use RQ$CREATE$SEGMENT to create the segments to be managed by the buffer pool. Once you create the segments, you can use RQ$RELEASE$BUFFER to add the segments to the buffer pool. Each buffer pool can manage as many as 8192 (8K) segments which can be of eight different sizes.

Condition Codes

E$OK	0000H	No exceptional conditions.
E$MEM	0002H	There isn't enough memory to create the requested buffer pool.
ENOTCONFIGURED	0008H	This system call is not part of the present configuration.
E$PARAM	8004H	The max$bufs parameter has a value greater that 8192.
E$SLOT	000CH	There is no room in the GDT for the buffer pool's descriptor.

The RQ$DELETE$BUFFER$POOL system call deletes a buffer pool.

```
CALL RQ$DELETE$BUFFER$POOL (buffer$pool, except$ptr);
```

Input Parameter

buffer$pool	A TOKEN for the buffer pool to be deleted. This buffer pool must have been created with the RQ$CREATE$BUFFER$POOL system call.

Output Parameter

except$ptr	A POINTER to a WORD to which the Operating System will return the condition code generated for this system call.

Description

This system call deletes a buffer pool originally created with RQ$CREATE$BUFFER$POOL. All buffers in the buffer pool and any information in them is also deleted.

A buffer pool cannot be deleted as long as a port is attached to it.

Condition Codes

E$OK	0000H	No exceptional conditions.
E$EXIST	0006H	The buffer$pool parameter is not a token for an existing object.
ENOTCONFIGURED	0008H	This system call is not part of the present configuration.
E$TYPE	8002H	The buffer$pool parameter is the token for an object that is not a buffer pool.

The RQ$RELEASE$BUFFER system call returns previously allocated buffer space to the specified buffer pool.

```
CALL RQ$RELEASE$BUFFER( buffer$pool, buffer$tkn, flags, except$ptr);
```

Input Parameters

buffer$pool — A TOKEN for the buffer pool that is to receive the released buffer.

buffer$tkn — A TOKEN to the buffer that is to be released.

flags — A WORD that is defined as follows:

Bits	Meaning
0	If 0, then the buffer$tkn parameter refers to a contiguous buffer. If 1, then the buffer$tkn parameter refers to a data chain.
1-15	Reserved should be set to zero.

Output Parameter

except$ptr — A POINTER to a WORD to which the Operating System will return the condition code generated for this system call.

Description

The RQ$RELEASE$BUFFER system call returns a buffer to a specified buffer pool. If the buffer pool is full, you will get an E$LIMIT exception and the buffer is still valid.

Condition Codes

E$OK	0000H	No exceptional conditions.
E$EXIST	0006H	Either the buffer$tkn or buffer$pool parameter, or both, does not refer to an existing object.
E$LIMIT	0004H	The calling task's job has already reached its' object limit.
ENOTCONFIGURED	0008H	This system call is not part of the present configuration.
E$TYPE	8002H	Either the buffer$pool parameter does not refer to a buffer$pool; or, the buffer$tkn parameter does not refer to a segment.

The RQ$REQUEST$BUFFER system call is used to get a buffer from an existing buffer pool.

```
buffer$token = RQ$REQUEST$BUFFER(buffer$pool, size, except$ptr);
```

Input Parameters

buffer$pool	A TOKEN for an existing buffer pool.
size	A DWORD specifying the desired size of the requested buffer. This value must be in the range of 1H through 0FFFFFEH.

Output Parameters

buffer$token	A TOKEN to a buffer that fills the request. This buffer is either a single segment, or a data chain block.
except$ptr	A POINTER to a WORD to which the Operating System will return the condition code generated for this system call.

Description

The RQ$REQUEST$BUFFER system call gets a buffer from an existing buffer pool. Ideally the data fits into an existing buffer. If a buffer large enough to hold the data is not available and the buffer pool supports data chains, a data chain will be created. A data chain is a series of buffers that contain parts of the entire message. The location of each individual block is contained in the data chain block. When creating data chains, the largest available buffer will be used for the first portion of the data chain, then the next buffer and so on. These available buffers may be larger than the data structures and data actually stored in them. Therefore, a data chain may use more physical space than the data would actually require.

The minimum buffer size for data chains is 1K in length and at least one buffer must be requested to permit the system to build the data chain block. The minimum data chain block size can be computed as:

(max_elements*8)+2 BYTES

where max_elements is an ICU parameter with a default value of 79H (127 decimal).

Condition Codes

E$OK	0000H	No exceptional conditions.
E$DATA$CHAIN	000DH	A data chain has been returned. The TOKEN points to the beginning of the data chain block.
E$EXIST	0006H	The buffer$pool parameter does not refer to an existing object.
E$MEM	0002H	The system could not locate enough memory to create the requested buffer from the buffer pool, as a data chain, or from free space.
ENOTCONFIGURED	0008H	This system call is not part of the present configuration.
E$PARAM	8004H	The size parameter is equal to zero, or is larger than 0FFFFFFH.
E$SLOT	000CH	The Global Descriptor table is full.
E$TYPE	8002H	The buffer$pool parameter refers to an object that is not a buffer pool.

The RQE$CHANGE$DESCRIPTOR system call changes the physical address or size of a descriptor that was established with the RQE$CREATE$DESCRIPTOR system call.

CAUTION

This system call can change a descriptor's address to refer to any area of physical memory, even if other descriptors already refer to that memory. Although you might want to have multiple descriptors refer to the same area of memory for aliasing purposes, take care not to overlap memory accidentally.

```
CALL RQE$CHANGE$DESCRIPTOR (descriptor, abs$addr, size, except$ptr);
```

Input Parameters

descriptor	A TOKEN for the descriptor to be changed.
abs$addr	A DWORD containing a full, 24-bit address. This is the address where you want the segment represented by this descriptor to start. If you supply a 0 for this parameter, the segment retains its current starting address.
size	A WORD indicating the size of the segment in bytes. If you supply a 0 for this parameter, the segment size is set to 64K bytes.

Output Parameters

except$ptr	A POINTER to a WORD to which the iRMX II Operating System will return the condition code generated by this system call.

Description

The RQE$CHANGE$DESCRIPTOR system call allows you to adjust certain entries in the Global Descriptor Table (GDT). You can change the base physical address and size of descriptors that were created with the RQE$CREATE$DESCRIPTOR system call.These descriptors represent 80286 segments of memory. You cannot change descriptors that represent other kinds of iRMX II objects (such as segments, tasks, or mailboxes), nor can you adjust descriptors for other 80286 constructs (such as call or task gates).

RQE$CHANGE$DESCRIPTOR

This system call is intended for system programs that need to access areas of memory in special ways. For example, an overlay loader could use this system call to transfer different-sized code blocks to memory. Other system programs can use this system call to alias reserved or system segments, giving them the ability to modify segments that are normally read-only or code segments. With RQE$CHANGE$DESCRIPTOR, a system program can minimize the number of descriptor slots it uses. Because the address and size of a descriptor is adjustable, one descriptor can access many different areas of memory.

This system call can change the address and size of a segment descriptor so that it refers to any area of memory. Therefore, when used improperly, it can corrupt system and user data and allow overwriting of program code. Use it with care.

Example

```
/*****************************************************************
*  This example illustrates the use of RQE$CHANGE$DESCRIPTOR by   *
*  creating a descriptor for a previously undefined area of       *
*  memory and then changing it.                                   *
*****************************************************************/

/* NUCLUS.EXT declares all system calls */
$INCLUDE(/rmx286/inc/NUCLUS.EXT)
DECLARE TOKEN   LITERALLY 'SELECTOR';
DECLARE desc$token  TOKEN;
DECLARE abs$addr    DWORD;
DECLARE size    WORD;
DECLARE status  WORD;
SAMPLEPROCEDURE:
  PROCEDURE;
  abs$addr = 200000H;   /* The absolute address of the
       memory area being given an
       address is 2M bytes. */
  size = 256;   /* The size of the block is
       256 bytes. */
/*****************************************************************
* The token desc$token is returned when the calling task invokes  *
* the CREATE$DESCRIPTOR system call.                              *
*****************************************************************/

  desc$token = RQE$CREATE$DESCRIPTOR (abs$addr, size, @status);

    •
    •     Typical PL/M-286 Statements
    •
```

```
/*****************************************************************
* The descriptor is changed to access a new area of memory whose  *
* base address is specified by abs$addr.                          *
*****************************************************************/

  abs$addr = A00000H;   /* The absolute address of the memory
       area is changed to 10M bytes. */
  size = 0512;  /* The size of the requested descriptor
       is 512 bytes. */
  CALL  RQE$CHANGE$DESCRIPTOR  (desc$token, abs$addr, size, @status);

•
•     Typical PL/M-286 Statements
•

END SAMPLEPROCEDURE;
```

Condition Codes

E$OK	0000H	No exceptional conditions.
E$EXIST	0006H	The descriptor parameter is not a token for an existing object.
ENOTCONFIGURED	0008H	This system call is not part of the present configuration.
E$PARAM	8004H	The absolute address is larger than 16M bytes.
E$TYPE	8002H	The descriptor parameter is a token for an object that is not an iRMX II descriptor.

The RQE$CREATE$DESCRIPTOR system call builds a descriptor for an 80286 memory segment, places the descriptor in the 80286 Global Descriptor Table (GDT), and returns a token for the descriptor.

CAUTION

This system call can set up a segment descriptor to refer to any area of physical memory, even if other descriptors already refer to that memory. Although you might want to have multiple descriptors refer to the same area of memory for aliasing purposes, take care not to overlap memory accidentally.

```
descriptor = RQE$CREATE$DESCRIPTOR (abs$addr, size, except$ptr);
```

Input Parameters

abs$addr	A DWORD containing a full, 24-bit physical address. This is the address where you want the segment represented by this descriptor to start.
size	A WORD indicating the size of the segment in bytes. If you supply a 0 for this parameter, the segment size is set to 64K bytes.

Output Parameters

descriptor	A TOKEN to which the Operating System will return a token for the new descriptor.
except$ptr	A POINTER to a WORD to which the iRMX II Operating System will return the condition code generated by this system call.

Description

Before the 80286 processor can access an area of memory (in protected mode), a descriptor for the memory segment must exist in one of the descriptor tables (the Global Descriptor Table or the Local Descriptor Table). For iRMX II objects (jobs, tasks, segments, mailboxes, etc.), the Operating System automatically creates the necessary descriptors when it creates the objects. The RQE$CREATE$DESCRIPTOR system call gives you the additional capability of adding your own memory-segment descriptors to the GDT.

When you set up a descriptor, you can specify the base physical address and size of the memory segment. The segment can lie anywhere in available memory, even outside the range managed by the Operating System. The memory can overlap that contained in other segments, if desired. The Operating System automatically sets up the new segment as a data segment with read/write access at privilege level 0.

This system call is intended for system programs that need to access areas of memory in special ways. For example, an overlay loader could use this system call to set up a data segment so that it could load a program into what would normally be a code segment. Other system programs can use this system call to alias reserved segments, giving them the ability to modify read-only segments or segments outside the range managed by the Operating System (and thus not accessible via CREATE$SEGMENT). Device drivers can use this system call to gain access to dual-port memory resident on controller boards.

A segment created with this system call can be deleted by calling either the RQE$DELETE$DESCRIPTOR or DELETE$SEGMENT system call. However, segments created with RQE$CREATE$DESCRIPTOR are marked as descriptors, not iRMX II segments. Unlike ordinary iRMX II segments (set up with CREATE$SEGMENT), the memory associated with these segments does not return to the iRMX II memory pool for reallocation when the segments are deleted.

This system call can set up a segment descriptor to refer to any area of memory. Therefore, when used improperly, it can corrupt system and user data and allow overwriting of program code. Use it with care.

Example

```
/*********************************************************************
* This example illustrates the use of RQE$CREATE$DESCRIPTOR.       *
*********************************************************************/

/* NUCLUS.EXT declares all system calls */

$INCLUDE(/rmx286/inc/NUCLUS.EXT)
DECLARE TOKEN   LITERALLY 'SELECTOR';
DECLARE desc$token  TOKEN;
DECLARE abs$addr    DWORD;
DECLARE size    WORD;
DECLARE status  WORD;
SAMPLEPROCEDURE:
  PROCEDURE;
  abs$addr = 200000H;   /* The absolute base address of the
       block of memory is 2M bytes. */
  size = 256;   /* The size of the block is 256
       bytes. */
```

RQE$CREATE$DESCRIPTOR

```
/*****************************************************************
* The token desc$token is returned when the calling task invokes  *
* the RQE$CREATE$DESCRIPTOR system call.                          *
*****************************************************************/

  desc$token = RQE$CREATE$DESCRIPTOR (abs$addr, size, @status);

    •
    •     Typical PL/M-286 Statements
    •

END SAMPLEPROCEDURE;
```

Condition Codes

E$OK	0000H	No exceptional conditions.
E$LIMIT	0004H	Creating the requested descriptor would exceed the job's object limit.
E$MEM	0002H	The memory available to the calling task's job is insufficient to create the descriptor.
ENOTCONFIGURED	0008H	This system call is not part of the present configuration.
E$PARAM	8004H	The absolute address specified is larger than 16M bytes.
E$SLOT	000CH	There is no room in the GDT for the new descriptor.

The RQE$DELETE$DESCRIPTOR system call removes a descriptor, originally defined with RQE$CREATE$DESCRIPTOR, from the Global Descriptor Table (GDT).

```
CALL RQE$DELETE$DESCRIPTOR (descriptor, except$ptr);
```

Input Parameter

descriptor	A TOKEN for the descriptor to be deleted.

Output Parameter

except$ptr	A POINTER to a WORD to which the iRMX II Operating System will return the condition code generated by this system call.

Description

This system call removes an entry in the GDT that was originally established with the RQE$CREATE$DESCRIPTOR system call. Once the descriptor is deleted, the GDT slot is returned to the memory manager, which can reassign it when another object is created. However, the memory that was addressed by the descriptor is not returned to the free space manager for reassignment.

Example

```
/*****************************************************************
 * This example illustrates the use of RQE$DELETE$DESCRIPTOR.  First *
 * the example creates a descriptor.  Then, when the descriptor is no *
 * longer needed, RQE$DELETE$DESCRIPTOR is used to delete it.       *
 *****************************************************************/

DECLARE TOKEN      LITERALLY 'SELECTOR';

/*  NUCLUS.EXT declares all system calls */
$INCLUDE(/rmx286/inc/NUCLUS.EXT)

    DECLARE desc$token        TOKEN;
    DECLARE abs$addr          DWORD;
    DECLARE size              WORD;
    DECLARE status            WORD;

SAMPLEPROCEDURE:
  PROCEDURE;
```

```
  abs$addr = 2000000H;              /* The absolute address of the
                        undefined memory area is 2M bytes. */
  size = 256;                 /* The size of the block is
                        256 bytes. */

/*************************************************************************
 *  The token desc$token is returned when the calling task invokes the  *
 *  RQE$CREATE$DESCRIPTOR system call.                                   *
 *************************************************************************/

  desc$token = RQE$CREATE$DESCRIPTOR (abs$addr, size, @status);

        •
        •     Typical PL/M-286 Statements
        •

/*************************************************************************
 * When the descriptor is no longer needed, it may be deleted by a  *
 *  task that knows the descriptor token.                            *
 *                                                                   *
*************************************************************************/

  CALL RQE$DELETE$DESCRIPTOR  (desc$token, @status);

        •
        •     Typical PL/M-286 Statements
        •

END SAMPLEPROCEDURE;
```

Condition Codes

E$OK	0000H	No exceptional conditions.
E$EXIST	0006H	Either the descriptor parameter is not a token for an existing object, or it represents a descriptor whose job is now being deleted.
ENOTCONFIGURED	0008H	This system call is not part of the present configuration.
E$TYPE	8002H	You are attempting to delete an object that isn't a descriptor.

The CATALOG$OBJECT system call places an entry for an object in an object directory.

```
CALL RQ$CATALOG$OBJECT (job, object, name, except$ptr);
```

Input Parameters

job — A TOKEN that indicates where the object is to be cataloged.

- If SELECTOR$OF(NIL), it indicates that the object is to be cataloged in the object directory of the job to which the calling task belongs.
- If a valid selector, it specifies the TOKEN for the job in whose object directory the object is to be cataloged.

object — A TOKEN for the object to be cataloged. A value of SELECTOR$OF(NIL) for this parameter indicates that a null token is being cataloged.

name — A POINTER to a STRING containing the name under which the object is to be cataloged. The name must not be over 12 characters long. Each character can be a byte consisting of any value from 0 to 0FFH.

Output Parameter

except$ptr — A POINTER to a WORD to which the iRMX II Operating System will return the condition code generated by this system call.

Description

The CATALOG$OBJECT system call places an entry for an object in the object directory of a specific job. The entry consists of both a name and a token for the object. There may be several such entries for a single object in a directory, because the object may have several names. (However, in a given object directory, only one object may be cataloged under a given name.) If another task is waiting, via the LOOKUP$OBJECT system call, for the object to be cataloged, that task is awakened when the entry is cataloged.

Example

```
/*****************************************************************
 *  This example illustrates how the CATALOG$OBJECT system call   *
 *  can be used to place an entry in an object directory.         *
*****************************************************************/

DECLARE TOKEN                   LITERALLY 'SELECTOR';
/* NUCLUS.EXT declares all system calls */
$INCLUDE(/rmx286/inc/NUCLUS.EXT)
    DECLARE mbx$token             TOKEN;
    DECLARE mbx$flags             WORD;
    DECLARE job$token             TOKEN;
    DECLARE status                WORD;
SAMPLEPROCEDURE:
    PROCEDURE;
   mbx$flags = 8;           /* designates four objects to be queued
                               on the high performance object
                               queue;  designates a first-in/
                               first-out task queue */
   job$token = SELECTOR$OF(NIL); /* indicates objects to be
                               cataloged into the object
                               directory of the  calling
                               task's job */
•
•     Typical PL/M-286 Statements
•
/*****************************************************************
*  The calling task creates an object, in this example a mailbox, *
*  before cataloging the object's token.                          *
*****************************************************************/
   mbx$token = RQ$CREATE$MAILBOX (mbx$flags,
                                  @status);
•
•     Typical PL/M-286 Statements
•
/*****************************************************************
 *  After creating the mailbox, the calling task catalogs the     *
 *  mailbox token in the object directory of its own job.         *
*****************************************************************/

    CALL RQ$CATALOG$OBJECT        (job$token,
                                   mbx$token,
                                   @(3, 'MBX'),
                                   @status);
•
•     Typical PL/M-286 Statements
•
END SAMPLEPROCEDURE;
```

Condition Codes

E$OK	0000H	No exceptional conditions.
EBADADDR	800FH	The pointer to the name is invalid. Either the selector does not refer to a valid segment, or the offset is outside the segment boundaries.
E$CONTEXT	0005H	At least one of the following is true: • The name being cataloged is already in the designated object directory. • The directory's maximum allowable size is 0.
E$EXIST	0006H	Either the job parameter, which is not SELECTOR$OF(NIL), or the object parameter is not a token for an existing object.
E$LIMIT	0004H	The designated object directory is full.
ENOTCONFIGURED	0008H	This system call is not part of the present Operating System configuration.
E$PARAM	8004H	The first BYTE of the STRING pointed to by the name parameter contains a zero or a value greater than 12.
E$TYPE	8002H	The job parameter is a token for an object which is not a job or is not SELECTOR$OF(NIL).

The RQE$CHANGE$OBJECT$ACCESS system call changes the access rights of iRMX II segments or composite objects.

```
CALL RQE$CHANGE$OBJECT$ACCESS (object, access, reserved, except$ptr);
```

Input Parameters

object — A TOKEN for an object whose access rights are being changed. This token must represent a segment or composite object.

access — A BYTE specifying the new access rights for the object. The following values are valid for iRMX II objects:

Data Segments	Binary Value	Hex Value
Read-only	10010000	90H
Read/write	10010010	92H

Code Segments	Binary Value	Hex Value
Execute-only	10011000	98H
Execute/read	10011010	9AH
Execute only (conforming)	10011100	9CH
Execute/read (conforming)	10011110	9EH

Other values are not appropriate for iRMX II applications.

reserved — A BYTE reserved for future enhancements. Always set this byte to 0.

Output Parameters

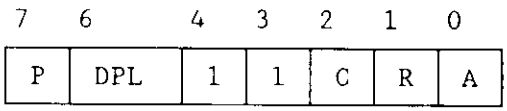
except$ptr — A POINTER to a WORD to which the iRMX II Operating System will return the condition code generated by this system call.

Description

Associated with each 80286 descriptor is an access field that specifies a variety of information about the object described by the descriptor. The RQE$CHANGE$OBJECT$ACCESS system call lets you modify this field for descriptors that represent segments and composite objects. Not all of the fields can be modified, but the fields that describe the segment type and access rights can be changed.

The access byte has the following general format. Bits that must be set a certain way for iRMX II applications are indicated as such.

Access Byte for Code Segments



P Present bit (1=yes). This bit must be 1 for iRMX II applications.

DPL Descriptor privilege level. These two bits must be 0 for iRMX II applications.

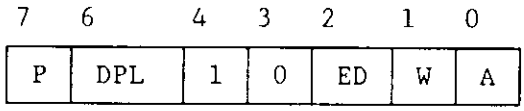
The next two bits must be set to 1 for code segments. Bit 4 indicates a segment descriptor. Bit 3 indicates an executable segment.

C Conforming segment (1=yes, 0=no).

R Readable segment (1=yes, 0=no).

A This bit must be set to zero.

Access Byte for Code Segments



P Present bit (1=yes). This bit must be 1 for iRMX II applications.

DPL Descriptor privilege level. These two bits must be 0 for iRMX II applications.

Bits 4 and 3 must be set as shown for data segments. Bit 4 indicates a segment descriptor. Bit 3 indicates a non-executable segment.

ED Expand down bit (1=expand down). This bit must be 0 for iRMX II applications.

W Writeable segment (1=yes, 0=no).

A This bit must be set to zero.

The description of the "access" input parameter lists the binary and hexadecimal values that are appropriate for iRMX II segments.

Example

```
/*****************************************************************
* This example illustrates the use of RQE$CHANGE$OBJECT$ACCESS   *
  by creating a segment and changing its access rights           *
*****************************************************************/

/* NUCLUS.EXT declares all system calls */

$INCLUDE(/rmx286/inc/NUCLUS.EXT)
DECLARE TOKEN   LITERALLY 'SELECTOR';
DECLARE obj$token   TOKEN;
DECLARE seg$size    WORD;
DECLARE status  WORD;
DECLARE access  BYTE;
DECLARE reserved    BYTE;
SAMPLEPROCEDURE:
  PROCEDURE;
  seg$size = 0256;  /* The size of the requested segment
                       is 256 bytes. */
•
•     Typical PL/M-286 Statements
•
/*****************************************************************
* The token obj$token is returned when the calling task invokes   *
* the CREATE$SEGMENT system call.                                 *
*****************************************************************/

  job$token = RQ$CREATE$SEGMENT   (seg$size, @status);

•
•     Typical PL/M-286 Statements
•
/*****************************************************************
* The access rights are changed to make a writeable data segment, *
* present in memory, and not accessed.                            *
*****************************************************************/

  access = 092H;    /* The bit configuration for a writeable
       data segment, present in memory and
       not accessed. */
  reserved = 0; /* Reserved parameters are always set
       to 0. */
  CALL RQE$CHANGE$OBJECT$ACCESS (obj$token, access, reserved, @status);

•
•     Typical PL/M-286 Statements
•

END SAMPLEPROCEDURE;
```

Condition Codes

E$OK	0000H	No exceptional conditions.
E$EXIST	0006H	The object whose access is to be changed does not exist or is not a valid iRMX II object.
ENOTCONFIGURED	0008H	This system call is not part of the present configuration.
E$TYPE	8002H	The obj$token parameter refers to an object that is neither a segment nor a composite object.

The RQEGETADDRESS system call returns the 24-bit physical address of a logical pointer.

```
phys$addr = RQE$GET$ADDRESS (log$addr, except$ptr);
```

Input Parameter

log$addr	A POINTER containing the segmented address for which the physical address is desired. The segmented address must be in the form: selector:offset.

Output Parameters

phys$addr	A DWORD in which the system call returns the 24-bit physical address of the log$addr parameter.
except$ptr	A POINTER to a WORD to which the Operating System will return the condition code generated by this system call.

Description

In protected virtual address mode, the base portion of an address (a selector) does not specify the physical location of the address. Rather, it points to a descriptor table, where that 24-bit physical address is found. This system call retrieves the 24-bit physical address for the selector portion of a pointer, adds the offset part of the pointer to that value, and returns the resulting physical address of the complete pointer.

Example

```
/*************************************************************************
 *  This example illustrates the use of RQE$GET$ADDRESS by creating      *
 *  a segment, converting the segment's selector to a pointer, and       *
 *  returning the physical address of the segment.                       *
 *************************************************************************/

DECLARE TOKEN     LITERALLY 'SELECTOR';

/*  NUCLUS.EXT declares all system calls */
$INCLUDE(/rmx286/inc/NUCLUS.EXT)
```

```
    DECLARE obj$token           TOKEN;
    DECLARE seg$size            WORD;
    DECLARE status              WORD;
    DECLARE log$addr            POINTER;
    DECLARE phys$addr           DWORD;

SAMPLEPROCEDURE:
  PROCEDURE;

  seg$size = 256;               /* The size of the requested segment is
                                   256 bytes. */

        •
        •     Typical PL/M-286 Statements
        •

/*************************************************************************
 *  The token obj$token is returned when the calling task invokes the   *
 *  CREATE$SEGMENT system call.                                         *
 *************************************************************************/

  obj$token = RQ$CREATE$SEGMENT (seg$size, @status);

/*************************************************************************
 *  The segment selector is converted to a pointer.                     *
 *************************************************************************/

  log$addr = BUILD$PTR(obj$token, 0);

/*************************************************************************
 *  The pointer with the logical address is used to get the physical    *
 *  address.                                                            *
 *************************************************************************/

  phys$addr = RQE$GET$ADDRESS (log$addr, @status);

        •
        •     Typical PL/M-286 Statements
        •

END SAMPLEPROCEDURE;
```

Condition Codes

E$OK	0000H	No exceptional conditions.
EBADADDR	800FH	The segmented address is invalid. Either the selector does not refer to a valid segment, or the offset is outside the segment boundaries.
ENOTCONFIGURED	0008H	This system call is not part of the present configuration.

The RQEGETOBJECT$ACCESS system call returns the access type of an object whose token is specified.

```
CALL RQE$GET$OBJECT$ACCESS (object, access$ptr, except$ptr);
```

Input Parameter

object — A TOKEN for an object whose access rights you want to see.

Output Parameters

access$ptr — A POINTER to a data structure with the following format:

```
STRUCTURE(
      ACCESS        BYTE,
      RESERVED      BYTE);
```

When control returns from this system call, the fields of this structure have the following values:

- ACCESS is a BYTE in which the system call returns the access rights for the object. The following values are typical for iRMX II objects:

Data Segments	Binary Value	Hex Value
Read-only	10010000	90H
Read/write	10010010	92H
Code Segments	**Binary Value**	**Hex Value**
Execute-only	10011000	98H
Execute/read	10011010	9AH
Execute only (conforming)	10011100	9CH
Execute/read (conforming)	10011110	9EH

- RESERVED is a reserved BYTE that must be set to 0.

except$ptr — A POINTER to a WORD to which the Operating System will return the condition code generated by this system call.

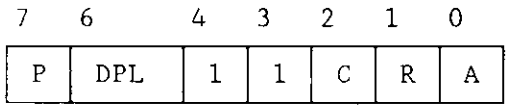
RQEGETOBJECT$ACCESS

Description

Associated with each 80286 descriptor is an access field that specifies a variety of information about the object described by the descriptor. The RQEGETOBJECT$ACCESS system call lets you view this field for descriptors that represent iRMX II objects. The RQE$CHANGE$OBJECT$ACCESS system call can be used to change some of this access information for segment and composite objects.

The access byte has the following general format. Bits that are normally set a certain way for iRMX II applications are indicated as such.

Access Byte for Code Segments



P Present bit (1=yes). This bit is 1 for iRMX II applications.

DPL Descriptor privilege level. These two bits are 0 for iRMX II applications.

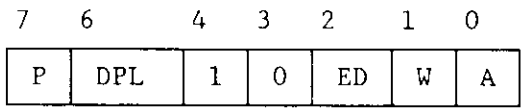
The next two bits are set to 1 for code segments. Bit 4 indicates a segment descriptor. Bit 3 indicates an executable segment.

C Conforming segment (1=yes, 0=no).

R Readable segment (1=yes, 0=no)

A This bit must be set to 0.

Access Byte for Code Segments



P Present bit (1=yes). This bit is 1 for iRMX II applications.

DPL Descriptor privilege level. These two bits are 0 for iRMX II applications.

Bits 4 and 3 are set as shown for data segments. Bit 4 indicates a segment descriptor. Bit 3 indicates a non-executable segment.

ED Expand down bit (1=expand down). This bit is 0 for iRMX II applications.

W Writeable segment (1=yes, 0=no).

A This bit must be set to 0.

The description of the "access" input parameter lists the binary and hexadecimal values that are appropriate for iRMX II objects.

Example

```
/*************************************************************************
* This example illustrates the use of RQE$GET$OBJECT$ACCESS by          *
* creating a segment and then requesting the object's access rights.    *
*************************************************************************/

/* NUCLUS.EXT declares all system calls */
$INCLUDE(/rmx286/inc/NUCLUS.EXT)

DECLARE TOKEN           LITERALLY 'SELECTOR';
DECLARE obj$token           TOKEN;
DECLARE seg$size            WORD;
DECLARE status          WORD;
DECLARE access$struc            STRUCTURE(
                        ACCESS   BYTE,
                        RESERVED BYTE);

SAMPLEPROCEDURE:
  PROCEDURE;

  seg$size = 512;               /* The size of the requested segment
                             is 512 bytes. */

        •
        •     Typical PL/M-286 Statements
        •

/*************************************************************************
* The token obj$token is returned when the calling task invokes the     *
* CREATE$SEGMENT system call.                                           *
*************************************************************************/

  obj$token = CREATE$SEGMENT   (seg$size, @status);

        •
        •     Typical PL/M-286 Statements
        •

/*************************************************************************
* The access rights of the segment object are requested.  The value     *
* returned should be 92H or 90H for a read/write object.                *
*************************************************************************/

  CALL RQE$GET$OBJECT$ACCESS (obj$token, @access$struc, @status);

        •
        •     Typical PL/M-286 Statements
        •

END SAMPLEPROCEDURE;
```

Condition Codes

E$OK	0000H	No exceptional conditions.
EBADADDR	800FH	The access$ptr pointer is invalid. Either the selector does not refer to a valid segment, or the offset is outside the segment boundaries.
E$EXIST	0006H	The object whose access is requested does not exist or is not a valid iRMX II object.
ENOTCONFIGURED	0008H	This system call is not part of the present configuration.

The GET$TYPE system call returns the encoded type of an object.

```
type$code = RQ$GET$TYPE (object, except$ptr);
```

Input Parameter

object — A TOKEN for an object whose type is desired.

Output Parameters

type$code — A WORD which contains the encoded type of the specified object. The types for iRMX II objects are encoded as follows:

Value	Type
1	job
2	task
3	mailbox
4	semaphore
5	region
6	segment
7	extension
100H	composite (user)
101H	composite (connection)
300H	composite (I/O job)
301H	composite (logical device)
8000H - 0FFFFH	user-created composites

User and connection composites are described in the *Extended iRMX II Basic I/O System User's Guide*. I/O jobs and logical device composites are described in the *Extended iRMX II Extended I/O System User's Guide*.

except$ptr — A POINTER to a WORD to which the Operating System will return the condition code generated by this system call.

Description

The GET$TYPE system call returns the type code for an object. For a composite, type$code contains the composite extension type, not the encoded object type.

Example

```
/*************************************************************************
 *  This example illustrates how the GET$TYPE system call can be used   *
 *  to return the encoded type of an object.                            *
 *************************************************************************/

DECLARE TOKEN      LITERALLY 'SELECTOR';

/*  NUCLUS.EXT declares all system calls */
$INCLUDE(/rmx286/inc/NUCLUS.EXT)

    DECLARE type$code                  WORD;
    DECLARE mbx$token                  TOKEN;
    DECLARE calling$tasks$job          TOKEN;
    DECLARE wait$forever               LITERALLY '0FFFFH';
    DECLARE object$token               TOKEN;
    DECLARE response                   TOKEN;
    DECLARE status                     WORD;

SAMPLEPROCEDURE:
    PROCEDURE;

        •
        •     Typical PL/M-286 Statements
        •

/*************************************************************************
 *  In order to invoke the GET$TYPE system call, the calling task must  *
 *  have the token for an object.  In this example, the calling task    *
 *  invokes the LOOKUP$OBJECT system call and then the RECEIVE$MESSAGE  *
 *  system call to receive the token for an object of unknown type      *
 *  (object$token).                                                     *
 *************************************************************************/

    calling$tasks$job = SELECTOR$OF(NIL);

    mbx$token = RQ$LOOKUP$OBJECT       (calling$tasks$job,
                                        @(3,'MBX'),
                                        wait$forever,
                                        @status);

        •
        •     Typical PL/M-286 Statements
        •

/*************************************************************************
 *  The RECEIVE$MESSAGE system call returns object$token to the calling *
 *  task after the calling task invoked LOOKUP$OBJECT to receive the    *
 *  token for the mailbox named 'MBX'.  'MBX' had been designated       *
 *  as the mailbox another task would use to send an object.            *
 *************************************************************************/
```

```
    object$token = RQ$RECEIVE$MESSAGE  (mbx$token,
                                        wait$forever,
                                        @response,
                                        @status);
        •
        •     Typical PL/M-286 Statements
        •

/*************************************************************************
 *  Using the type code returned by the GET$TYPE system call, the       *
 *  calling task can find out if the object is a job, task,             *
 *  mailbox, region, segment, semaphore, extension, or composite.       *
 *************************************************************************/

    type$code = RQ$GET$TYPE            (object$token,
                                        @status);
            •
            •  Typical PL/M Statements
            •

END SAMPLEPROCEDURE;
```

Condition Codes

E$OK	0000H	No exceptional conditions.
E$EXIST	0006H	The object parameter is not a token for an existing object.
ENOTCONFIGURED	0008H	This system call is not part of the present configuration.

The LOOKUP$OBJECT system call returns a token for a cataloged object.

```
object = RQ$LOOKUP$OBJECT (job, name$ptr, time$limit, except$ptr);
```

Input Parameters

job — A TOKEN indicating the object directory to be searched.

- If a valid selector, the TOKEN must contain a token for the job whose object directory is to be searched.
- If SELECTOR$OF(NIL), the object directory to be searched is that of the calling task's job.

name$ptr — A POINTER to a STRING which contains the name under which the object is cataloged. During the lookup operation, upper and lower case letters are treated as being different.

time$limit — A WORD indicating the task's willingness to wait.

- If zero, the WORD indicates that the calling task is not willing to wait.
- If 0FFFFH, the WORD indicates that the task will wait as long as is necessary.
- If between 0 and 0FFFFH, the WORD indicates the number of clock intervals that the task is willing to wait. The length of a clock interval is a configuration option. Refer to the *Extended iRMX II Interactive Configuration Utility Reference Manual* for further information.

Output Parameters

object — A TOKEN containing the requested object token.

except$ptr — A POINTER to a WORD to which the iRMX II Operating System will return the condition code generated by this system call.

Description

The LOOKUP$OBJECT system call returns the token for an object after searching for its name in the specified object directory. Because it is possible that the object is not cataloged at the time of the call, the calling task has the option of waiting, either indefinitely or for a specific period of time, for another task to catalog the object.

Example

```
/*************************************************************************
 *  This example illustrates how the LOOKUP$OBJECT system call can be   *
 *  used to return a token for a cataloged object.                       *
 *************************************************************************/

DECLARE TOKEN      LITERALLY 'SELECTOR';

/*  NUCLUS.EXT declares all system calls */
$INCLUDE(/rmx286/inc/NUCLUS.EXT)
    DECLARE mbx$token                  TOKEN;
    DECLARE calling$tasks$job          TOKEN;
    DECLARE wait$forever               LITERALLY '0FFFFH';
    DECLARE status                     WORD;
SAMPLEPROCEDURE:
    PROCEDURE;

        •
        •     Typical PL/M-286 Statements
        •

/*************************************************************************
 *  In this example, the calling task invokes LOOKUP$OBJECT in order to *
 *  search the object directory of the calling task's job for an object *
 *  with the name 'MBX'.                                                 *
 *************************************************************************/
    calling$tasks$job = SELECTOR$OF(NIL);
    mbx$token = RQ$LOOKUP$OBJECT       (calling$tasks$job,
                                        @(3,'MBX'),
                                        wait$forever,
                                        @status);
        •
        •     Typical PL/M-286 Statements
        •

END SAMPLEPROCEDURE;
```

Condition Codes

E$OK	0000H	No exceptional conditions.
EBADADDR	800FH	The pointer to the name string is invalid. Either the selector does not refer to a valid segment, or the offset is outside the segment boundaries.
E$CONTEXT	0005H	The specified job has an object directory of size 0.

E$EXIST	0006H	At least one of the following is true: • The job parameter (which is not SELECTOR$OF(NIL)) is not a token for an existing object. • The name was found, but the cataloged object has a null (NIL) token.
E$LIMIT	0004H	The specified object directory is full and the object being looked up has not yet been cataloged. This code (rather than E$TIME) is returned when a full object directory does not contain the requested object and the calling task is not willing to wait.
ENOTCONFIGURED	0008H	This system call is not part of the present configuration.
E$PARAM	8004H	The first byte of the string pointed to by the name parameter contains a value greater than 12 or equal to 0.
E$TIME	0001H	One of the following is true: • The calling task indicated its willingness to wait a certain amount of time, but the waiting period elapsed before the object became available. • The task was not willing to wait, the entry indicated by the name parameter is not in the specified object directory, and the object directory is not full.
E$TYPE	8002H	The job parameter contains a token for an object that is not a job.

The UNCATALOG$OBJECT system call removes an entry for an object from an object directory.

```
CALL RQ$UNCATALOG$OBJECT (job, name, except$ptr);
```

Input Parameters

job — A TOKEN indicating the job of the object directory from which an entry is to be deleted.

- If a valid selector, the TOKEN contains a token for the job from whose object directory the specified entry is to be deleted.
- If SELECTOR$OF(NIL), the entry is to be deleted from the object directory of the calling task's job.

name — A POINTER to a STRING containing the name of the object whose entry is to be deleted.

Output Parameter

except$ptr — A POINTER to a WORD to which the iRMX II Operating System will return the condition code generated by this system call.

Description

The UNCATALOG$OBJECT system call deletes an entry from the object directory of the specified job.

Example

```
/*************************************************************************
 *  This example illustrates how the UNCATALOG$OBJECT system call can   *
 *  be used.                                                            *
 *************************************************************************/

 DECLARE TOKEN      LITERALLY 'SELECTOR';

/*  NUCLUS.EXT declares all system calls */
$INCLUDE(/rmx286/inc/NUCLUS.EXT)
```

```
    DECLARE seg$token    TOKEN;
    DECLARE size         WORD;
    DECLARE mbx$token    TOKEN;
    DECLARE mbx$flags    WORD;
    DECLARE no$response  LITERALLY '0';
    DECLARE status       WORD;
    DECLARE job$token    TOKEN;

SAMPLEPROCEDURE:
    PROCEDURE;

    size = 64;                          /* designates new segment to contain
                                           64 bytes */
    mbx$flags = 0;                      /* designates four objects to be
                                           queued on the high performance
                                           object queue;  designates a /
                                           first-in first-out task queue */
    job$token = SELECTOR$OF(NIL);  /*indicates objects to be cataloged
                                     into the object directory of the
                                     calling task's job */
        •
        •     Typical PL/M-286 Statements
        •

/************************************************************************
 *  The calling task creates a segment and a mailbox and catalogs the   *
 *  mailbox TOKEN.  The calling task then uses the TOKENs for both      *
 *  objects to send a message.                                          *
 ************************************************************************/

    seg$token = RQ$CREATE$SEGMENT  (size,
                                    @status);
    mbx$token = RQ$CREATE$MAILBOX  (mbx$flags,
                                    @status);

/************************************************************************
 *  It is not mandatory for the calling task to catalog the mailbox     *
 *  token in order to send a message.  It is necessary, however, to     *
 *  catalog the mailbox token if a task in another job is to receive    *
 *  the message.                                                        *
 ************************************************************************/

    CALL RQ$CATALOG$OBJECT              (job$token,
                                         mbx$token,
                                         @(3, 'MBX'),
                                         @status);
        •
        •     Typical PL/M-286 Statements
        •
```

```
/*************************************************************************
 *  The calling task invokes the SEND$MESSAGE system call to send the    *
 *  token for the segment to the specified mailbox.                      *
 *************************************************************************/

   CALL RQ$SEND$MESSAGE              (mbx$token,
                                     seg$token,
                                     no$response,
                                     @status);
        •
        •     Typical PL/M-286 Statements
        •

/*************************************************************************
 *  When the mailbox is no longer needed and there is no need to keep    *
 *  its token cataloged, it may be deleted by any task that knows its    *
 *  token.                                                               *
 *************************************************************************/

   CALL RQ$UNCATALOG$OBJECT          (job$token,
                                     @(3,'MBX'),
                                     @status);
   CALL RQ$DELETE$MAILBOX            (mbx$token,
                                     @status);
        •
        •     Typical PL/M-286 Statements
        •

END SAMPLEPROCEDURE;
```

Condition Codes

E$OK	0000H	No exceptional conditions.
EBADADDR	800FH	The pointer to the string is invalid. Either the selector doesn't refer to a valid segment, or the offset is outside the segment boundaries.
E$CONTEXT	0005H	The specified object directory does not contain an entry with the designated name.
E$EXIST	0006H	The job parameter is neither zero nor a token for an existing object.
ENOTCONFIGURED	0008H	This system call is not part of the present configuration.

E$PARAM	8004H	The first byte of the STRING pointed to by the name parameter contains a value greater than 12 or equal to 0.
E$TYPE	8002H	The job parameter is a token for an object that is not a job.

The GET$EXCEPT$HANDLER system call returns information about the calling task's exception handler.

```
CALL RQ$GET$EXCEPTION$HANDLER (exception$info$ptr, except$ptr);
```

Output Parameters

exception$info$ptr — A POINTER to a structure of the following form:

```
STRUCTURE(
    EXCEPTION$HANDLER$PTR       POINTER,
    EXCEPTION$MODE              BYTE);
```

where, after the call,

- EXCEPTION$HANDLER$POINTER points to the first instruction of the exception handler. If this pointer is NIL, the calling task's exception handler is the system default exception handler.
- EXCEPTION$MODE contains an encoded indication of the calling task's current exception mode. The value is interpreted as follows:

Value	When to Pass Control to Exception Handler
0	Never
1	On programmer errors only
2	On environmental conditions only
3	On all exceptional conditions

except$ptr — A POINTER to a WORD to which the Operating System will return the condition code generated by this system call.

Description

The GET$EXCEPTION$HANDLER system call returns both the address of the calling task's exception handler and the current value of the task's exception mode.

Example

```
/*************************************************************************
 *  This example illustrates how the GET$EXCEPTION$HANDLER system call   *
 *  can be used to return information about the calling task's           *
 *  exception handler.                                                   *
 *************************************************************************/

DECLARE TOKEN     LITERALLY 'SELECTOR';

/*  NUCLUS.EXT declares all system calls */
$INCLUDE(/rmx286/inc/NUCLUS.EXT)

    DECLARE x$handler  STRUCTURE   (x$handler$pointer POINTER,
                                    x$mode             BYTE);
    DECLARE status                                     WORD;

SAMPLEPROCEDURE:
    PROCEDURE;
        •
        •     Typical PL/M-286 Statements
        •

/*************************************************************************
 *  The address of the calling task's exception handler and the value    *
 *  of the task's exception mode (which specifies when to pass control   *
 *  to the exception handler) are both returned when the calling task    *
 *  invokes the GET$EXCEPTION$HANDLER system call.                       *
 *************************************************************************/

    CALL RQ$GET$EXCEPTION$HANDLER  (@x$handler, @status);
        •
        •     Typical PL/M-286 Statements
        •

END SAMPLEPROCEDURE;
```

Condition Codes

E$OK	0000H	No exceptional conditions.
EBADADDR	800FH	The pointer is invalid. Either the selector does not refer to a valid segment, or the offset is outside the segment boundaries.
ENOTCONFIGURED	0008H	This system call is not part of the present configuration.

The SET$EXCEPTION$HANDLER system call assigns an exception handler to the calling task.

```
CALL RQ$SET$EXCEPTION$HANDLER (exception$info$ptr, except$ptr);
```

Input Parameter

exception$info$ptr — A POINTER to a structure of the following form:

```
STRUCTURE(
   EXCEPTION$HANDLER$PTR      POINTER,
   EXCEPTION$MODE             BYTE);
```

where:

- exception$handler$ptr points to the first instruction of the exception handler.
- exception$mode contains an encoded indication of the calling task's intended exception mode. The value is interpreted as follows:

Value	When to Pass Control to Exception Handler
0	Never
1	On programmer errors only
2	On environmental conditions only
3	On all exceptional conditions

If EXCEPTION$HANDLER$PTR equals NIL, the exception handler of the calling task's parent job is assigned.

Output Parameter

except$ptr — A POINTER to a WORD to which the iRMX II Operating System will return the condition code generated by this system call.

SET$EXCEPTION$HANDLER

Description

The SET$EXCEPTION$HANDLER system call enables a task to set its exception handler and exception mode attributes. If you want to designate the Debugger as the exception handler to interactively examine system objects and lists, the following code sets up the needed structure in PL/M-286:

```
DECLARE  except$ptr STRUCTURE (ptr       POINTER,
                               mode      BYTE);/* establish a
                                            structure for
                                           exception handlers */

DECLARE exception WORD;

except$ptr.ptr = @my_excep_handler  /*"@my_excep_handler is your own
                                      procedure, here it  designates the
                                      debugger as the exception handler*/

x.mode = ZERO$ONE$TWO$OR$THREE;     /* the mode is a value 0-3 */
CALL  RQ$SET$EXCEPTION$HANDLER (@x, @exception);
```

Example

```
/*************************************************************************
 *  This example illustrates how the SET$EXCEPTION$HANDLER system call  *
 *  can be used to assign an exception handler to the calling task.     *
*************************************************************************/

DECLARE TOKEN      LITERALLY 'SELECTOR';

/*  NUCLUS.EXT declares all system calls */
$INCLUDE(/rmx286/inc/NUCLUS.EXT)

    EXCEPTIONHANDLER: PROCEDURE EXTERNAL;
    END EXCEPTIONHANDLER;
    DECLARE X$HANDLER$STRUCTURE     LITERALLY 'STRUCTURE(  ptr     POINTER,
                                                           mode    BYTE)';
                                     /* establishes a structure for
                                      exception handlers */
    DECLARE x$handler                X$HANDLER$STRUCTURE;
                                     /* using the exception handler
                                      structure, the pointer to the
                                      old exception handler is
                                      defined */
    DECLARE new$x$handler            X$HANDLER$STRUCTURE;
                                     /* using the exception handler
                                        structure, the new exception
                                        handler is defined */
```

```
    DECLARE all$exceptions          LITERALLY '3';
                                    /* control is passed to the exception
                                       handler on all exceptional
                                       conditions */
    DECLARE PTR$OVERLAY             LITERALLY 'STRUCTURE( offset  WORD,
                                                          base    TOKEN)';
                                   /* establishes a structure for
                                       overlays */
    DECLARE seg$pointer             POINTER;
    DECLARE seg$pointer$ovly        PTR$OVERLAY AT (@seg$pointer);
                                    /* using the overlay structure, the
                                       first instruction of the
                                       exception handler is identified */
    DECLARE status                  WORD;

SAMPLEPROCEDURE:
    PROCEDURE;

seg$pointer = @EXCEPTIONHANDLER;     /* pointer to exception handler */
new$x$handler.offset = seg$pointer$ovly.offset;
                                    /* offset of the first instruction
                                       of the exception handler */
new$x$handler.base = seg$pointer$ovly.base;
                                    /* base address of the exception
                                       handler CPU segment containing
                                       the first instruction of the
                                       exception handler */
new$x$handler.mode = all$exceptions;  /* pass control on all conditions */

        •
        •    Typical PL/M-286 Statements
        •

/*************************************************************************
 *  The address of the calling task's exception handler and the value   *
 *  of the task's exception mode (when to pass control to the exception *
 *  handler) are both returned when the calling task invokes the        *
 *  GET$EXCEPTION$HANDLER system call.                                  *
 *************************************************************************/

    CALL RQ$GET$EXCEPTION$HANDLER     (@x$handler,
                                       @status);

        •
        •    Typical PL/M-286 Statements
        •
```

```
/*************************************************************************
 *  The calling task may invoke the SET$EXCEPTION$HANDLER system call   *
 *  to first set a new exception handler and then to later reset the    *
 *  old exception handler.                                              *
*************************************************************************/

    CALL RQ$SET$EXCEPTION$HANDLER     (@new$x$handler,
                                       @status);
        •
        •    Typical PL/M-286 Statements
        •

/*************************************************************************
 *  No longer needing the new exception handler, the calling task uses  *
 *  the address and mode of the old exception handler to return         *
 *  exception handling to its original exception handler.               *
*************************************************************************/

    CALL RQ$SET$EXCEPTION$HANDLER     (@x$handler,
                                         @status);
        •
        •    Typical PL/M-286 Statements
        •

END SAMPLEPROCEDURE;
```

Condition Codes

E$OK	0000H	No exceptional conditions.
EBADADDR	800FH	The exception$info$ptr is invalid. Either the selector does not refer to a valid segment, or the offset is outside the segment boundaries.
ENOTCONFIGURED	0008H	This system call is not part of the present configuration.
E$PARAM	8004H	The exception$mode parameter is greater than 3.

The DISABLE system call disables an interrupt level.

```
CALL RQ$DISABLE (level, except$ptr);
```

Input Parameter

level — A WORD that specifies an interrupt level encoded as follows (bit 15 is the high-order bit):

Bits	Value
15-7	Reserved bits that should be set to zero
6-4	First digit of the interrupt level (0-7)
3	If one, the level is a master level and bits 6-4 specify the entire level number
	If zero, the level is a slave level and bits 2-0 specify the second digit
2-0	Second digit of the interrupt level (0-7), if bit 3 is zero

Output Parameter

except$ptr — A POINTER to a WORD to which the Operating System will return the condition code generated by this system call. All exceptional conditions must be processed in-line. Control does not pass to an exception handler.

Description

The DISABLE system call disables the specified interrupt level. It has no effect on other levels. To be disabled, a level must have an interrupt handler assigned to it. Otherwise, the Nucleus returns an E$CONTEXT Exception code.

You must not disable the level reserved for the system clock. You determine this level during system configuration (refer to the *Extended iRMX II Interactive Configuration Utility Reference Manual*).

Example

```
/*****************************************************************
 *  This example illustrates how the DISABLE system call can be     *
 *  used to disable an interrupt level.                              *
*****************************************************************/

DECLARE TOKEN      LITERALLY 'SELECTOR';

/*  NUCLUS.EXT declares all system calls */
$INCLUDE(/rmx286/inc/NUCLUS.EXT)

INTERRUPTHANDLER: PROCEDURE INTERRUPT EXTERNAL;
END INTERRUPTHANDLER;

    DECLARE interrupt$level$7     LITERALLY '78H';
                            /* specifies master interrupt level 7 */
    DECLARE interrupt$task$flag   BYTE;
    DECLARE interrupt$handler     POINTER;
    DECLARE data$segment          TOKEN;
    DECLARE status                WORD;
    DECLARE job$token             TOKEN;

SAMPLEPROCEDURE:
    PROCEDURE;

    interrupt$task$flag = 0;  /* indicates no interrupt task on level
                                 7 */

    data$segment = SELECTOR$OF(NIL);  /* indicates that interrupt
                                         handler will load its own
                                         data segment */

        •
        •     Typical PL/M-286 Statements
        •

/*****************************************************************
 *  An interrupt level must have an interrupt handler or an interrupt   *
 *  task assigned to it.  Invoking the SET$INTERRUPT system call, the   *
 *  calling task assigns INTERRUPTHANDLER to interrupt level 7.        *
 *****************************************************************/

    CALL RQ$SET$INTERRUPT         (interrupt$level$7,
                                   interrupt$task$flag,
                                   @INTERRUPTHANDLER,
                                   data$segment,
                                   @status);
```

```
        •
        •     Typical PL/M-286 Statements
        •

/*************************************************************************
*  The SET$INTERRUPT system call enabled interrupt level 7.  In order  *
*  to disable level 7, the calling task invokes the DISABLE system     *
*  call.                                                                *
*************************************************************************/

    CALL RQ$DISABLE                (interrupt$level$7,
                                    @status);

END SAMPLEPROCEDURE;
```

Condition Codes

E$OK	0000H	No exceptional conditions.
E$CONTEXT	0005H	The level indicated by the level parameter is already disabled or has no interrupt handler assigned to it.
ENOTCONFIGURED	0008H	This system call is not part of the present configuration.
E$PARAM	8004H	The level parameter is invalid.

The ENABLE system call enables an interrupt level.

```
CALL RQ$ENABLE (level, except$ptr);
```

Input Parameter

level — A WORD that specifies an interrupt level that is encoded as follows (bit 15 is the high-order bit):

Bits	Value
15-7	Reserved bits that should be set to zero.
6-4	First digit of the interrupt level (0-7)
3	If one, the level is a master level and bits 6-4 specify the entire level number
	If zero, the level is a slave level and bits 2-0 specify the second digit
2-0	Second digit of the interrupt level (0-7), if bit 3 is zero

Output Parameter

except$ptr — A POINTER to a WORD to which the Operating System will return the condition code generated by this system call.

Description

The ENABLE system call enables the specified interrupt level. The level must have an interrupt handler assigned to it. A task must not enable the level associated with the system clock.

Example

```
/*************************************************************************
 *  This example illustrates how the ENABLE system call can be used to  *
 *  enable an interrupt level.                                          *
 *************************************************************************/
```

```
DECLARE TOKEN       LITERALLY 'SELECTOR';

/* NUCLUS.EXT declares all system calls */
$INCLUDE(/rmx286/inc/NUCLUS.EXT)

INTERRUPTHANDLER: PROCEDURE INTERRUPT EXTERNAL;
END INTERRUPTHANDLER;

    DECLARE interrupt$level$7     LITERALLY '78H';
                                  /* specifies master interrupt level 7*/
    DECLARE interrupt$task$flag   BYTE;
    DECLARE interrupt$handler     POINTER;
    DECLARE data$segment          TOKEN;
    DECLARE status                WORD;

SAMPLEPROCEDURE:
    PROCEDURE;

    interrupt$task$flag = 0;      /* indicates no interrupt task on level
                                     7 */

    data$segment = SELECTOR$OF(NIL);  /* indicates that interrupt handler
                                      will load its own data segment */

    •
    •     Typical PL/M-286 Statements
    •

/*************************************************************************
 *  An interrupt level must have an interrupt handler or an interrupt   *
 *  task assigned to it.  Invoking the SET$INTERRUPT system call, the   *
 *  calling task assigns INTERRUPTHANDLER to interrupt level 7.        *
 *************************************************************************/

    CALL RQ$SET$INTERRUPT         (interrupt$level$7,
                                   interrupt$task$flag,
                                   @INTERRUPTHANDLER,
                                   data$segment,
                                   @status);

        •
        •     Typical PL/M-286 Statements
        •

/*************************************************************************
 *  The SET$INTERRUPT system call enabled interrupt level 7.  In order  *
 *  to illustrate the use of the ENABLE system call, interrupt level 7  *
 *  must first be disabled.  The calling task invokes the DISABLE       *
 *  system call to disable interrupt level 7.                           *
 *************************************************************************/
```

```
   CALL RQ$DISABLE              (interrupt$level$7,
                                 @status);

/*************************************************************************
 *  When an interrupt level needs to be enabled, a task must invoke the *
 *  ENABLE system call.                                                  *
 *************************************************************************/

   CALL RQ$ENABLE               (interrupt$level$7,
                                 @status);
       •
       •     Typical PL/M-286 Statements
       •

END SAMPLEPROCEDURE;
```

Condition Codes

E$OK	0000H	No exceptional conditions.
E$CONTEXT	0005H	At least one of the following is true: • A non-interrupt task tried to enable a level that was already enabled. • There is not an interrupt handler assigned to the specified level. • There has been an interrupt overflow on the specified level.
ENOTCONFIGURED	0008H	This system call is not part of the present configuration.
E$PARAM	8004H	The level parameter is invalid.

The END$INIT$TASK system call is used by an initialization task of a first-level job to inform the root task that it has completed its synchronous initialization process.

```
CALL RQ$END$INIT$TASK;
```

Description

When the initialization task finishes its synchronous initialization, it must inform the root task that it is finished, so that the root task can resume execution and create another first-level job. When you call END$INIT$TASK, the root task resumes execution, allowing it to create the next first-level job. You must include this system call in the initialization task of each first-level job, even if the jobs require no synchronous initialization. Refer to the *Extended iRMX II Interactive Configuration Utility Reference Manual* for more information on first-level jobs and the initialization process.

The ENTER$INTERRUPT system call is used by interrupt handlers to load a previously-specified segment base address into the DS register.

```
CALL RQ$ENTER$INTERRUPT(level, except$ptr);
```

Input Parameter

level — A WORD specifying an interrupt level that is encoded as follows (bit 15 is the high-order bit):

Bits	Value
15-7	Reserved bits that should be set to zero.
6-4	First digit of the interrupt level (0-7)
3	If one, the level is a master level and bits 6-4 specify the entire level number If zero, the level is a slave level and bits 2-0 specify the second digit
2-0	Second digit of the interrupt level (0-7), if bit 3 is zero

Output Parameter

except$ptr — A POINTER to a WORD to which the Operating System will return the condition code generated by this system call. For this system call, all exceptional conditions must be processed in-line. Control does not pass to an exception handler.

Description

ENTER$INTERRUPT, on behalf of the calling interrupt handler, loads a base address value into the DS register. The value is what was specified when the interrupt handler was set up by an earlier call to SET$INTERRUPT.

If the handler is going to call an interrupt task, ENTER$INTERRUPT allows the handler to place data in the CPU data segment that will be used by the interrupt task. This provides a mechanism for the interrupt handler to pass data to the interrupt task.

Example

```
/*************************************************************************
 *  This example illustrates how the ENTER$INTERRUPT system call can be *
 *  used to load a segment base address into the data segment register. *
 *************************************************************************/

DECLARE TOKEN       LITERALLY 'SELECTOR';

/*  NUCLUS.EXT declares all system calls */
$INCLUDE(/rmx286/inc/NUCLUS.EXT)

    DECLARE the$first$word  WORD;
    DECLARE E$OK                    LITERALLY '00H';
    DECLARE interrupt$level$7       LITERALLY '78H';
                                    /* specifies master interrupt level 7 */

    DECLARE interrupt$task$flag     BYTE;
    DECLARE interrupt$handler       POINTER;
    DECLARE data$segment            TOKEN;
    DECLARE status                  WORD;
    DECLARE interrupt$status        WORD;
    DECLARE ds$pointer              POINTER;
    DECLARE PTR$OVERLAY             LITERALLY 'STRUCTURE (offset   WORD,
                                                          base     TOKEN)';
                                    /* establishes a structure for
                                       overlays */
    DECLARE ds$pointer$ovly         PTR$OVERLAY AT (@ds$pointer);
                                    /* using the overlay structure, the
                                       base address of the interrupt
                                       handler's data segment is
                                       identified */

INTERRUPTHANDLER: PROCEDURE INTERRUPT PUBLIC;       /* ENTER$INTERRUPT
                                                    establishes the
                                                    actual level. */
    •
    •     Typical PL/M-286 Statements
    •

/*************************************************************************
 *  The calling interrupt handler invokes the ENTER$INTERRUPT system    *
 *  call which loads a base address value (defined by                   *
 *  ds$pointer$ovly.base) into the data segment register.               *
 *************************************************************************/

    CALL RQ$ENTER$INTERRUPT       (interrupt$level$7,
                                   @interrupt$status);
    CALL INLINEERRORPROCESS     (interrupt$status);
```

```
        •
        •     Typical PL/M-286 Statements
        •

/*************************************************************************
 *  Interrupt handlers that do not invoke interrupt tasks need to       *
 *  invoke the EXIT$INTERRUPT system call to send an end-of-interrupt    *
 *  signal to the hardware.                                              *
 *************************************************************************/

    CALL RQ$EXIT$INTERRUPT        (interrupt$level$7,
                                   @interrupt$status);
    CALL INLINEERRORPROCESS     (interrupt$status);
END INTERRUPTHANDLER;

INLINEERRORPROCESS: PROCEDURE (int$status);
    DECLARE  int$status   WORD;

    IF int$status <> E$OK THEN
         DO;
              •
              •     Typical PL/M-286 Statements
              •
         END;
END INLINEERRORPROCESS;

SAMPLEPROCEDURE:
    PROCEDURE;

    ds$pointer = @the$first$word; /* a dummy identifier used to point to
                                     interrupt handler's data segment */
    data$segment = ds$pointer$ovly.base;
                                  /* identifies the base address of the
                                     interrupt handler's data segment */

    interrupt$task$flag = 0;      /* indicates no interrupt task on level
                                     7 */

    •
    •     Typical PL/M-286 Statements
    •

/*************************************************************************
 *  By first invoking the SET$INTERRUPT system call, the calling task    *
 *  sets up an interrupt level.                                          *
 *************************************************************************/
```

```
    CALL RQ$SET$INTERRUPT          (interrupt$level$7,
                                    interrupt$task$flag,
                                    @INTERRUPTHANDLER,
                                    data$segment,
                                    @status);
    •
    •     Typical PL/M-286 Statements
    •

END SAMPLEPROCEDURE;
```

Condition Codes

E$OK	0000H	No exceptional conditions.
E$CONTEXT	0005H	No segment base value has previously been specified in the call to SET$INTERRUPT.
ENOTCONFIGURED	0008H	This system call is not included in the present configuration.
E$PARAM	8004H	The level parameter is invalid.

The EXIT$INTERRUPT system call is used by interrupt handlers when they don't invoke interrupt tasks; this call sends an end-of-interrupt signal to the hardware.

```
CALL RQ$EXIT$INTERRUPT (level, except$ptr);
```

Input Parameter

level — A WORD specifying an interrupt level that is encoded as follows (bit 15 is the high-order bit):

Bits	Value
15-7	Reserved bits that should be set to zero
6-4	First digit of the interrupt level (0-7)
3	If one, the level is a master level and bits 6-4 specify the entire level number If zero, the level is a slave level and bits 2-0 specify the second digit of the interrupt level
2-0	Second digit of the interrupt level (0-7), if bit 3 is zero

Output Parameter

except$ptr — A POINTER to a WORD to which the Operating System will return the condition code generated by this system call. All exceptional conditions must be processed in-line, as control does not pass to an exception handler.

Description

The EXIT$INTERRUPT system call sends an end-of-interrupt signal to the hardware. This sets the stage for re-enabling interrupts. The re-enabling actually occurs when control passes from the interrupt handler to an application task.

Example

```
/*************************************************************************
 *  This example illustrates how the EXIT$INTERRUPT system call can be  *
 *  used to send an end-of-interrupt signal to the hardware.            *
 *************************************************************************/

DECLARE TOKEN       LITERALLY 'SELECTOR';

/*  NUCLUS.EXT declares all system calls */
$INCLUDE(/rmx286/inc/NUCLUS.EXT)

    DECLARE interrupt$level$7   LITERALLY '78H;
                                /* specifies master interrupt level 7 */
    DECLARE E$OK                LITERALLY '00h';
    DECLARE interrupt$task$flag BYTE;
    DECLARE interrupt$handler   POINTER;
    DECLARE data$segment        TOKEN;
    DECLARE status              WORD;
    DECLARE interrupt$status    WORD;

INTERRUPTHANDLER: PROCEDURE INTERRUPT PUBLIC;          /*  ENTER$INTERRUPT
                                                       establishes actual
                                                       level */
        •
        •     Typical PL/M-286 Statements
        •

/*************************************************************************
 *  Interrupt handlers that do not invoke interrupt tasks need to       *
 *  invoke the EXIT$INTERRUPT system call to send an end-of-interrupt   *
 *  signal to the hardware.                                             *
 *************************************************************************/

    CALL RQ$EXIT$INTERRUPT        (interrupt$level$7,
                                   @interrupt$status);
    IF interrupt$status <> E$OK THEN
         DO;
              •
              •     Typical PL/M-286 Statements
              •
         END;

END INTERRUPTHANDLER;

SAMPLEPROCEDURE:
    PROCEDURE;
```

```
    interrupt$task$flag = 0;     /* indicates no interrupt task on
                                    level 7 */
    data$segment = SELECTOR$OF(NIL);  /* indicates that the interrupt
                                         handler will load its own data
                                         segment */
    •
    •     Typical PL/M-286 Statements
    •

/************************************************************************
 *  By first invoking the SET$INTERRUPT system call, the calling task   *
 *  sets up an interrupt level.                                         *
 ************************************************************************/

    CALL RQ$SET$INTERRUPT       (interrupt$level$7,
                                 interrupt$task$flag,
                                 @INTERRUPTHANDLER,
                                 data$segment,
                                 @status);
    •
    •     Typical PL/M-286 Statements
    •

END SAMPLEPROCEDURE;
```

Condition Codes

E$OK	0000H	No exceptional conditions.
E$CONTEXT	0005H	The SET$INTERRUPT system call has not been invoked for the specified level.
ENOTCONFIGURED	0008H	This system call is not part of the present configuration.
E$PARAM	8004H	The level parameter is invalid.

The GET$LEVEL system call returns the number of the level of the highest priority interrupt being serviced.

```
level = RQ$GET$LEVEL (except$ptr);
```

Output Parameters

level — A WORD whose value is interpreted as follows (bit 15 is the high-order bit):

Bits	Value
15-8	Reserved bits that are set to zero
7	If zero, some level is being serviced and bits 6-0 are significant
	If one, no level is being serviced and bits 6-0 are not significant
6-4	First digit of the interrupt level (0-7)
3	If one, the level is a master level and bits 6-4 specify the entire level number
	If zero, the level is a slave level and bits 2-0 specify the second digit
2-0	Second digit of the interrupt level (0-7), if bit 3 is zero

except$ptr — A POINTER to a WORD to which the Operating System will return the condition code generated by this system call.

Description

The GET$LEVEL system call returns to the calling task the highest (numerically lowest) level which an interrupt handler has started servicing but has not yet finished.

Example

```
/*************************************************************************
 *  This example illustrates how the GET$LEVEL system call can be used. *
 *************************************************************************/

DECLARE TOKEN      LITERALLY 'SELECTOR';

/*  NUCLUS.EXT declares all system calls */
$INCLUDE(/rmx286/inc/NUCLUS.EXT)

    DECLARE interrupt$level  WORD;
    DECLARE status     WORD;

SAMPLEPROCEDURE:
    PROCEDURE;

        •
        •     Typical PL/M-286 Statements
        •

/*************************************************************************
 * The GET$LEVEL system call returns to the interrupt handler the number *
 * of  the highest interrupt level being serviced.                        *
 *************************************************************************/

    interrupt$level = RQ$GET$LEVEL (@status);

        •
        •     Typical PL/M-286 Statements
        •

END SAMPLEPROCEDURE;
```

Condition Codes

E$OK	0000H	No exceptional conditions.
ENOTCONFIGURED	0008H	This system call is not part of the present configuration.

The RESET$INTERRUPT system call cancels the assignment of an interrupt handler to a level.

```
CALL RQ$RESET$INTERRUPT (level, except$ptr);
```

Input Parameter

level — A WORD specifying an interrupt level. This word must be encoded as follows (bit 15 is the high-order bit):

Bits	Value
15-7	Reserved bits that should be set to zero.
6-4	First digit of the interrupt level (0-7).
3	If one, the level is a master level and bits 6-4 specify the entire level number. If zero, the level is a slave level and bits 2-0 specify the second digit.
2-0	Second digit of the interrupt level (0-7), if bit 3 is zero.

Output Parameter

except$ptr — A POINTER to a WORD to which the iRMX II Operating System will return the condition code generated by this system call.

Description

The RESET$INTERRUPT system call cancels the assignment of the current interrupt handler to the specified interrupt level. If an interrupt task has also been assigned to the level, the interrupt task is deleted. RESET$INTERRUPT also disables the level.

The level reserved for the system clock should not be reset and is considered invalid. This level is a configuration option (refer to the *Extended iRMX II Interactive Configuration Utility Reference Manual* for further information).

Example

```
/*************************************************************************
 *  This example illustrates how the RESET$INTERRUPT system call can be *
 *  used to cancel the assignment of an interrupt handler to an         *
 *  interrupt level.                                                    *
 *************************************************************************/

DECLARE TOKEN      LITERALLY 'SELECTOR';

/*  NUCLUS.EXT declares all system calls */
$INCLUDE(/rmx286/inc/NUCLUS.EXT)
INTERRUPTHANDLER:  PROCEDURE INTERRUPT EXTERNAL;
END INTERRUPTHANDLER;

    DECLARE task$token                  TOKEN;
    DECLARE priority$level$210           LITERALLY '210';
    DECLARE start$address               POINTER;
    DECLARE data$segment                TOKEN;
    DECLARE stack$pointer               POINTER;
    DECLARE stack$size$512              LITERALLY '512'; /*new task's stack
                                                          size is 512 bytes*/
    DECLARE task$flags                  WORD;
    DECLARE interrupt$level$7           LITERALLY '78H';
         /* specifies master interrupt level 7 */
    DECLARE interrupt$task$flag         BYTE;
    DECLARE interrupt$status            WORD;
    DECLARE status                      WORD;

INTERRUPTTASK: PROCEDURE PUBLIC;

    interrupt$task$flag = 001H;         /* indicates that calling task is
                                           to be the interrupt task */
    data$segment = SELECTOR$OF(NIL);  /* use own data segment */

/*************************************************************************
 *  The first system call in this example, SET$INTERRUPT, makes the     *
 *  calling task (INTERRUPTTASK) the interrupt task for the interrupt  *
 *  level.                                                              *
 *************************************************************************/
  CALL RQ$SET$INTERRUPT          (interrupt$level$7,
                                  interrupt$task$flag,
                                  @INTERRUPTHANDLER,
                                  data$segment,
                                  @interrupt$status);
/*************************************************************************
 *  The second system call, WAIT$INTERRUPT, is used by the interrupt    *
 *  task to signal its readiness to service an interrupt.               *
 *************************************************************************/
```

```
    CALL RQ$WAIT$INTERRUPT        (interrupt$level$7,
                                   @interrupt$status);
        •
        •     Typical PL/M-286 Statements
        •

/*************************************************************************
 *  When the interrupt task invokes the RESET$INTERRUPT system call,     *
 *  the assignment of the current interrupt handler to interrupt level   *
 *  7 is canceled and, because an interrupt task has also been assigned *
 *  to the level, the interrupt task is deleted.                         *
 *************************************************************************/

    CALL RQ$RESET$INTERRUPT       (interrupt$level$7,
                                   @interrupt$status);
END INTERRUPTTASK;

SAMPLEPROCEDURE:
    PROCEDURE;
    start$address = @INTERRUPTTASK;
                                  /* 1st instruction of interrupt task */
    stack$pointer = NIL;          /* automatic stack allocation  */
    task$flags = 0;               /* indicates no floating-point
                                     instructions */
    data$segment = SELECTOR$OF(NIL);  /* use own data segment */
        •
        •     Typical PL/M-286 Statements
        •
/*************************************************************************
 *  In this example, the SAMPLEPROCEDURE is needed to create the task   *
 *  labeled INTERRUPTTASK.                                               *
 *************************************************************************/

    task$token = RQ$CREATE$TASK   (priority$level$66,
                                   start$address,
                                   data$segment,
                                   stack$pointer,
                                   stack$size$512,
                                   task$flags,
                                   @status);

END SAMPLEPROCEDURE;
```

Condition Codes

E$OK	0000H	No exceptional conditions.
E$CONTEXT	0005H	There is not an interrupt handler assigned to the specified level.
ENOTCONFIGURED	0008H	This system call is not part of the present configuration.
E$PARAM	8004H	The level parameter is invalid.

The SET$INTERRUPT system call assigns an interrupt handler to an interrupt level and, optionally, makes the calling task the interrupt task for the level.

```
CALL RQ$SET$INTERRUPT (level, interrupt$task$flag,
                       interrupt$handler, interrupt$handler$ds,
                       except$ptr);
```

Input Parameters

level — A WORD containing an interrupt level that is encoded as follows (bit 15 is the high-order bit):

Bits	Value
15-7	Reserved bits that should be set to zero
6-4	First digit of the interrupt level (0-7)
3	If one, the level is a master level and bits 6-4 specify the entire level number If zero, the level is a slave level and bits 2-0 specify the second digit
2-0	Second digit of the interrupt level (0-7), if bit 3 is zero

interrupt$task$flag — A BYTE indicating the interrupt task that services the interrupt level. The value of this parameter indicates the number of outstanding SIGNAL$INTERRUPT requests that can exist. When this limit is reached, the associated interrupt level is disabled. The maximum value for this parameter is 255 decimal. The *Extended iRMX II Nucleus User's Guide* describes this feature in more detail.

- If zero, indicates that no interrupt task is to be associated with the special level and that the new interrupt handler will not call SIGNAL$INTERRUPT.

CAUTION

If a task sets the interrupt$task$flag to zero, the designated interrupt handler should not be part of a Human Interface application that is loaded into dynamic memory. If such an application is stopped (via a CONTROL-C entered at a terminal), subsequent interrupts to the vector table entry set by this system call could cause unpredictable results.

- If unequal to zero, indicates that the calling task is to be the interrupt task that will be invoked by the interrupt handler being set. The priority of the calling task is adjusted by the Nucleus according to the interrupt level being serviced. Be certain that priorities set in this manner do not violate the max$priority attribute of the containing job.

interrupt$handler — A POINTER to the first instruction of the interrupt handler.

interrupt$handler$ds — A TOKEN that specifies the interrupt handler's data segment.

- If a valid selector, it contains the base address of the interrupt handler's data segment. See the description of ENTER$INTERRUPT in this manual for information concerning the significance of this parameter.
- if SELECTOR$OF(NIL), the parameter indicates that the interrupt handler will load its own data segment and may not invoke ENTER$INTERRUPT.

  It is often desirable for an interrupt handler to pass information to the interrupt task that it calls. The following PL/M-286 statements, when included in the interrupt task's code (with the first statement listed here being the first statement in the task's code), will extract the DS register value used by the interrupt task and make it available to the interrupt handler, which in turn can access it by calling ENTER$INTERRUPT:

```
DECLARE begin        WORD;   /* A DUMMY VARIABLE */
DECLARE data$ptr     POINTER;
DECLARE data$address STRUCTURE (offset WORD,
                                base TOKEN) AT (@DATA$PTR);
                           /* this makes accessible
                              the two halves of the
                              pointer DATA$PTR */

data$ptr = @begin;  /* puts the whole address of
                        the data segment into
                        data$ptr and
                       data$address */
```

```
ds$base = data$address.base;

    CALL RQ$SET$INTERRUPT (...,ds$base,...);
```

Output Parameter

except$ptr — A POINTER to a WORD to which the iRMX II Operating System will return the condition code generated by this system call.

Description

The SET$INTERRUPT system call is used to inform the Nucleus that the specified interrupt handler is to service interrupts which come in at the specified level. In a call to SET$INTERRUPT, a task must indicate whether the interrupt handler will invoke an interrupt task and whether the interrupt handler has its own data segment. If the handler is to invoke an interrupt task, the call to SET$INTERRUPT also specifies the number of outstanding SIGNAL$INTERRUPT requests that the handler can make before the associated interrupt level is disabled. This number generally corresponds to the number of buffers used by the handler and interrupt task. Refer to the *Extended iRMX II Nucleus User's Guide* for further information.

If there is to be an interrupt task, the calling task is that interrupt task. If there is no interrupt task, SET$INTERRUPT also enables the specified level, which must be disabled at the time of the call.

Example

```
/*************************************************************************
 *  This example illustrates how the SET$INTERRUPT system call can be   *
 *  used.                                                                *
*************************************************************************/

DECLARE TOKEN      LITERALLY 'SELECTOR';

/*  NUCLUS.EXT declares all system calls */
$INCLUDE(/rmx286/inc/NUCLUS.EXT)
INTERRUPTHANDLER: PROCEDURE INTERRUPT EXTERNAL;
END INTERRUPTHANDLER;

    DECLARE interrupt$level$7  LITERALLY '78H';
                               /* specifies master interrupt level 7 */
    DECLARE interrupt$task$flag BYTE;
    DECLARE data$segment       TOKEN;
    DECLARE status             WORD;
```

```
SAMPLEPROCEDURE:
    PROCEDURE;
    interrupt$task$flag = 0;    /* indicates no interrupt task on level 7 */
    data$segment = SELECTOR$OF(NIL); /* indicates that the interrupt
                                        handler will load its own data
                                        segment */
        •
        •     Typical PL/M-286 Statements
        •
/*************************************************************************
 *  An interrupt level must have an interrupt handler or an interrupt    *
 *  task assigned to it.  Invoking the SET$INTERRUPT system call, the    *
 *  calling task assigns INTERRUPTHANDLER to interrupt level 7.          *
*************************************************************************/
    CALL RQ$SET$INTERRUPT   (interrupt$level$7,
                             interrupt$task$flag,
                             @INTERRUPTHANDLER,
                             data$segment,
                             @status);
        •
        •     Typical PL/M-286 Statements
        •
END SAMPLEPROCEDURE;
```

Condition Codes

E$OK	0000H	No exceptional conditions.
EBADADDR	800FH	Either the pointer to the interrupt handler or the selector for the data segment is invalid. Either one of the selectors does not refer to a valid segment, or the offset is outside the segment boundaries.
E$CONTEXT	0005H	One of the following is true: • The task is already an interrupt task. • The specified level already has an interrupt handler assigned to it. • The job containing the calling task or the calling task itself is in the process of being deleted.

ENOTCONFIGURED	0008H	This system call is not part of the present configuration
E$PARAM	8004H	One of the following is true: • The level parameter is invalid or would cause the task to have a priority not allowed by its job. • The programmable interrupt controller (PIC) corresponding to the specified level is not part of the hardware configuration.

The SIGNAL$INTERRUPT system call is used by an interrupt handler to activate an interrupt task.

```
CALL RQ$SIGNAL$INTERRUPT (level, except$ptr);
```

Input Parameter

level — A WORD containing an interrupt level that is encoded as follows (bit 15 is the high-order bit):

Bits	Value
15-7	Reserved bits that should be set to zero.
6-4	First digit of the interrupt level (0-7)
3	If one, the level is a master level and bits 6-4 specify the entire level number If zero, the level is a slave level and bits 2-0 specify the second digit
2-0	Second digit of the interrupt level (0-7), if bit 3 is zero

Output Parameter

except$ptr — A POINTER to a WORD to which the iRMX II Operating System will return the condition code generated by this system call. All exceptional conditions must be processed in-line, as control does not pass to an exceptional handler.

Description

An interrupt handler uses SIGNAL$INTERRUPT to start up its associated interrupt task. The interrupt task runs in its own environment with higher (and possibly the same) level interrupts enabled, whereas the interrupt handler runs in the environment of the interrupted task with all interrupts disabled. The interrupt task can also make use of exception handlers, whereas the interrupt handler always receives exceptions in-line.

Example

```
/*************************************************************************
 *  This example illustrates how the SIGNAL$INTERRUPT system call can   *
 *  be used to activate an interrupt task.                               *
 *************************************************************************/

$include (/RMX286/INC/error.lit)

DECLARE TOKEN      LITERALLY 'SELECTOR';

/*  NUCLUS.EXT declares all system calls */
$INCLUDE(/rmx286/inc/NUCLUS.EXT)

    DECLARE interrupt$level$7     LITERALLY '78H';
                                  /* specifies master interrupt level 7*/
    DECLARE the$first$word        WORD;
    DECLARE interrupt$task$flag   BYTE;
    DECLARE interrupt$handler     POINTER;
    DECLARE data$segment          TOKEN;
    DECLARE status                WORD;
    DECLARE interrupt$status      WORD;
    DECLARE ds$pointer            POINTER;
    DECLARE PTR$OVERLAY           LITERALLY 'STRUCTURE (offset   WORD,
                                                        base     TOKEN)';
                                  /* establishes a structure for
                                     overlays */
    DECLARE ds$pointer$ovly       PTR$OVERLAY AT (@ds$pointer);
                                  /* using the overlay structure, the
                                     base address of the interrupt
                                     handler's data segment is
                                     identified */
INTERRUPTHANDLER: PROCEDURE INTERRUPT PUBLIC;

        •
        •     Typical PL/M-286 Statements
        •

/*************************************************************************
 *  The calling interrupt handler invokes the ENTER$INTERRUPT system    *
 *  call which loads a base address value (defined by                    *
 *  ds$pointer$ovly.base) into the data segment register.  This          *
 *  register provides a mechanism for the interrupt handler to pass      *
 *  data to the interrupt task to be started up by the SIGNAL$INTERRUPT *
 *  system call.                                                         *
 *************************************************************************/

    CALL RQ$ENTER$INTERRUPT       (interrupt$level$7,
                                   @interrupt$status);
    CALL INLINEERRORPROCESS     (interrupt$status);
```

```
        •
        •     Typical PL/M-286 Statements
        •

/*************************************************************************
 *  The interrupt handler uses SIGNAL$INTERRUPT to start up its          *
 *  associated interrupt task.                                           *
 *************************************************************************/
    CALL RQ$SIGNAL$INTERRUPT      (interrupt$level$7,
                                   @interrupt$status);
    CALL INLINEERRORPROCESS     (interrupt$status);

END INTERRUPTHANDLER;

INLINEERRORPROCESS: PROCEDURE(int$status);
    DECLARE int$status  WORD;

    IF int$status <> E$OK THEN
         DO;
             •
             •     Typical PL/M-286 Statements
             •
         END;

END INLINEERRORPROCESS;

SAMPLEPROCEDURE:
    PROCEDURE;
    ds$pointer = @the$first$word; /* a dummy identifier used to point to
                                     interrupt handler's data segment */
    data$segment = ds$pointer$ovly.base;
                                  /* identifies the base address of the
                                     interrupt handler's data segment */
    interrupt$task$flag = 01H;    /* indicates that calling task is to be
                                     interrupt task */
        •
        •     Typical PL/M-286 Statements
        •

/*************************************************************************
 *  By first invoking the SET$INTERRUPT system call, the calling task    *
 *  sets up an interrupt level and becomes the interrupted task for      *
 *  level 7.                                                             *
 *************************************************************************/
    CALL RQ$SET$INTERRUPT         (interrupt$level$7,
                                   interrupt$task$flag,
                                   @INTERRUPTHANDLER,
                                   data$segment,
                                   @status);
```

```
        •
        •     Typical PL/M-286 Statements
        •
END SAMPLEPROCEDURE;
```

Condition Codes

E$OK	0000H	No exceptional conditions.
E$CONTEXT	0005H	No interrupt task is assigned to the specified level.
E$INTERRUPT$OVERFLOW	000AH	The interrupt task has accumulated more than the maximum allowable number of SIGNAL$INTERRUPT requests. It had reached its saturation point and then called ENABLE to allow the handler to receive further interrupt signals. It subsequently received an additional SIGNAL$INTERRUPT request before calling WAIT$INTERRUPT or RQE$TIMED$INTERRUPT.
E$INTERRUPT$SATURATION	0009H	The interrupt task has accumulated the maximum allowable number of SIGNAL$INTERRUPT requests. This is an informative message only. It does not indicate an error.
E$LIMIT	0004H	An overflow has occurred because the interrupt task has received more than 255 SIGNAL$INTERRUPT requests.
ENOTCONFIGURED	0008H	This system call is not part of the present configuration.
E$PARAM	8004H	The level parameter is invalid.

The RQE$TIMED$INTERRUPT system call is used by an interrupt task to signal its readiness to service an interrupt for a certain period of time.

```
CALL RQE$TIMED$INTERRUPT (level, time, except$ptr);
```

Input Parameters

level — A WORD specifying an interrupt level that the task will service. This word is encoded as follows (bit 15 is the high-order bit):

Bits	Value
15-7	Reserved bits that should be set to zero
6-4	First digit of the interrupt level (0-7)
3	If one, the level is a master level and bits 6-4 specify the entire level number If zero, the level is a slave level and bits 2-0 specify the second digit
2-0	Second digit of the interrupt level (0-7), if bit 3 is zero

time — A WORD specifying the number of clock intervals the interrupt task is willing to wait for the interrupt to occur. A value of 0FFFFH means that the task is willing to wait forever.

Output Parameter

except$ptr — A POINTER to a WORD to which the iRMX II Operating System will return the condition code generated by this system call.

Description

The RQE$TIMED$INTERRUPT system call is similar to the WAIT$INTERRUPT system call. Interrupt tasks can invoke it immediately after initializing and immediately after servicing interrupts. Such a call suspends an interrupt task until the interrupt handler for the same level resumes it by invoking SIGNAL$INTERRUPT. Unlike WAIT$INTERRUPT, RQE$TIMED$INTERRUPT permits the interrupt task to limit the time that it will wait. If the time limit expires before an interrupt occurs, the interrupt task is resumed without an interrupt occurring.

While the interrupt task is processing, all lower level interrupts are disabled. The associated interrupt level is either disabled or enabled, depending on the option originally specified with the SET$INTERRUPT system call. If the associated interrupt level is enabled, all SIGNAL$INTERRUPT calls that the handler makes (up to the limit specified with SET$INTERRUPT) are logged. If this count of SIGNAL$INTERRUPT calls is greater than zero when the interrupt task calls RQE$TIMED$INTERRUPT, the task is not suspended. Instead it continues processing the next SIGNAL$INTERRUPT request.

If the associated interrupt level is disabled while the interrupt task is running and the number of outstanding SIGNAL$INTERRUPT requests is less than the user-specified limit, the call to RQE$TIMED$INTERRUPT enables that level.

Example

```
/*************************************************************************
 *  This example illustrates how the RQE$TIMED$INTERRUPT system call can *
 *  be used to signal a task's readiness to service an interrupt.        *
 *************************************************************************/

DECLARE TOKEN      LITERALLY 'SELECTOR';

/*  NUCLUS.EXT declares all system calls */
$INCLUDE(/rmx286/inc/NUCLUS.EXT)

INTERRUPTHANDLER: PROCEDURE INTERRUPT EXTERNAL;
END INTERRUPTHANDLER;

    DECLARE task$token              TOKEN;
    DECLARE priority$level$150      LITERALLY '150';
    DECLARE time                    WORD;
    DECLARE start$address           POINTER;
    DECLARE data$segment            TOKEN;
    DECLARE stack$pointer           POINTER;
    DECLARE stack$size$512          LITERALLY '512'; /* new task's stack
                                                        size is 512 bytes */
    DECLARE task$flags               WORD;
    DECLARE interrupt$level$7        LITERALLY '78H';
                                  /* specifies master interrupt level 7 */
```

```
    DECLARE interrupt$task$flag       BYTE;
    DECLARE interrupt$handler         POINTER;
    DECLARE interrupt$status          WORD;
    DECLARE status                    WORD;

INTERRUPTTASK: PROCEDURE PUBLIC;

    interrupt$task$flag = 01H;    /* indicates that calling task is to
                                     be an interrupt task */
    data$segment = SELECTOR$OF(NIL);  /* use own data segment */

/*************************************************************************
 *  The first system call in this example, SET$INTERRUPT, makes the     *
 *  calling task (INTERRUPTTASK) the interrupt task for interrupt      *
 *  level seven.                                                        *
 *************************************************************************/

    CALL RQ$SET$INTERRUPT         (interrupt$level$7,
                                   interrupt$task$flag,
                                   @INTERRUPTHANDLER,
                                   data$segment,
                                   @interrupt$status);
        •
        •     Typical PL/M-286 Statements
        •

/*************************************************************************
 *  The calling interrupt task invokes RQE$TIMED$INTERRUPT to suspend   *
 *  itself until the interrupt handler for the same level resumes the   *
 *  task by invoking the SIGNAL$INTERRUPT system call.                  *
 *************************************************************************/

    time = 100;                  /* Interrupt task will wait 100 clock
                                    intervals */

    CALL RQE$TIMED$INTERRUPT$    (interrupt$level$7,
                                  time,
                                  @interrupt$status);
        •
        •     Typical PL/M-286 Statements
        •

/*************************************************************************
 *  When the interrupt task invokes the RESET$INTERRUPT system call,    *
 *  the assignment of the current interrupt handler to interrupt level  *
 *  7 is canceled and, because an interrupt task has also been          *
 *  assigned to the line, the interrupt task is deleted.                *
 *************************************************************************/
```

```
    CALL RQ$RESET$INTERRUPT       (interrupt$level$7,
                                   @interrupt$status);
END INTERRUPTTASK;SAMPLEPROCEDURE:
    PROCEDURE;

    start$address = @INTERRUPTTASK;  /* 1st instruction of interrupt
                                        task */
    stack$pointer = NIL;             /* automatic stack allocation */
    task$flags = 0;                  /* designates no floating-point
                                        instructions */
    data$segment = SELECTOR$OF(NIL); /* use own data segment */
        •
        •     Typical PL/M-286 Statements
        •

/*************************************************************************
 *  In this example, the calling task invokes the system call           *
 *  CREATE$TASK to create a task labeled INTERRUPTTASK.                 *
 *************************************************************************/

    task$token = RQ$CREATE$TASK   (priority$level$150,
                                   start$address,
                                   data$segment,
                                   stack$pointer,
                                   stack$size$512,
                                   task$flags,
                                   @status);
        •
        •     Typical PL/M-286 Statements
        •

END SAMPLEPROCEDURE;
```

Condition Codes

E$OK	0000H	No exceptional conditions.
E$CONTEXT	0005H	The calling task is not the interrupt task for the given level.
ENOTCONFIGURED	0008H	This system call is not part of the present configuration.
E$PARAM	8004H	The level parameter is invalid.
E$TIME	0001H	The time limit specified by the interrupt task expired before an interrupt occurred.

The WAIT$INTERRUPT system call is used by an interrupt task to signal its readiness to service an interrupt.

```
CALL RQ$WAIT$INTERRUPT (level, except$ptr);
```

Input Parameter

level — A WORD specifying an interrupt level which is encoded as follows (bit 15 is the high-order bit):

Bits	Value
15-7	Reserved bits that should be set to zero
6-4	First digit of the interrupt level (0-7)
3	If one, the level is a master level and bits 6-4 specify the entire level number If zero, the level is a slave level and bits 2-0 specify the second digit
2-0	Second digit of the interrupt level (0-7), if bit 3 is zero

Output Parameter

except$ptr — A POINTER to a WORD to which the iRMX II Operating System will return the condition code generated by this system call.

Description

The WAIT$INTERRUPT system call is used by interrupt tasks immediately after initializing and immediately after servicing interrupts. Such a call suspends an interrupt task until the interrupt handler for the same level resumes it by invoking SIGNAL$INTERRUPT.

While the interrupt task is processing, all lower level interrupts are disabled. The associated interrupt level is either disabled or enabled, depending on the option originally specified with the SET$INTERRUPT system call. If the associated interrupt level is enabled, all SIGNAL$INTERRUPT calls that the handler makes (up to the limit specified with SET$INTERRUPT) are logged. If this count of SIGNAL$INTERRUPT calls is greater than zero when the interrupt task calls WAIT$INTERRUPT, the task is not suspended. Instead it continues processing the next SIGNAL$INTERRUPT request.

If the associated interrupt level is disabled while the interrupt task is running and the number of outstanding SIGNAL$INTERRUPT requests is less than the user-specified limit, the call to WAIT$INTERRUPT enables that level.

Example

```
/*************************************************************************
 *  This example illustrates how the WAIT$INTERRUPT system call can be  *
 *  used to signal a task's readiness to service an interrupt.          *
 *************************************************************************/

DECLARE TOKEN      LITERALLY 'SELECTOR';

/*  NUCLUS.EXT declares all system calls */
$INCLUDE(/rmx286/inc/NUCLUS.EXT)
INTERRUPTHANDLER: PROCEDURE INTERRUPT EXTERNAL;
END INTERRUPTHANDLER;

    DECLARE task$token              TOKEN;
    DECLARE priority$level$150      LITERALLY '150';
    DECLARE start$address           POINTER;
    DECLARE data$segment            TOKEN;
    DECLARE stack$pointer           POINTER;
    DECLARE stack$size$512          LITERALLY '512'; /* new task's stack
                                                        size is 512 bytes */
    DECLARE task$flags               WORD;
    DECLARE interrupt$level$7        LITERALLY '78H';
                                  /* specifies master interrupt level 7 */
    DECLARE interrupt$task$flag      BYTE;
    DECLARE interrupt$handler        POINTER;
    DECLARE interrupt$status         WORD;
    DECLARE status                   WORD;

INTERRUPTTASK: PROCEDURE PUBLIC;

    interrupt$task$flag = 01H;    /* indicates that calling task is to
                                     be interrupt task */
    data$segment = SELECTOR$OF(NIL);  /* use own data segment */
```

WAIT$INTERRUPT

```
/*************************************************************************
 *  The first system call in this example, SET$INTERRUPT, makes the      *
 *  calling task (INTERRUPTTASK) the interrupt task for interrupt        *
 *  level seven.                                                         *
 *************************************************************************/

    CALL RQ$SET$INTERRUPT          (interrupt$level$7,
                                    interrupt$task$flag,
                                    @INTERRUPTHANDLER,
                                    data$segment,
                                    @interrupt$status);
        •
        •      Typical PL/M-286 Statements
        •
/*************************************************************************
 *  The calling interrupt task invokes WAIT$INTERRUPT to suspend itself  *
 *  until the interrupt handler for the same level resumes the task by   *
 *  invoking the SIGNAL$INTERRUPT system call.                           *
 *************************************************************************/

    CALL RQ$WAIT$INTERRUPT         (interrupt$level$7,
                                    @interrupt$status);
        •
        •      Typical PL/M-286 Statements
        •

/*************************************************************************
 *  When the interrupt task invokes the RESET$INTERRUPT system call,     *
 *  the assignment of the current interrupt handler to interrupt level   *
 *  7 is canceled and, because an interrupt task has also been           *
 *  assigned to the line, the interrupt task is deleted.                 *
 *************************************************************************/

    CALL RQ$RESET$INTERRUPT        (interrupt$level$7,
                                    @interrupt$status);
END INTERRUPTTASK;

SAMPLEPROCEDURE:
    PROCEDURE;
    start$address = @INTERRUPTTASK;  /* 1st instruction of interrupt
                                        task */
    stack$pointer = NIL;            /* automatic stack allocation */
    task$flags = 0;                 /* designates no floating-point
                                       instructions */
    data$segment = SELECTOR$OF(NIL); /* use own data segment */
        •
        •      Typical PL/M-286 Statements
        •
```

```
/*************************************************************************
 *  In this example the calling task invokes the system call           *
 *  CREATE$TASK to create a task labeled INTERRUPTTASK.                *
 *************************************************************************/

   task$token = RQ$CREATE$TASK   (priority$level$150,
                                 start$address,
                                 data$segment,
                                 stack$pointer,
                                 stack$size$512,
                                 task$flags,
                                 @status);
        •
        •     Typical PL/M-286 Statements
        •

END SAMPLEPROCEDURE;
```

Condition Codes

E$OK	0000H	No exceptional conditions.
E$CONTEXT	0005H	The calling task is not the interrupt task for the given level.
ENOTCONFIGURED	0008H	This system call is not part of the present configuration.
E$PARAM	8004H	The level parameter is invalid.

The ALTER$COMPOSITE system call replaces components of composite objects.

CAUTION

Composite objects require the creation of extension objects. Jobs that create extension objects cannot be deleted until all the extension objects are deleted. Therefore you should avoid creating composite objects in Human Interface applications. If a Human Interface application creates extension objects, the application cannot be deleted asynchronously (via a CONTROL-C entered at a terminal.)

```
CALL RQ$ALTER$COMPOSITE (extension, composite, component$index,
                         replacing$obj, except$ptr);
```

Input Parameters

extension	A TOKEN for the extension type object corresponding to the composite object being altered.
composite	A TOKEN for the composite object being altered.
component$index	A WORD whose value specifies the location (starting at location 1) in the component list of the component to be replaced.
replacing$obj	A TOKEN for the replacement component object or zero, which represents no object.

Output Parameter

except$ptr	A POINTER to a WORD to which the iRMX II Operating System will return the condition code generated by this system call.

Description

The ALTER$COMPOSITE system call changes a component of a composite object. Any component in a composite object can be replaced either with a token for another object or with a placeholding SELECTOR$OF(NIL) that represents no object.

The component$index indicates the position of the target token in the list of components.

Example

See the example in section "The GET BYTE Procedure" of the *Extended iRMX II Nucleus User's Guide*.

Condition Codes

E$OK	0000H	No exceptional conditions.
E$CONTEXT	0005H	The composite parameter is not compatible with the extension parameter.
E$EXIST	0006H	The extension, composite, or object parameter(s) is not a token for an existing object.
ENOTCONFIGURED	0008H	This system call is not part of the present configuration.
E$TYPE	8002H	One or both of the extension or composite parameters is a token for an object that is not of the correct object type.
E$PARAM	8004H	The component$index parameter refers to a nonexistent position in the component object list.

The CREATE$COMPOSITE system call creates a composite object.

CAUTION

Composite objects require the creation of extension objects. Jobs that create extension objects cannot be deleted until all the extension objects are deleted. Therefore you should avoid creating composite objects in Human Interface applications. If a Human Interface application creates extension objects, the application cannot be deleted asynchronously (via a CONTROL-C entered at a terminal.)

```
composite=RQ$CREATE$COMPOSITE (extension, token$list, except$ptr);
```

Input Parameters

extension — A TOKEN for an extension type representing a license to create a composite object.

token$list — A POINTER to a structure of the form:

```
DECLARE
    token$list   STRUCTURE(
    num$slots     WORD,
    num$used      WORD,
    tokens(*)     TOKEN);
```

where:

num$slots — Number of slots for component objects that the composite object will contain. This number represents the maximum number of component objects that the composite object can handle. If num$slots is greater than num$used, the values in the extra slots are set to SELECTOR$OF(NIL).

num$used — Number of token elements to include in the composite. If num$used is greater than num$slots, the extra components are ignored.

token(*) — Tokens that will actually constitute the composite object.

Output Parameters

composite	A TOKEN to which the Operating System returns the new composite token.
except$ptr	A POINTER to a WORD to which the iRMX II Operating System will return the condition code generated by this system call.

Description

The CREATE$COMPOSITE system call creates a composite object of the specified extension type. It accepts a list of tokens that specify the component objects and returns a token for the new composite object. The token$list parameter points to a structure that contains the list of tokens.

The first element in the token list (num$slots) indicates the number of slots available in the composite object; that is the maximum number of component objects that can be part of the composite. Because you might not fill all the slots when you create the composite object, the second element (num$used) indicates the number of tokens that should be included in the composite. These tokens follow num$used in the structure. CREATE$COMPOSITE selects tokens to include beginning with the first token in the token list.

If the number of token elements to include in the composite (num$used) is less than the number of component slots (num$slots), CREATE$COMPOSITE fills the remaining slots with the value SELECTOR$OF(NIL).

If, on the other hand, the number of component slots (num$slots) is less than the number of token elements to include in the composite (num$used), CREATE$COMPOSITE ignores the remaining tokens in the token list.

Example

See "CREATE_RING_BUFFER Procedure" in the *Extended iRMX II Nucleus User's Guide*.

Condition Codes

E$OK	0000H	No exceptional conditions.
EBADADDR	800FH	The token$list pointer is invalid. Either the selector does not refer to a valid segment, or the offset is outside the segment boundaries.

E$EXIST	0006H	The extension parameter or one or more of the non-zero token$list parameters is not a token for an existing object.
E$LIMIT	0004H	The calling task's job has already reached its object limit.
E$MEM	0002H	The memory available to the calling task's job is insufficient to create a composite.
ENOTCONFIGURED	0008H	This system call is not part of the present configuration.
E$PARAM	8004H	The specified number of components is zero.
E$SLOT	000CH	There is no room in the GDT for the composite object's descriptor.
E$TYPE	8002H	The extension parameter is a token for an object that is not an extension object.

The DELETE$COMPOSITE system call deletes a composite object.

CAUTION

Composite objects require the creation of extension objects. Jobs that create extension objects cannot be deleted until all the extension objects are deleted. Therefore you should avoid creating composite objects in Human Interface applications. If a Human Interface application creates extension objects, the application cannot be deleted asynchronously (via a CONTROL-C entered at a terminal).

```
CALL RQ$DELETE$COMPOSITE (extension, composite, except$ptr);
```

Input Parameters

extension	A TOKEN for the extension type used as a license to create the composite object to be deleted.
composite	A TOKEN for the composite object to be deleted.

Output Parameter

except$ptr	A POINTER to a WORD to which the iRMX II Operating System will return the condition code generated by this system call.

Description

The DELETE$COMPOSITE system call deletes the specified composite object, but not its component objects.

Example

See the example in section "The Initialization Part" of the *Extended iRMX II Nucleus User's Guide*.

DELETE$COMPOSITE

Condition Codes

E$OK	0000H	No exceptional conditions.
E$CONTEXT	0005H	The extension type does not match the composite parameter.
E$EXIST	0006H	One or both of the extension or composite parameters is not a token for an existing object.
ENOTCONFIGURED	0008H	This system call is not part of the present configuration.
E$TYPE	8002H	One or both of the extension or composite parameters is a token for an object that is not of the correct type.

The INSPECT$COMPOSITE system call returns a list of the component tokens contained in a composite object.

CAUTION

Composite objects require the creation of extension objects. Jobs that create extension objects cannot be deleted until all the extension objects are deleted. Therefore you should avoid creating composite objects in Human Interface applications. If a Human Interface application creates extension objects, the application cannot be deleted asynchronously (via a CONTROL-C entered at a terminal).

```
CALL RQ$INSPECT$COMPOSITE (extension, composite, token$list$ptr,
                           except$ptr);
```

Input Parameters

extension	A TOKEN for the extension object corresponding to the composite object being inspected.
composite	A TOKEN for the composite object being inspected.

Output Parameters

token$list — A POINTER to a structure of the form:

```
DECLARE
    token$list$ptr   STRUCTURE(
                     num$slots    WORD,
                     num$used     WORD,
                     tokens(*)    TOKEN);
```

The system call returns information in the fields of this structure, as follows:

num$slots — Number of positions available for tokens in token$list (an upper limit on the number of tokens to be returned). You fill in this field to tell the system call how many tokens to return.

num$used — Number of component tokens making up the composite object.

	token(*)	The tokens that actually constitute the composite object.
except$ptr		A POINTER to a WORD to which the iRMX II Operating System will return the condition code generated by this system call.

Description

The INSPECT$COMPOSITE system call accepts a token for a composite object and returns a list of tokens for the components of the composite object.

The calling task must supply the num$slots value in the data structure pointed to by the token$list parameter. The Nucleus fills in the remaining fields in that structure. If num$slots is set to zero, the Nucleus will fill in only the num$used field.

If the num$slots value is smaller than the actual number of component tokens, only that number (num$slots) of tokens will be returned.

Example

See the "DELETE_RING_BUFFER Procedure" example in the *Extended iRMX II Nucleus User's Guide*.

Condition Codes

E$OK	0000H	No exceptional conditions.
EBADADDR	800FH	The pointer to the token$list structure is invalid. Either the selector does not refer to a valid segment, or the offset is outside the segment boundaries.
E$CONTEXT	0005H	The composite parameter is not compatible with the extension parameter.
E$EXIST	0006H	The composite and/or extension parameter(s) is not a token for an existing object.
ENOTCONFIGURED	0008H	This system call is not part of the present configuration.
E$TYPE	8002H	One or both of the extension or composite parameters is a token for an object that is not of the correct type.

The CREATE$EXTENSION system call creates a new object type.

CAUTION

Jobs that create extension objects cannot be deleted until the extension object is deleted. Therefore, you should avoid creating extension objects in Human Interface applications. If a Human Interface application creates extension objects, the application cannot be deleted asynchronously (via a CONTROL-C entered at a terminal.)

```
extension=RQ$CREATE$EXTENSION (type$code, deletion$mailbox,
                               except$ptr);
```

Input Parameters

type$code	A WORD containing the type code for the new type. The type code for the new type can be any value from 8000H to 0FFFFH and must not be currently in use. (The type codes 0 through 7FFFH are reserved for Intel products.)
deletion$mailbox	A TOKEN for the mailbox where objects of the new type are sent whenever the extension type or their containing job is deleted. A SELECTOR$OF(NIL) value indicates no deletion mailbox is desired.

Output Parameters

extension	A TOKEN to which the Operating System will return a token for the new type.
except$ptr	A POINTER to a WORD to which the iRMX II Operating System will return the condition code generated by this system call.

Description

The CREATE$EXTENSION system call returns a token for the newly created extension object type.

You can specify a deletion mailbox when the extension type is created. If you do, a task in your type manager for the new type must wait at the deletion mailbox for tokens of objects of the new extension type that are to be deleted. Tokens of objects are sent to the deletion mailbox for deletion either when their extension type is deleted or when their containing job is deleted; they are not sent there when being deleted by DELETE$COMPOSITE. The task servicing the deletion mailbox may do anything with the composite objects sent to it, but it must delete them.

If you do not want to specify a deletion mailbox, set the token value for deletion$mailbox to SELECTOR$OF(NIL). If the extension type has no deletion mailbox, composite objects of that type are deleted automatically, and the type manager is not informed. The advantage of having a deletion mailbox is that the type manager has the opportunity to do more than merely delete the composite objects.

A job containing a task that creates an extension object cannot be deleted until the extension object is deleted.

Example

See the example in section "The Initialization Part" of the *Extended iRMX II Nucleus User's Guide*.

Condition Codes

E$OK	0000H	No exceptional conditions.
E$CONTEXT	0005H	The calling task's job is being deleted.
E$EXIST	0006H	The deletion$mailbox parameter is not a token for an existing object.
E$LIMIT	0004H	The calling task's job has reached its object limit.
E$MEM	0002H	The memory available to the calling task's job is not sufficient to create an extension.
ENOTCONFIGURED	0008H	This system call is not part of the present configuration.
E$SLOT	000CH	There is no room in the GDT for the extension's descriptor.

E$PARAM	8004H	The type$code parameter is invalid.
E$TYPE	8002H	The deletion$mailbox parameter is a token for an object that is not a mailbox.

The DELETE$EXTENSION system call deletes an extension object and all composites of that type.

CAUTION

Jobs that create extension objects cannot be deleted until the extension object is deleted. Therefore, you should avoid creating extension objects in Human Interface applications. If a Human Interface application creates extension objects, the application cannot be deleted asynchronously (via a CONTROL-C entered at a terminal).

```
CALL RQ$DELETE$EXTENSION (extension, except$ptr);
```

Input Parameter

extension	A TOKEN for the extension object to be deleted.

Output Parameter

except$ptr	A POINTER to a WORD to which the iRMX II Operating System will return the condition code generated by this system call.

Description

The DELETE$EXTENSION system call deletes the specified extension object and all composite objects of that type, making the corresponding type code available for reuse.

If you specified a deletion mailbox when you created the extension, all the composite objects created subsequently with that extension type are sent to the deletion mailbox. You must delete all the composite objects sent to the deletion mailbox. The DELETE$EXTENSION system call is not completed until all of the composite objects have been deleted.

If an extension has no deletion mailbox, composite objects created by the CREATE$EXTENSION system call are deleted without informing the type manager.

The job containing the task that created the extension object cannot be deleted until the extension object is deleted.

Example

```
/*************************************************************************
 * This example illustrates how the DELETE$EXTENSION system call     *
 * can  be used.                                                     *

*************************************************************************/

DECLARE TOKEN       LITERALLY 'SELECTOR';

/*  NUCLUS.EXT declares all system calls */
$INCLUDE(/rmx286/inc/NUCLUS.EXT)

    DECLARE ext$token          TOKEN;
    DECLARE type$code          WORD;
    DECLARE delete$mbx$token   TOKEN;
    DECLARE status             WORD;

SAMPLEPROCEDURE:
    PROCEDURE;

    type$code = 08000h;                 /*  this is a valid value for a
                                            new type  */

    delete$mbx$token = SELECTOR$OF(NIL); /*  No deletion mailbox is
                                            desired for this new type */

/*************************************************************************
 *  In order to delete an extension, a task must know the token for *
 *  that extension.  In this example, the needed token is known     *
 *  because the calling task creates the extension.                 *

*************************************************************************/

    ext$token = RQ$CREATE$EXTENSION  (type$code,
                                      delete$mbx$token,
                                      @status);
        •
        •     Typical PL/M-286 Statements
        •

/*************************************************************************
 *  When the extension is no longer needed, it may be deleted by    *
 *   any task that knows the token for the extension.               *

*************************************************************************/

    CALL RQ$DELETE$EXTENSION  (ext$token, @status);


END SAMPLEPROCEDURE;
```

Condition Codes

E$OK	0000H	No exceptional conditions.
E$EXIST	0006H	The extension parameter is not a token for an existing object.
E$MEM	0002H	The memory available to the calling task's job is not sufficient to complete this operation.
ENOTCONFIGURED	0008H	This system call is not part of the present configuration.
E$TYPE	8002H	The extension parameter is a token for an object that is not an extension object.

The DISABLE$DELETION system call makes an object immune to ordinary deletion.

CAUTION

DISABLE$DELETION makes an object immune to ordinary deletion by increasing the disabling depth of an object. If a Human Interface application contains objects whose disabling depths are greater than one, the application cannot be deleted asynchronously (via a CONTROL-C entered at a terminal). Therefore you should not use DISABLE$DELETION (and have no need to use ENABLE$DELETION or FORCE$DELETE) in Human Interface applications.

```
CALL RQ$DISABLE$DELETION (object, except$ptr);
```

Input Parameter

object	A TOKEN for the object whose deletion is to be disabled.

Output Parameter

except$ptr	A POINTER to a WORD to which the Operating System will return the condition code generated by this system call.

Description

The DISABLE$DELETION system call increases by one the disabling depth of an object, making it immune to ordinary deletion. If an object's disabling depth is two or greater, it is also immune to forced deletion. If a task attempts to delete the object while it is immune, the task sleeps until the immunity is removed. At that time, the object is deleted and the task is awakened.

The ENABLE$DELETION system call is used to decrease the disabling depth of an object, making it susceptible to ordinary deletion.

NOTES

If an object within a job has had its deletion disabled, then the containing job cannot be deleted until that object has had its deletion re-enabled.

Disabling deletion of a suspended task causes the calling task to hang until the suspended task is resumed.

An attempt to raise an object's disabling depth above 255 causes an E$LIMIT exceptional condition.

Example

```
/*************************************************************************
 *  This example illustrates how the DISABLE$DELETION system call can   *
 *  be used to make an object immune to ordinary deletion.              *
 *************************************************************************/

DECLARE TOKEN      LITERALLY 'SELECTOR';

/*  NUCLUS.EXT declares all system calls */
$INCLUDE(/rmx286/inc/NUCLUS.EXT)

    DECLARE task$token            TOKEN;
    DECLARE calling$task          LITERALLY '0';
    DECLARE status                WORD;

SAMPLEPROCEDURE:
    PROCEDURE;

    •
    •     Typical PL/M-286 Statements
    •

/*************************************************************************
 *  In this example the calling task will be the object to become       *
 *  immune to ordinary deletion.  GET$TASK$TOKEN is invoked by the      *
 *  calling task to obtain its own token.                               *
 *************************************************************************/

    task$token = RQ$GET$TASK$TOKENS  (calling$task,
                                      @status);

    •
    •     Typical PL/M-286 Statements
    •
```

```
/*************************************************************************
 *  Using its own token, the calling task invokes the DISABLE$DELETION  *
 *  system call to increase its own disabling depth by one.  This makes *
 *  the calling task immune to ordinary deletion.                       *
 *************************************************************************/

   CALL RQ$DISABLE$DELETION      (task$token, @status);
   •
   •     Typical PL/M-286 Statements
   •

END SAMPLEPROCEDURE;
```

Condition Codes

E$OK	0000H	No exceptional conditions.
E$EXIST	0006H	The object parameter is not a token for an existing object.
E$LIMIT	0004H	The object's disabling depth is already 255.
ENOTCONFIGURED	0008H	This system call is not part of the present configuration.

The ENABLE$DELETION system call enables the deletion of objects that have had deletion disabled.

CAUTION

DISABLE$DELETION makes an object immune to ordinary deletion by increasing the disabling depth of an object. If a Human Interface application contains objects whose disabling depths are greater than one, the application cannot be deleted asynchronously (via a CONTROL-C entered at a terminal). Therefore you should not use DISABLE$DELETION (and have no need to use ENABLE$DELETION or FORCE$DELETE) in Human Interface applications.

```
CALL RQ$ENABLE$DELETION (object, except$ptr);
```

Input Parameter

object	A TOKEN for the object whose deletion is to be enabled.

Output Parameter

except$ptr	A POINTER to a WORD to which the Operating System will return the condition code generated by this system call.

Description

The ENABLE$DELETION system call decreases by one the disabling depth of an object. If there is a pending deletion request against the object, and the ENABLE$DELETION call makes the object eligible for deletion, the object is deleted and the task which made the deletion request is awakened.

Example

```
/*************************************************************************
 *  This example illustrates how the ENABLE$DELETION system call can be *
 *  used to enable the deletion of a task that had been deletion        *
 *  disabled.                                                           *
 *************************************************************************/

DECLARE TOKEN       LITERALLY 'SELECTOR';
```

```
/*  NUCLUS.EXT declares all system calls */
$INCLUDE(/rmx286/inc/NUCLUS.EXT)

    DECLARE task$token          TOKEN;
    DECLARE calling$task        LITERALLY 'SELECTOR$OF(NIL)';
    DECLARE status              WORD;

SAMPLEPROCEDURE:
    PROCEDURE;

        •
        •     Typical PL/M-286 Statements
        •

/*************************************************************************
 *  In this example the calling task will be the object to become       *
 *  immune to deletion.  The GET$TASK$TOKEN is invoked by the calling   *
 *  task to obtain its own token.                                       *
 *************************************************************************/

    task$token = RQ$GET$TASK$TOKENS     (calling$task, @status);
        •
        •     Typical PL/M-286 Statements
        •

/*************************************************************************
 *  Using its own token, the calling task invokes the DISABLE$DELETION  *
 *  system call to increase its own disabling depth by one.  This makes *
 *  the calling task immune to ordinary deletion.                        *
 *************************************************************************/

    CALL RQ$DISABLE$DELETION       (task$token, @status);
        •
        •     Typical PL/M-286 Statements
        •

/*************************************************************************
 *  In order to allow itself to be deleted, the calling task invokes    *
 *  the ENABLE$DELETION system call.  This system call decreases by one *
 *  the disabling depth of an object.  In this example, the object is   *
 *  the calling task.                                                   *
 *************************************************************************/


    CALL RQ$ENABLE$DELETION            (task$token, @status);
        •
        •     Typical PL/M-286 Statements
        •

END SAMPLEPROCEDURE;
```

Condition Codes

E$OK	0000H	No exceptional conditions.
E$CONTEXT	0005H	The object's deletion is not disabled.
E$EXIST	0006H	The object parameter is not a token for an existing object.
ENOTCONFIGURED	0008H	This system call is not part of the present configuration.

The FORCE$DELETE system call deletes objects whose disabling depths are zero or one.

CAUTION

DISABLE$DELETION makes an object immune to ordinary deletion by increasing the disabling depth of an object. If a Human Interface application contains objects whose disabling depths are greater than one, the application cannot be deleted asynchronously (via a CONTROL-C entered at a terminal). Therefore you should not use DISABLE$DELETION (and have no need to use ENABLE$DELETION or FORCE$DELETE) in Human Interface applications.

```
CALL RQ$FORCE$DELETE (extension, object, except$ptr);
```

Input Parameters

extension	If the object to be deleted is a composite object, this parameter is a TOKEN for the extension type associated with the composite object to be deleted. Otherwise, the extension parameter is ignored.
object	A TOKEN for the object that is to be deleted.

Output Parameter

except$ptr	A POINTER to a WORD to which the Operating System will return the condition code generated by this system call.

Description

The FORCE$DELETE system call deletes objects whose disabling depths are zero or one. If an object has a deletion depth of two or more, the calling task is put to sleep until the deletion depth is decreased to one. At that time, the object is deleted and the task is awakened. If the wrong extension parameter is specified when deleting a composite, FORCE$DELETE issues an E$CONTEXT error and returns without deleting the composite. If the object to be force deleted is not a composite, the extension parameter is ignored.

Example

```
/*************************************************************************
 *  This example illustrates how the FORCE$DELETE system call can be     *
 *  used to force the deletion of a task that has had deletion           *
 *  disabled.                                                            *
 *************************************************************************/

DECLARE TOKEN      LITERALLY 'SELECTOR';

/*  NUCLUS.EXT declares all system calls */
$INCLUDE(/rmx286/inc/NUCLUS.EXT)

    DECLARE sem$token             TOKEN;
    DECLARE ext$token             TOKEN;
    DECLARE init$value            WORD;
    DECLARE max$value             WORD;
    DECLARE sem$flags             WORD;
    DECLARE status                WORD;

SAMPLEPROCEDURE:
    PROCEDURE;

    init$value = 1;       /* the new semaphore has one initial unit */

    max$value = 10h;      /* the new semaphore can have a maximum of
                             16 units  */

    sem$flags = 0;        /* designates a first-in/first-out task queue */

/*************************************************************************
 *  In this example, the calling task creates the object to become       *
 *  immune to deletion.  The CREATE$SEMAPHORE is invoked by the calling  *
 *  task to create a semaphore                                           *
 *************************************************************************/

    sem$token = RQ$CREATE$SEMAPHORE    (init$value,
                                        max$value,
                                        sem$flags,
                                        @status);
        •
        •     Typical PL/M-286 Statements
        •

/*************************************************************************
 *  Using the semaphore token, the calling task invokes the              *
 *  DISABLE$DELETION system call to increase the disabling depth by one.*
 *  This makes the semaphore immune to ordinary deletion.                *
 *************************************************************************/

    CALL RQ$DISABLE$DELETION      (sem$token, @status);
```

```
/*************************************************************************
 *  In order to delete the semaphore, the calling task invokes           *
 *  the FORCE$DELETE system call.  This system call deletes the semaphore*
 *  even though the disabling depth of the semaphore is one.             *
 *************************************************************************/

   ext$token = SELECTOR$OF(NIL);  /* there is no extension object, so
                                     set the extension parameter to 0 */

   CALL RQ$FORCE$DELETE           (ext$token,
                                   sem$token,
                                   @status);
       •
       •     Typical PL/M-286 Statements
       •

END SAMPLEPROCEDURE;
```

Condition Codes

E$OK	0000H	No exceptional conditions.
E$CONTEXT	0005H	The wrong extension type was used in the extension parameter of the FORCE$DELETE system call.
E$EXIST	0006H	One or both of the object or extension parameters is not a token for an existing object.
ENOTCONFIGURED	0008H	This system call is not part of the present configuration.
E$TYPE	8002H	The extension parameter is a token for an object that is not an extension object.

The RQESETOS$EXTENSION system call dynamically associates an entry point of a user-written OS extension with a call gate. It can also clear that association.

```
CALL RQE$SET$OS$EXTENSION (gate$number, start$address, except$ptr);
```

Input Parameters

gate$number — A WORD specifying the number of the call gate to be associated with the OS extension. This number lists the entry number of that call gate in the GDT. For example, if the designated call gate is the third descriptor listed in the GDT, use a value of 3 for this parameter. The call gate you indicate must have been reserved for this purpose during system configuration. This system call cannot establish new call gates.

start$address — A POINTER to the first instruction of the OS extension. Setting this parameter to NIL disables the OS extension previously associated with the call gate.

Output Parameter

except$ptr — A POINTER to a WORD to which the iRMX II Operating System will return the condition code generated by this system call.

Description

This function request is used to set up OS extensions so that tasks can invoke them just as they would any other system call. This process involves forming an association between the OS extension and a call gate.

To form this association, a call gate must already be created specifically for use with your OS extensions. You must set up this call gate during system configuration by using the ICU. You can also form the association between call gate and OS extension during configuration. If you do that, you do not need to invoke this system call. Refer to the *Extended iRMX II Interactive Configuration Utility Reference Manual* for more information about configuring OS extensions.

The RQESETEXTENSION system call can also be used to terminate the association between the call gate and a particular OS extension. If you plan to use the same call gate for multiple OS extensions, you must terminate the association with one OS extension before establishing an association with another. If a task attempts to invoke an OS extension that has been disabled in this manner, a null operation occurs.

Example

```
/*************************************************************************
 *  This example illustrates how the RQE$SET$OS$EXTENSION system call   *
 *  sets the call gate used by an OS extension.  The example assumes    *
 *  the gate number was reserved during configuration.                  *
 *************************************************************************/

DECLARE TOKEN      LITERALLY 'SELECTOR';

/*  NUCLUS.EXT declares all system calls */
$INCLUDE(/rmx286/inc/NUCLUS.EXT)

    DECLARE gate$440              LITERALLY '440';
    DECLARE status                WORD;

ENTRY$440:  PROCEDURE EXTERNAL;
    END ENTRY$440;

SAMPLEPROCEDURE:
    PROCEDURE;
        •
        •     Typical PL/M-286 Statements
        •

/*************************************************************************
 *  The calling task invokes the RQE$SET$OS$EXTENSION system call to    *
 *  set the call gate at entry 440 in the GDT.  The entry point address *
 *  is also specified.                                                  *
 *************************************************************************/

    CALL RQE$SET$OS$EXTENSION          (gate$440, @ENTRY$440, @status);
        •
        •   Typical PL/M-286 Statements
        •

END SAMPLEPROCEDURE;
```

Condition Codes

E$OK	0000H	No exceptional conditions.
EBADADDR	800FH	The pointer to the start address is invalid. Either the selector doesn't refer to a valid segment, or the offset is outside the segment boundaries.
E$CONTEXT	0005H	The specified call gate is already associated with an OS extension. Before you can set the call gate again, you must first reset it (call RQESETOS$EXTENSION and specify NIL for the start$address parameter).
ENOTCONFIGURED	0008H	This system call is not part of the present configuration.
E$PARAM	8004H	The gate$number parameter does not specify a valid call gate.
E$TYPE	8002H	The specified call gate is already in use.

The SIGNAL$EXCEPTION system call is normally used with OS extensions to signal the occurrence of an exceptional condition.

```
CALL RQ$SIGNAL$EXCEPTION (exception$code, param$num, stack$ptr,
                          first$reserved$word, second$reserved$word,
                          except$ptr);
```

Input Parameters

exception$code — A WORD containing the code (see list in the *Extended iRMX II Nucleus User's Guide*) for the exceptional condition detected.

param$num — A BYTE containing the number of the parameter that caused the exceptional condition. If no parameter is at fault, param$num equals zero.

stack$ptr — A WORD that, if not zero, must contain the value of the stack pointer saved on entry to the operating system extension (see the entry procedure in the *Extended iRMX II Nucleus User's Guide* for an example). The top five words in the stack (where BP is at the top of the stack) must be as follows:

FLAGS	Saved by software interrupt
CS	to OS extension
IP	
DS	Saved by OS extension
BP	on entry

Upon completion of SIGNAL$EXCEPTION, control is returned to either of two instructions. If stack$pointer contains NIL, control returns to the instruction following the call to SIGNAL$EXCEPTION. Otherwise, control returns to the instruction identified in CS and IP.

first$reserved$word — A WORD reserved for Intel use. Set this parameter to zero.

second$reserved$word — A WORD reserved for Intel use. Set this parameter to zero.

Output Parameter

except$ptr — A POINTER to a WORD to which the Operating System will return the condition code generated by this system call.

SIGNAL$EXCEPTION

Description

Operating system extensions use the SIGNAL$EXCEPTION system call to signal the occurrence of exceptional conditions. Depending on the exceptional condition and the calling task's exception mode, control may or may not pass directly to the task's exception handler.

If the exception handler does not get control, the exceptional condition code is returned to the calling task. The task can then access the code by checking the contents of the word pointed to by the except$ptr parameter for its call (not for the call to SIGNAL$EXCEPTION).

Example

```
/*************************************************************************
 *  This example illustrates how the SIGNAL$EXCEPTION system call can   *
 *  be used to signal the occurrence of the exceptional condition       *
 *  E$CONTEXT.                                                          *
 *************************************************************************/

DECLARE TOKEN       LITERALLY 'SELECTOR';

/*  NUCLUS.EXT declares all system calls */
$INCLUDE(/rmx286/inc/NUCLUS.EXT)

    DECLARE e$context             LITERALLY '5H';
    DECLARE param$num             BYTE;
    DECLARE stack$pointer         WORD;
    DECLARE reserved$word         LITERALLY '0';
    DECLARE status                WORD;

SAMPLEPROCEDURE:
    PROCEDURE;
    param$num = 0;                /* no parameter at fault */
    stack$pointer = NIL;          /* return control to instruction
                                     following call */
        •
        •    Typical PL/M-286 Statements
        •

/*************************************************************************
 *  In this example the SIGNAL$EXCEPTION system call is invoked by      *
 *  extensions of the Operating System to signal the occurrence of an   *
 *  E$CONTEXT exceptional condition.                                    *
 *************************************************************************/
```

```
    CALL RQ$SIGNAL$EXCEPTION      (e$context,
                                  param$num,
                                  stack$pointer,
                                  reserved$word,
                                  reserved$word,
                                  @status);
        •
        •   Typical PL/M-286 Statements
        •

END SAMPLEPROCEDURE;
```

Condition Codes

E$OK	0000H	No exceptional conditions.

The ACCEPT$CONTROL system call requests immediate access to data protected by a region.

CAUTION

Tasks that use regions cannot be deleted while they access data protected by the region. Therefore, you should avoid using regions in Human Interface applications. If a task in a Human Interface application uses regions, the application cannot be deleted asynchronously (via a CONTROL-C entered at a terminal) while the task is in the region.

```
CALL RQ$ACCEPT$CONTROL (region, except$ptr);
```

Input Parameter

region	A TOKEN for the target region.

Output Parameter

except$ptr	A POINTER to a WORD to which the iRMX II Operating System will return the condition code generated by this system call.

Description

The ACCEPT$CONTROL system call provides access to data protected by a region if access is immediately available. If access is not immediately available, the E$BUSY condition code is returned and the calling task remains ready.

Once a task has gained control of a region, it should not suspend or delete itself while in control of the region. Doing so will lock the region and prevent other tasks from gaining access.

Example

```
/*****************************************************************
* This example illustrates how the ACCEPT$CONTROL system call    *
*  can be  used to access data protected by a region.            *
*****************************************************************/
```

```
DECLARE TOKEN                       LITERALLY 'SELECTOR';
/* NUCLUS.EXT declares all system calls */
$INCLUDE(/rmx286/inc/NUCLUS.EXT)
    DECLARE region$token            TOKEN;
    DECLARE priority$queue          LITERALLY '1'; /* tasks wait in
                                                      priority order */
    DECLARE status                  WORD;
SAMPLEPROCEDURE:
    PROCEDURE;
    •
    •   Typical PL/M-286 Statements
    •
/*************************************************************************
 *  In order to access the data within a region, a task must know the   *
 *  token for that region.  In this example, the needed token is known  *
 *  because the calling task creates the region.                        *
 *************************************************************************/
    region$token = RQ$CREATE$REGION  (priority$queue,
                                      @status);
    •
    •   Typical PL/M-286 Statements
    •
/*****************************************************************
 *  At some point in the task, access is needed to the data      *
 * protected by the region.  The calling task then invokes the   *
 * ACCEPT$CONTROL system call and obtains access to the data     *
 *  if access is immediately available.                          *
*****************************************************************/
    CALL RQ$ACCEPT$CONTROL           (region$token,
                                      @status);

    •
    •   Typical PL/M-286 Statements
    •

/*****************************************************************
 *  When the task is ready to relinquish access to the data      *
 *  protected by the region, it invokes the SEND$CONTROL         *
 *  system call.                                                 *
*****************************************************************/
    CALL RQ$SEND$CONTROL             (@status);
    •
    •     Typical PL/M-286 Statements
    •

END SAMPLEPROCEDURE;
```

Condition Codes

E$OK	0000H	No exceptional conditions.
E$BUSY	0003H	Another task currently has access to the protected data.
E$CONTEXT	0005H	The calling task currently has access to the region in question.
E$EXIST	0006H	The region parameter is not a token for an existing object.
ENOTCONFIGURED	0008H	This system call is not part of the present configuration.
E$TYPE	8002H	The region parameter is a token for an object that is not a region.

The CREATE$REGION system call creates a region.

CAUTION

Tasks that use regions cannot be deleted while they are in control of the region. Using regions in a Human Interface application task can cause situations where the application cannot be deleted asynchronously (via a CONTROL-C entered at a terminal) while the task is in the region. Therefore, you should avoid using regions in Human Interface applications.

```
region = RQ$CREATE$REGION (region$flags, except$ptr);
```

Input Parameters

region$flags	A WORD that specifies the queuing protocol of the new region. If the low-order bit equals zero, tasks await access in FIFO order. If the low-order bit equals one, tasks await access in priority order. The other bits in the WORD are reserved and should be set to zero.

Output Parameters

region	A TOKEN to which the Operating System will return a token for the new region.
except$ptr	A POINTER to a WORD to which the iRMX II Operating System will return the condition code generated by this system call.

Description

The CREATE$REGION system call creates a region and returns a token for the region.

Example

```
/*****************************************************************
 * This example illustrates how the CREATE$REGION system call    *
 * can be  used.                                                 *
 *****************************************************************/
```

CREATE$REGION

```
DECLARE TOKEN       LITERALLY 'SELECTOR';

/* NUCLUS.EXT declares all system calls */
$INCLUDE(/rmx286/inc/NUCLUS.EXT)

    DECLARE region$token    TOKEN;
    DECLARE priority$queue  LITERALLY '1';
                            /* tasks wait in priority order */
    DECLARE status          WORD;

SAMPLEPROCEDURE:
    PROCEDURE;

        •
        •     Typical PL/M-286 Statements
        •

/*****************************************************************
 *  The token region$token is returned when the calling task     *
 *  invokes the CREATE$REGION system call.                       *
 *****************************************************************/

    region$token = RQ$CREATE$REGION  (priority$queue,
                                      @status);
        •
        •     Typical PL/M-286 Statements
        •

END SAMPLEPROCEDURE;
```

Condition Codes

E$OK	0000H	No exceptional conditions.
E$LIMIT	0004H	The calling task's job has reached its object limit.
E$MEM	0002H	The memory pool of the calling task's job does not contain a sufficiently large block to satisfy the request.
ENOTCONFIGURED	0008H	This system call is not part of the present configuration.
E$SLOT	000CH	There isn't enough room in the GDT for another descriptor.

The DELETE$REGION system call deletes a region.

CAUTION

Tasks which use regions cannot be deleted while they access data protected by the region. Therefore, you should avoid using regions in Human Interface applications. If a task in a Human Interface application uses regions, the application cannot be deleted asynchronously (via a CONTROL-C entered at a terminal) while the task is in the region.

```
CALL RQ$DELETE$REGION (region, except$ptr);
```

Input Parameter

region — A TOKEN for the region to be deleted.

Output Parameter

except$ptr — A POINTER to a WORD to which the iRMX II Operating System will return the condition code generated by this system call.

Description

The DELETE$REGION system call deletes a region. If a task that has access to data protected by the region requests that the region be deleted, the task receives an E$CONTEXT exceptional condition. If a task requests deletion while another task has access, deletion is delayed until access is surrendered. If two more more tasks request deletion of a region that another task has access to, a deadlock results. A deadlock also results when a task attempts to delete another task that is in the process of trying to delete an occupied region. When the region is deleted, any waiting tasks awaken with an E$EXIST exceptional condition.

Example

```
/*****************************************************************
 *  This example illustrates how the DELETE$REGION system call can  *
 *  be used.                                                        *

*****************************************************************/
```

DELETE$REGION

```
DECLARE TOKEN      LITERALLY 'SELECTOR';

/*  NUCLUS.EXT declares all system calls */
$INCLUDE(/rmx286/inc/NUCLUS.EXT)

    DECLARE region$token     TOKEN;
    DECLARE priority$queue   LITERALLY '1'; /* tasks wait in
                                               priority order */
    DECLARE status     WORD;

SAMPLEPROCEDURE:
    PROCEDURE;
        •
        •     Typical PL/M-286 Statements
        •

/*********************************************************************
 *  In order to delete a region, a task must know the token for     *
 *  that region.  In this example, the needed token is known        *
 *  because the calling task creates the region.                    *

*********************************************************************/

    region$token = RQ$CREATE$REGION    (priority$queue, @status);
        •
        •     Typical PL/M-286 Statements
        •

/*********************************************************************
 *  When the region is no longer needed, it may be deleted by       *
 *   any task that knows the token for the region.                  *

*********************************************************************/

    CALL RQ$DELETE$REGION (region$token, @status);

        •
        •     Typical PL/M-286 Statements
        •

END SAMPLEPROCEDURE;
```

Condition Codes

E$OK	0000H	No exceptional conditions.
E$CONTEXT	0005H	The deletion is being requested by a task that currently holds access to data protected by the region.
E$EXIST	0006H	The region parameter is not a token for an existing object.
ENOTCONFIGURED	0008H	This system call is not part of the present configuration.
E$TYPE	8002H	The region parameter is a token for an object that is not a region.

The RECEIVE$CONTROL system call allows the calling task to gain access to data protected by a region.

CAUTION

Tasks which use regions cannot be deleted while they access data protected by the region. Therefore, you should avoid using regions in Human Interface applications. If a task in a Human Interface application uses regions, the application cannot be deleted asynchronously (via a CONTROL-C entered at a terminal) while the task is in the region.

```
CALL RQ$RECEIVE$CONTROL (region, except$ptr);
```

Input Parameter

region	A TOKEN for the region protecting the data to which the calling task wants access.

Output Parameter

except$ptr	A POINTER to a WORD to which the iRMX II Operating System will return the condition code generated by this system call.

Description

The RECEIVE$CONTROL system call requests access to data protected by a region. If no task currently has access, entry is immediate. If another task currently has access, the calling task is placed in the region's task queue and goes to sleep. The task remains asleep until it gains access to the data.

If the region has a priority-based task queue, the priority of the task currently having access is temporarily boosted, if necessary, to match that of the task at the head of the queue.

Example

```
/*************************************************************************
 *  This example illustrates how the RECEIVE$CONTROL system call can be *
 *  used to gain access to data protected by a region.                   *
*************************************************************************/
```

```
DECLARE TOKEN       LITERALLY 'SELECTOR';

/*  NUCLUS.EXT declares all system calls */
$INCLUDE(/rmx286/inc/NUCLUS.EXT)

    DECLARE region$token                TOKEN;
    DECLARE priority$queue              LITERALLY '1'; /* tasks wait in
                                                          priority order */
    DECLARE status                      WORD;

SAMPLEPROCEDURE:
    PROCEDURE;

        •
        •     Typical PL/M-286 Statements
        •

/*************************************************************************
 *  In order to access the data within a region, a task must know the   *
 *  token for that region.  In this example, the needed token is known  *
 *  because the calling task creates the region.                        *
*************************************************************************/

    region$token = RQ$CREATE$REGION     (priority$queue,
                                         @status);
        •
        •     Typical PL/M-286 Statements
        •

/*************************************************************************
 *  When access to the data protected by a region is needed, the        *
 *  calling task may invoke the RECEIVE$CONTROL system call.            *
 *************************************************************************/

    CALL RQ$RECEIVE$CONTROL  (region$token,
                              @status);
        •
        •     Typical PL/M-286 Statements
        •

END SAMPLEPROCEDURE;
```

Condition Codes

E$OK	0000H	No exceptional conditions.
E$CONTEXT	0005H	The region parameter refers to a region already accessed by the calling task.
E$EXIST	0006H	The region parameter is not a token for an existing object.
ENOTCONFIGURED	0008H	This system call is not part of the present configuration.
E$TYPE	8002H	The region parameter contains a token for an object that is not a region.

The SEND$CONTROL system call allows a task to surrender access to data protected by a region.

CAUTION

Tasks that use regions cannot be deleted while they access data protected by the region. Therefore, you should avoid using regions in Human Interface applications. If a task in a Human Interface application uses regions, the application cannot be deleted asynchronously (via a CONTROL-C entered at a terminal) while the task is in the region.

```
CALL RQ$SEND$CONTROL (except$ptr);
```

Output Parameter

except$ptr — A POINTER to a WORD to which the iRMX II Operating System will return the condition code generated by this system call.

Description

When a task finishes with data protected by a region, the task invokes the SEND$CONTROL system call to surrender access. If the task is using more than one set of data, each of which is protected by a region, the SEND$CONTROL system call surrenders the most recently obtained access. When access is surrendered, the system allows the next task in line to gain access.

If a task calling SEND$CONTROL has had its priority boosted while it had access through a region, its priority is restored when it relinquishes the access.

Example

```
/*************************************************************************
 *  This example illustrates how the SEND$CONTROL system call can be     *
 *  used to surrender access to data protected by a region.              *
 *************************************************************************/

DECLARE TOKEN       LITERALLY 'SELECTOR';

/*  NUCLUS.EXT declares all system calls */
$INCLUDE(/rmx286/inc/NUCLUS.EXT)
```

```
    DECLARE region$token          TOKEN;
    DECLARE priority$queue        LITERALLY '1'; /* tasks wait in
                                                    priority order*/
    DECLARE status                WORD;
       •
       •     Typical PL/M-286 Statements
       •

SAMPLEPROCEDURE:
    PROCEDURE;

/*************************************************************************
 *  In order to access the data within a region, a task must know the    *
 *  token for that region.  In this example, the needed token is known   *
 *  because the calling task creates the region.                         *
 *************************************************************************/
    region$token = RQ$CREATE$REGION    (priority$queue,
                                        @status);
       •
       •     Typical PL/M-286 Statements
       •

/*************************************************************************
 *  When access to the data protected by a region is needed, the         *
 *  calling task may invoke the RECEIVE$CONTROL system call.             *
 *************************************************************************/
    CALL RQ$RECEIVE$CONTROL            (region$token,
                                        @status);
       •
       •     Typical PL/M-286 Statements
       •

/*************************************************************************
 *  When a task finishes using data protected by a region, the task      *
 *  invokes the SEND$CONTROL system call to surrender access.            *
 *************************************************************************/

    CALL RQ$SEND$CONTROL  (@status);

       •
       •     Typical PL/M-286 Statements
       •

END SAMPLEPROCEDURE;
```

Condition Codes

E$OK	0000H	No exceptional conditions.
E$CONTEXT	0005H	The calling task does not have access to data protected by any region.
ENOTCONFIGURED	0008H	This system call is not part of the present configuration.

The ATTACH$BUFFER$POOL system call associates a buffer pool with one or more ports.

```
RQ$ATTACH$BUFFER$POOL(buffer$pool$tkn, port$tkn, except$ptr);
```

Input Parameters

buffer$pool$tkn	A TOKEN identifying the buffer pool to be attached to the port.
port$tkn	A TOKEN identifying the port that is to gain the use of the buffer pool.

Output Parameters

except$ptr	A POINTER to a WORD in which the Operating System will return the condition code generated for this system call.

Description

The RQ$ATTACH$BUFFER$POOL system call makes a buffer pool's memory resources available to a port. A single buffer pool can have several ports attached to it, but a port may have only one buffer pool attached. Both the port and the attached buffer pool must belong to the same job.

The Nucleus Communication Service will allocate buffers from this buffer pool to satisfy receive operations of associated ports. The applications, however, are responsible for returning these buffers to the buffer pool when they are no longer needed.

Condition Codes

E$OK	0000H	No exceptional conditions.
E$CONTEXT	0005H	The port and the buffer pool tokens refer to objects that are not in the same job.
E$EXIST	0006H	Either the port$tkn or the buffer$pool$tkn parameter does not refer to an existing object.
ENOTCONFIGURED	0008H	This system call is not part of the present configuration.

E$STATE	0007H	The specified port already has a buffer pool attached.
E$TYPE	8002H	Either buffer$pool$tkn or the port$tkn parameter refers to an object that is not the correct type.
E$PROTOCOL	80E0H	The port specified in the port$tkn parameter is of the signal type, not the data communication type.

The RQ$ATTACH$PORT system call forwards all messages sent to the port that issued the call to another port known as a sink port.

```
CALL RQ$ATTACH$PORT(port$tkn, sink$port, except$ptr);
```

Input Parameters

port$tkn	A TOKEN for the port that will forward its messages.
sink$port	A TOKEN for the port that will receive the forwarded messages.

Output Parameters

except$ptr	A POINTER to a WORD to which the Operating System will return the condition code generated for this system call.

Description

The RQ$ATTACH$PORT system call is used to forward messages from one port to another. The port that issues the RQ$ATTACH$PORT system call is referred to as the source port, that is its messages will be sent to the attached port. The port that is attached by the call is referred to as the sink port, that is, the messages sent to the source port are forwarded to it.

More that one source port can be attached to a single sink port. Using a sink port allows a single task to receive messages from several connected source ports. The source ports can be on a remote agent, that is, another board in the system.

Messages that are already queued at the source port are not forwarded, only messages that are sent after the RQ$ATTACH$PORT system call is issued. Only one level of forwarding is supported. A sink port may not issue an RQ$ATTACH$PORT and forward messages from its source port on to another port.

If a source port issues an RQ$SEND$RSVP system call with the "use RECEIVE$REPLY" option, the response message is not forwarded to the sink port, it will be sent to the source port that issued the call.

Condition Codes

E$OK	0000H	No exceptional conditions.
E$EXIST	0006H	Either the port parameter or the sink$port parameter refers to an object that is not a port.
ENOTCONFIGURED	0008H	This system call is not part of the present configuration.
E$PROTOCOL	80E0H	The port specified in the port$tkn is of the signal type, not the data communication type.
E$STATE	0007H	The forwarding port is already attached to a sink port, or the sink port is attached to another source port.
E$TYPE	8002H	Either the port parameter or the parameter sink$port parameter is not an existing object.

The RQ$BROADCAST system call sends a control message to every agent on the iPSB bus.

```
CALL RQ$BROADCAST(port$tkn, socket, control$ptr, except$ptr);
```

Input Parameters

port$tkn	A TOKEN that indicates the port that is sending the broadcast message.
socket	A DWORD (host$id:port$id) that is the remote port that is to receive the broadcast message.
control$ptr	A POINTER to a control message.

Output Parameters

except$ptr	A POINTER to a WORD to which the Operating System will return the condition code generated for this system call.

Description

This system call sends a control message to each board on a message-passing bus. The host$id portion of the socket is ignored. This call can broadcast a message to one port on each board.

Condition Codes

E$OK	0000H	No exceptional conditions.
E$EXIST	0006H	The port$tkn parameter does not refer to an existing object.
ENOTCONFIGURED	0008H	This system call is not part of the present configuration.
ENUCBAD$BUF	80E2H	One or more of the following is true: • control$ptr is not a valid pointer to a buffer • The buffer pointed to by control$ptr or data$ptr is not large enough to hold the message.

E$PROTOCOL	80E0H	The specified destination port is a signal service port.
E$TRANSMISSION	000BH	A NACK (negative acknowledgment), timeout, bus or agent error, or retry expiration occurred during the transmission of the message.
E$TYPE	8002H	The port$tkn parameter refers to an object that is not a port.

The RQ$CANCEL system call performs synchronous cancellation of RSVP message transmission.

```
CALL RQ$CANCEL (port$tkn, trans$id, except$ptr);
```

Input Parameters

port$tkn	A TOKEN indicating the port that was the source of a previous send RSVP operation.
trans$id	A WORD that is the transaction ID of the message transmission to be canceled.

Output Parameters

except$ptr	A POINTER to a WORD in which the Operating System will return the condition code generated for this system call.

Description

The RQ$CANCEL system call performs a synchronous termination of an RSVP message transmission. In the case of canceling an RQ$SEND$RSVP system call, the RSVP buffer, if any, is disassociated from the port. The transaction ID of an RQ$SEND$RSVP system call can be canceled by the Nucleus Communication Service after the initial request is made, but before a response is received. That is, the transaction is canceled whether or not the receiving task has done a receive via the RQ$RECEIVE or the RQ$RECEIVE$REPLY system call.

Condition Codes

E$OK	0000H	No exceptional conditions.
E$EXIST	0006H	The port$tkn parameter does not refer to an existing object.
ENOTCONFIGURED	0008H	This system call is not part of the present configuration.
E$PROTOCOL	80E0H	The specified destination port is of the signal service only type.

E$TRANS$ID	00E8H	Either the trans$id parameter is invalid, or the entire transaction is already complete. The transaction is considered to be complete if the Nucleus Communication Service has received a response.
E$TYPE	8002H	The port$tkn parameter refers to an object that is not a port.

The RQ$CONNECT system call locally connects a port and assigns a default remote socket.

```
CALL RQ$CONNECT (port$tkn, socket, except$ptr);
```

Input Parameters

port$tkn	A TOKEN to a port object.
socket	A DWORD that identifies the remote socket. Sockets are identified by a host$id:port$id combination.

Output Parameter

except$ptr	A POINTER to a WORD to which the Operating System will return the condition code generated by this system call.

Description

The RQ$CONNECT system call creates a connection between the sending task and a remote port. A default remote socket is also assigned, if no socket is specified during a send or receive operation the default socket is used. Issuing an RQ$CONNECT system call using a zero (0) for the socket parameter disconnects the calling task's port.

While a port is connected, all messages sent from it go to the remote port specified in the socket parameter. Only messages sent by the remote port specified in the socket parameter will be received. Any message that comes in from another port will not be delivered to the connected port.

Exception Codes

E$OK	0000H	No exceptional conditions.
E$EXIST	0006H	The port parameter does not refer to an existing object.
E$HOST$ID	00E2H	The host$id portion of the socket does not refer to an agent (board) that is currently in the message space.

ENOTCONFIGURED	0008H	This system call is not part of the present configuration.
E$PROTOCOL	80E0H	The port specified in the port$tkn parameter is of the signal type, not the data communication type.
E$STATE	0007H	The port specified in the port$tkn parameter parameter is already the sink port of a forwarded port and cannot be connected again. Only one level of port forwarding is supported.
E$TYPE	8002H	The port parameter refers to an object that is not a port.

The CREATE$PORT system call creates a port object that can be used to send and receive MULTIBUS II messages between bus agents. An agent is any board on the bus.

```
port$tkn = RQ$CREATE$PORT (num$trans, info$ptr, except$ptr);
```

Input Parameters

num$trans — A WORD that identifies the message-passing protocol associated with this port. Supported values are: 2, which indicates data transport service, and 3, which indicates the Message Interrupt Controller signal service.

info$ptr — A POINTER to an information segment that is protocol-dependent. The segment contents are associated with the port object.

The definition of the structure for the signal protocol is:

```
DECLARE port$creation$info LITERALLY STRUCTURE (
                              message$id      BYTE,
                              reserved        BYTE,
                              type            BYTE,
                              reserved        BYTE
                              flags           WORD);
```

where:

message$id — The slot ID of the remote agent. The message$id must be in the range of 0 to 19 decimal.

reserved — Reserved for future use. This value should be set to zero.

type The message protocol of the port as specified by:

Value	Meaning
0-1	Reserved for the Nucleus Communications Service
2	Data Transport Service
3	Signal Service
4-0FFH	Intel Reserved, should be set to 0H.

flags A WORD whose bit encoding defines the port's task queuing discipline as:

Bit	Meaning
0	Reserved, should be set to zero.
1	Task queue discipline. If set (1), then tasks are queued according to priority. If not set (0), then tasks are queued in FIFO order.
2-15	Reserved, should be set to zero.

The definition of the structure for the signal protocol is:

```
DECLARE port$creation$info LITERALLY STRUCTURE (
                            port$id         WORD,
                            type            BYTE,
                            reserved        BYTE,
                            flags           WORD);
```

where:

port$id A WORD value that identifies the port. Port ID values are:

ID Range	Explanation
0	The Nucleus Communication Service will assign the port ID.
1-7FFH	Reserved values.
800H-0FFFH	Available to users.
1000H-0FFFFH	Reserved for CREATE$PORT.

type The message protocol of the port as specified by:

Value	Meaning
0-1	Reserved for the Nucleus Communications Service
2	Data Transport Service
3	Signal Service
4-0FFH	Intel Reserved, should be set to 0H.

flags A WORD whose bit encoding defines the port's task queuing discipline as:

Bit	Meaning
0	Reserved, should be set to zero.
1	Task queue discipline. If set (1), then tasks are queued according to priority. If not set (0), then tasks are queued in FIFO order.
2	Defines whether or not the port will perform message fragmentation if an incoming message is too large for any single buffer. 0H is fragmentation enabled, 1H is fragmentation disabled.
3-15	Reserved, should be set to zero.

Output Parameters

port$tkn A TOKEN to which the Operating System will return a token for the new port.

except$ptr A POINTER to a WORD to which the Operating System will return the condition code generated for this system call.

Description

The CREATE$PORT system call creates a port object and returns a token for the newly created port. The new port counts as one debit against the total number of objects permitted on a single iRMX II board. Once the port is established, the task can send and receive messages or signals through the port. Other tasks created within the same job can also use the port.

For ports of the signal service type, only one connection can be established between any two agents. Attempting to connect more than one port to the same agent results in an E$CONTEXT exceptional condition.

NOTE

Ports of the signal service type receive messages before ports of the data transport type. Therefore, if you create both types of ports on one board, only the signal service ports will receive messages from the remote agent associated with it. Ports of the data transport type will not receive messages from the associated agent.

Condition Codes

E$OK	0000H	No exceptional conditions.
E$CONTEXT	0005H	The signal service protocol was specified and the agent$id given already has a port associated with it.
E$LIMIT	0004H	The calling task's job has already reached its object limit.
E$MEM	0002H	The memory available to the calling task's job is not sufficient to create a port.
ENOTCONFIGURED	0008H	This system call is not part of the present configuration.
ENUCBAD$BUF	80E2H	The info$ptr is invalid or points to a buffer that is not large enough.
E$PARAM	8004H	The protocol$type field does not specify the signal service, or the agent$id was invalid (i.e., greater than 19 decimal).

E$PORT$ID$USED	80E7H	The port$id specified for a data transaction port is in use.
E$SLOT	000CH	There isn't enough room in the GDT for another descriptor.

The DELETE$PORT system call deletes a port.

```
CALL RQ$DELETE$PORT (port$tkn, except$ptr);
```

Input Parameter

port$tkn	A TOKEN for the port to be deleted.

Output Parameter

except$ptr	A POINTER to a WORD to which the Operating System will return the condition code generated for this system call.

Description

The DELETE$PORT system call deletes the specified port. If any tasks are in the port's receive task queue at the moment of deletion, they are awakened with an E$EXIST exceptional condition. Deleting the port counts as a credit toward the object total of the containing job. Any messages queued at the port are discarded and , if the port is forwarded, forwarding is severed.

Condition Codes

E$OK	0000H	No exceptional conditions.
E$EXIST	0006H	Either the port parameter is not a token for an existing object or it represents a port whose job is in the process of being deleted.
ENOTCONFIGURED	0008H	This system call is not part of the present configuration.
E$TYPE	8002H	The port parameter is a token for an object that is not a port

The RQ$DETACH$BUFFER$POOL system call ends the association between a buffer pool and a port.

```
buffer$pool$tkn = RQ$DETACH$BUFFER$POOL(port$tkn, except$ptr);
```

Input Parameters

port$tkn	A TOKEN identifying the port that is detaching the buffer pool.

Output Parameters

buffer$pool$tkn	A TOKEN that is returned as a result of the call. The returned TOKEN references the buffer pool that was detached.
except$ptr	A POINTER to a WORD to which the Operating System will return the condition code generated for this system call.

Description

The RQ$DETACH$BUFFER$POOL system call breaks the association between a port and buffer pool. This call does not delete the buffer pool. The TOKEN received as a result of this call can be used to attach the buffer pool to a different port, or to reattach it to the same port.

Condition Codes

E$OK	0000H	No exceptional conditions.
E$EXIST	0006H	The port$tkn parameter is not a TOKEN for an existing object.
ENOTCONFIGURED	0008H	This system call is not part of the present configuration.
E$PROTOCOL	80E0H	The port specified in the port$tkn parameter is of the signal type, not the data communication type.

E$STATE	0007H	No port is associated with the specified port.
E$TYPE	8002H	The port$tkn parameter refers to an object that is not a port.

The RQ$DETACH$PORT system call ends message forwarding from the source port to the sink port.

```
CALL RQ$DETACH$PORT (port$tkn, except$ptr);
```

Input Parameter

port$tkn	A TOKEN for the source port that is to be detached.

Output Parameter

except$ptr	A POINTER to a WORD to which the Operating System will return the condition code generated for this system call.

Description

The RQ$DETACH$PORT system call ends message forwarding from a source port to a sink port. In message forwarding, messages originally sent to the source port are forwarded to the sink port. If an RQ$DETACH$PORT is issued and messages are queued at the sink port, they remain at the sink port until removed with a receive operation.

Condition Codes

E$OK	0000H	No exceptional conditions.
E$EXIST	0006H	The port parameter does not refer to an existing port.
ENOTCONFIGURED	0008H	This system call is not part of the present configuration.
E$PROTOCOL	80E0H	The port specified in the port$tkn parameter is of the signal type, not the data communication type.

E$STATE	0007H	The port issuing the call does not have a sink port attached.
E$TYPE	8002H	The port parameter refers to an object that is not a port.

The RQGETHOST$ID system call returns the host ID of the board (agent) that the task is running on.

```
host$id = RQ$GET$HOST$ID(except$ptr);
```

Input Parameters

None

Output Parameters

except$ptr	A POINTER to a WORD in which the Operating System will place the condition code generated for this system call.

Description

The RQGETHOST$ID system call returns the host ID for the board on which the issuing task is running. The host ID is the first part of the host$id:port$id pair that makes up a socket. A socket is a destination for (either return or remote) messages. This system call is used to construct local sockets to be used as return addresses for messages.

Condition Codes

E$OK	0000H	No exceptional conditions.
ENOTCONFIGURED	0008H	This system call is not part of the present configuration.

The GET$PORT$ATTRIBUTES system call returns information about how the specified port is set up.

```
CALL RQ$GET$PORT$ATTRIBUTES(port$tkn, info$ptr, except$ptr);
```

Input Parameters

port$tkn — A TOKEN for the port about which you need information.

info$ptr — A POINTER to a structure into which the Operating System will write the information about the port.

```
DECLARE get$port$info LITERALLY 'STRUCTURE
        port$id                 WORD,
        type                    BYTE,
        reserved                BYTE,
        num$trans               WORD,
        reserved (2)            WORD,
        sink$port               TOKEN,
        default$remote$socket   DWORD,
        buffer$pool             TOKEN,
        flags                   WORD,
        reserved                BYTE)';
```

where

port$id — is the unique port id for the port

type — a WORD that specifies the type of messages that can be sent to and from this port, the types are:

Type Value	Meaning
0-1	Reserved for the Nucleus Communications Service
2	Send/Receive data messages
3	Send/Receive dataless (signal) messages.
4-0FFH	Intel Reserved

reserved — A WORD that is reserved and should be set to zero.

num$trans	The number of simultaneous transactions that can be outstanding at this port.
reserved	A WORD that is reserved and should be set to zero.
reserved	A WORD that is reserved and should be set to zero.
sink$port	A TOKEN for another port that receives forwarded messages from the port you are examining. This parameter contains a zero if there is no sink port, that is, the port being examined has not issued an RQ$ATTACH$PORT system call.
default$remote-socket	A host$id:port$id combination that specifies a default destination/source for all messages sent/received at this port. This parameter contains a zero if there is no default remote socket, that is the port being examined has not issued an RQ$CONNECT system call.
buffer$pool	A TOKEN for the buffer pool, if any, that is attached to this port. This parameter contains a zero if the port being examined does not have a buffer pool attached, that is the port being examined has not issued an RQ$ATTACH$BUFFER$POOL system call.
flags	A WORD whose value is interpreted as follows:

Bit	Meaning
0	Reserved
1	How messages are queued at the port. If zero, the message queue is FIFO. If one, the message queue is priority.
2	Defines if the port supports RSVP request message fragmentation. If zero, then fragmentation is supported. If one, then fragmentation is not supported.
4-15	Reserved

exception$ptr	A POINTER to a WORD to which the Operating System will return the condition code generated for this system call.

Description

The RQGETPORT$ATTRIBUTES system call returns information about the specified port.

Condition Codes

E$OK	0000H	No exceptional conditions.
E$EXIST	0006H	The port$tkn parameter does not refer to an existing object.
ENOTCONFIGURED	0008H	This system call is not part of the present configuration.
ENUCBAD$BUF	80E2H	The info$ptr parameter is invalid or points to a buffer that is not large enough.
E$TYPE	8002H	The port$tkn parameter refers to an object that is not a port.

The RQ$RECEIVE system call accepts a message at a port.

```
data$ptr = RQ$RECEIVE(port$tkn, time$limit, info$ptr, except$ptr);
```

Input Parameters

port$tkn — A TOKEN for the port that is issuing the call.

time$limit — A WORD that specifies the maximum time to wait for the message to arrive at the port specified in the port$tkn parameter. Acceptable values are:

Value	Meaning
0	Do not wait
65535	Wait forever
1-65534	Wait the specified number of clock intervals

info$ptr — A POINTER to a STRUCTURE of the following type:

```
STRUCTURE(
    flags               WORD,
    status              WORD,
    trans$id            WORD,
    data$length         DWORD,
    forwarding$port     TOKEN,
    remote$socket       SOCKET,
    control$msg(20)     BYTE,
    reserved(4)         BYTE);
```

where:

flags is a WORD with the following encoded meaning:

Bit	Name
0-3	data$type
4-7	receive$type
8-15	reserved

where:

data$type defines whether data$ptr points to a data chain (01B) or a single buffer (00B.) Other values are reserved.

receive$type is an indicator of the type of message received as follows:

Value	Message Type
0000B	Transactionless message (RQ$SEND or similar call)
0001B	Transmission or system status message
0010B	Transaction request message (RQ$SEND$RSVP or similar call)
0100B	Transaction response message (RQ$SEND$REPLY or similar call)

status contains the send message status. The status codes are:

Status	Meaning
E$OK	A new message has been successfully received
E$CANCELED	A SEND$RSVP transaction has been remotely canceled.
ENOLOCAL$BUFFER	This error applies to two cases: If the receive$type parameter indicates a request message, the local port's buffer pool does not contain a buffer large enough to hold the message so the RQ$RECEIVE$FRAGMENT system call is required (message fragmentation.) If the receive$type parameter indicates a response message, the RSVP buffer supplied in the RQ$SEND$RSVP system call is not large enough to hold the response.
ENOREMOTE$BUFFER	The remote port's buffer pool does not have a buffer large enough to hold the message and message fragmentation is disabled.
E$TRANSMISSION	A NACK (Negative Acknowledgment), MPC Failsafe timeout, bus or agent error, or retry expiration occurred during the transmission of the message.

trans$id	A WORD that contains the transaction ID for this message. If trans$id is zero, a new transactionless message has been received. If trans$id is not zero, it either indicates a request or response message has been received, or it indicates an asynchronous transmission status message has been received.
data$length	A DWORD that indicates the length of the data message received. If **receive$type** indicates a newly received message, then data$length contains the length of the successfully received message. If **receive$type** and **status** indicate request message fragmentation, the data$length contains the length of all the message fragments that will be received using the RQ$RECEIVE$FRAGMENT system call.
forwarding$port	A TOKEN indicates a port. The indicated port is the source port for the port that is actually receiving the message.
remote$socket	A SOCKET (host$id:port$id) that indicates the remote message source.
control$msg	The 20-byte long control part of a data message.

Output Parameters

except$ptr	A POINTER to a WORD that will contain the condition code generated by the Operating System for this system call.
data$ptr	A POINTER that indicates the starting address of the data portion (if any) of the message after it has been received.

Description

The RQ$RECEIVE system call accepts a message at a port. If the message contains a data portion, a pointer to the buffer used to store the data portion is returned. When the buffer is no longer required the application should return it to the buffer pool using the RQ$RELEASE$BUFFER system call. If enough buffer space is not attached, the message is rejected by the receiving host.

Condition Codes

E$OK	0000H	No exceptional conditions.
E$EXIST	0006H	The port$tkn parameter does not refer to an existing object.
ENOTCONFIGURED	0008H	This system call is not part of the current configuration.
ENUCBAD$BUF	80E2H	The info$ptr parameter points to a buffer that either does not exist, or is not large enough.
E$PROTOCOL	80E0H	The port specified in the port$tkn parameter is of the signal type.
E$TIME	0001H	The time the task is willing to wait, specified in the time$limit parameter, expired before a message was received.
E$TYPE	8002H	The port$tkn parameter is a token for an object that is not a port

The RQ$RECEIVE$FRAGMENT system call accepts a part (fragment) of a request (RSVP) data message.

```
CALL RQ$RECEIVE$FRAGMENT (port$tkn, socket, rsvp$trans$id,
                          fragment$ptr, fragment$length, flags,
                          except$ptr);
```

Input Parameters

port$tkn — A TOKEN to the port issuing the call.

socket — A DWORD (host$id:port$id) that specifies the port from which the original RSVP message was sent. If the port issuing the RQ$RECEIVE$FRAGMENT system call is connected, the socket parameter is ignored.

rsvp$trans$id — A WORD value that is used to identify this particular message transaction. A transaction ID is generated each time an RQ$SEND$RSVP system call is issued.

fragment$ptr — A POINTER to a buffer in which the message fragment will be placed. If this POINTER is NIL, the receive of the message fragments is terminated.

fragment$length — A DWORD that defines the length of the fragment. If this length is zero, fragmented transmission of a request message is terminated.

flags — A WORD that defines the type of message fragment. The encoded values are:

Value	Meaning
01B	The data is in data chain form and fragment$ptr points to a data chain block.
00B	The data is in a single buffer and fragment$ptr points to the buffer.

Output Parameters

except$ptr — A POINTER to a WORD in which the Operating System will place the condition code generated for this system call.

Description

The RQ$RECEIVE$FRAGMENT system call accepts a message fragment that is sent from a remote host via an RQ$SEND$RSVP system call. The data that satisfied the RQ$SEND$RSVP system call could not be placed into a single buffer, therefore the message had to be broken into sections called fragments.

Condition Codes

E$OK	0000H	No exceptional conditions.
E$DISCONNECTED	00E9H	The socket parameter is equal to zero and the port is not connected.
E$EXIST	0006H	The port$tkn parameter does not refer to an existing object.
ENOTCONFIGURED	0008H	This system call is not part of the current configuration.
ENUCBAD$BUF	80E2H	The fragment$ptr parameter points to a buffer that either does not exist, or is not large enough.
E$PROTOCOL	80E0H	The port specified in the port$tkn parameter is of the signal type.
E$TIME	0001H	The system receive fragment failsafe timeout expired before the fragment was received
E$TRANS$ID	00E8H	The value is the rsvp$trans$id parameter does not specify a currently valid transaction.
E$TYPE	8002H	The port$tkn parameter is a token for an object that is not a port.

The RQ$RECEIVE$REPLY system call accepts a message that is a reply to an earlier request.

```
data$ptr=RQ$RECEIVE$REPLY (port$tkn, rsvp$tran$id, time$limit,
                           info$ptr, except$ptr);
```

Input Parameters

port$tkn — A TOKEN to the port object issuing the call. This port must not be a sink port.

rsvp$trans$id — The transaction ID returned from the associated RQ$SEND$RSVP system call.

time$limit — A WORD indicating the length of time the task is willing to wait for the reply, supported values are:

Value	Meaning
0	No wait
65535	Wait forever
1-65534	The number of clock cycles to wait

info$ptr — A POINTER to a structure of the following form:

```
STRUCTURE(
    flags           WORD,
    status          WORD,
    trans$id        WORD,
    data$length     DWORD,
    forwarding$port TOKEN,
    remote$socket   SOCKET,
    control$msg(20) BYTE,
    reserved        BYTE);
```

where:

flags is a WORD with the following encoded meaning:

Bit	Name
0-3	data$type
4-7	receive$type
8-15	reserved

where:

data$type defines the whether data$ptr points to a data chain (01B) or a single buffer (00B.) Other values are reserved.

receive$type is an indicator of the type of message received as follows:

Value	Message Type
0001B	Transmission or system status message
0100B	Transaction response message

status contains the send message status. The status codes are:

Status	Meaning
E$OK	A new message has been successfully received
E$CANCELED	A SEND$RSVP transaction has been remotely canceled.
ENOLOCAL$BUFFER	If the receive$type parameter indicates a response message, the RSVP buffer supplied in the RQ$SEND$RSVP system call is not large enough to hold the response.
ENOREMOTE$BUFFER	The remote port's buffer pool was not large enough to hold the message and message fragmentation is turned off.
E$TRANSMISSION	A NACK (Negative Acknowledgment, timeout, bus or agent error, or retry expiration occurred during the transmission of the message.
trans$id	A WORD that contains the transaction ID for this message. Trans$id indicates a response message has been received, or it indicates an erroneous asynchronous transmission status message has been received.

data$length	A DWORD that indicates the length of the data message received. If **receive$type** indicates a newly received message, then data$length contains the length of the successfully received message.
forwarding$port	A TOKEN indicating a port. The indicated port is the source port for the port that is actually receiving the message. This field does not apply to the RQ$RECEIVE$REPLY system call.
remote$socket	A SOCKET (host$id:port$id) that indicates the remote message source.
control$msg	The 20-byte long control part of a data message.
reserved	A reserved BYTE.

Output Parameters

except$ptr	A POINTER to a WORD that will contain the condition code generated by the Operating System for this system call.
data$ptr	A POINTER that indicates the starting address of the data portion (if any) of the message after it has been received.

Description

The RQ$RECEIVE$REPLY system call is issued when a task wants to wait for a reply to an RSVP message that it sent previously. This call cannot be issued by a sink port (a port that accepts messages forwarded from another port.)

Condition Codes

E$OK	0000H	No exceptional conditions.
E$EXIST	0001H	The port$tkn parameter does not refer to an existing object.
ENOTCONFIGURED	0008H	This system call is not part of the current configuration.

ENUCBAD$BUF	80E2H	The info$ptr parameter points to a buffer that either does not exist, or is not large enough.
E$PROTOCOL	80E0H	The port specified in the port$tkn parameter is of the signal type.
E$TIME	0006H	The time the task is willing to wait, specified in the time$limit parameter, expired before a message was received.
E$TRANS$ID	00E8H	Either an invalid transaction ID has been supplied, or the transaction was canceled before the response was received.
E$TYPE	8002H	The port$tkn parameter is a token for an object that is not a port.

The RECEIVE$SIGNAL system call receives a signal from a remote host at a specified port.

```
CALL RQ$RECEIVE$SIGNAL (port$tkn, wait$time, except$ptr);
```

Input Parameters

port$tkn — A TOKEN for the port where the signal is expected to arrive.

wait$time — A WORD that specifies how long the calling task is willing to wait.

- If zero, the calling task is not willing to wait.
- If 0FFFFH, the calling task will wait as long as necessary.
- If 1-0FFFEH, the calling task will wait that number of clock intervals.

Output Parameters

except$ptr — A POINTER to a WORD to which the Operating System will return the condition code generated by this system call.

Description

The RECEIVE$SIGNAL system call causes the calling task to either receive a signal or to wait the specified number of clock intervals at the specified port.

If a signal is already queued at the port, the calling task receives the signal. Otherwise, the task either goes to the end of the receive task queue to wait a specified amount of time, or it is not willing to wait. In the latter case, or if the task's waiting period elapses without a signal arriving, the task receives an E$TIME exceptional condition.

When a signal arrives and there are tasks on the receive task queue, the task at the head of the queue receives the signal. If no tasks are waiting in the queue, the signal is queued at the port. The next task to invoke RECEIVE$SIGNAL receives one of the queued signals.

Condition Codes

E$OK	0000H	No exceptional conditions.
E$EXIST	0006H	The port$tkn parameter does not refer to an existing object.
ENOTCONFIGURED	0008H	This system call is not part of the present configuration.
E$PROTOCOL	80E0H	The port specified in the port$tkn parameter is of the signal type, not the data communication type.
E$TIME	0001H	One of the following is true: • The calling task was not willing to wait and no signal was queued at the port. • The task's designated waiting period elapsed before the desired signal arrived.
E$TYPE	8002H	The port$tkn parameter refers to an object that is not a port.

The RQ$SEND system call sends a data message from a port to a port on another board.

```
TRANS$ID = RQ$SEND (port$t, socket, control$ptr, data$ptr,
                    data$length, flags, except$ptr);
```

Input Parameters

port$tkn	A TOKEN for the port to which a message is to be sent.
socket	A DWORD (host$id:port$id) that specifies a unique board/port combination that is the message destination. If the sending port has issued an RQ$CONNECT then it has a default socket and this parameter is ignored.
control$ptr	A POINTER to the control portion of a message. If data$ptr = NULL or data$length = 0, then the control message is 20 bytes in length. Otherwise, the control message is 16 bytes in length.
data$ptr	A POINTER to a data message. If this parameter is NULL, then there is no optional data portion for this message. If this parameter is not NULL then it points to either a contiguous buffer or a data chain block. If the data$type field of the flags parameter is set (1), then this pointer points to a data chain block; otherwise, (0) it points to a contiguous buffer.
data$length	A DWORD that indicates the length of the data message.
flags	A WORD whose value is interpreted as follows:

Bits	Name
0-3	data$type
4-7	mode
8-15	Reserved (set to zero)

where:

data$type	Describes the format in which the data is to be sent. If set (0001B), the data will be sent as a data$chain and data$ptr is a POINTER to the data chain block. If not set (0000B), the data is a single logical segment and data$ptr is a POINTER to a buffer.

mode	Defines the transmission mode. If set (1), the transmission is asynchronous. If not set (0), the transmission is synchronous. If the transmission is asynchronous, the send status must be received by an RQ$RECEIVE system call.

Output Parameters

trans$id	A WORD that is used to identify this particular message transmission. If no data is being sent, data$ptr = NULL, then the value returned is zero.
except$ptr	A POINTER to a WORD to which the Operating System will return the condition code generated for this system call.

Description

The RQ$SEND system call sends a data message from a port to a port on another board. If the remote port to which the message is sent does not have adequate buffer space to receive the message an ENOREMOTE$BUFFER error will be returned. No message fragmentation will be performed if this call is used.

Condition Codes

E$OK	0000H	No exceptional conditions.
E$DISCONNECTED	00E9H	The socket parameter is equal to zero and the port is not connected.
E$EXIST	0006H	The port$tkn parameter does not point to an existing object.
E$HOST$ID	00E2H	The host$id portion of the socket parameter does not refer to an agent (board) that is currently in the message space.
ENOREMOTE$BUFFER	00E3H	The receiving agent could not allocate a buffer to hold the message.
ENOTCONFIGURED	0008H	This system call is not part of the present configuration.
ENUCBAD$BUF	80E2H	Either the control$ptr or data$ptr parameter is invalid or points to a buffer that is not large enough.

E$PROTOCOL	80E0H	The port specified in the port$tkn parameter is of the signal type, not the data communication type.
E$RESOURCE$LIMIT	00E6H	The number of simultaneous messages, has been reached. This field is set during system configuration.
E$TRANS$LIMIT	00EAH	A transmission resource limitation has been encountered. An insufficient number of transaction buffers was specified during system configuration (The Max No. of Simultaneous Transactions parameter in the Interactive Configuration Utility's Nucleus Communication screen.)
E$TRANSMISSION	000BH	A NACK (negative acknowledgment), timeout, bus or agent error, or retry expiration occurred during the transmission of the message.
E$TYPE	8002H	The port$tkn parameter refers to an object that is not a port

The SEND$RSVP system call initiates a request/response message interchange.

```
trans$id = RQ$SEND$RSVP (port$tkn, socket, control$ptr, data$ptr,
                         data$length, rsvp$data$ptr, rsvp$data$length,
                         flags, except$ptr);
```

Input Parameters

port$tkn	The TOKEN that identifies the port sending the RSVP message.
socket	A DWORD (host$id:port$id) that identifies the remote destination. If the sending port has a default socket, this parameter is ignored.
control$ptr	A POINTER to the control portion of a message. If data$ptr = NULL or data$length = 0, then the control message is 20 bytes in length. Otherwise, the control message is 16 bytes in length.
data$ptr	A POINTER to a data message. If NULL, then a control message is sent. If not NULL, then it is a POINTER to either a contiguous buffer, or a data chain block.
data$length	A DWORD that is the length of the data message.
rsvp$data$ptr	A POINTER to a buffer into which the expected response is to be placed. This buffer must be a contiguous block.
rsvp$data$length	A DWORD that defines the length of the RSVP message buffer.
flags	A WORD whose value is interpreted as follows:

Bits	Name
0-3	data$type
4-7	mode
8	receive$reply
9-15	Reserved (set to zero)

where:

data$type	Describes the format in which the data is to be sent. If set (0001B), the data will be sent as a data$chain and data$ptr is a POINTER to the data chain block. If not set (0000B), the data is a single logical segment and data$ptr is a POINTER to a buffer.
mode	Defines the transmission mode. If set (1), the transmission is asynchronous. If not set (0), the transmission is synchronous.
receive$reply	Defines which system call, receive$reply or receive, will be used to receive the response message. If set (1), then RECEIVE is being used, if not set (0), then RECEIVE$REPLY is being used.

Output Parameters

trans$id	A WORD that is used to identify this particular message transmission. If no data is being sent, data$ptr = NULL, then the value returned is zero.
except$ptr	A POINTER to a WORD to which the Operating System will return the condition code generated for this system call.

Description

The RQ$SEND$RSVP system call initiates a request with implied response interchange. Typically RSVP interchanges are used to transfer data from one agent to another. The parameter rsvp$data$ptr is used to supply a POINTER to a buffer that is the destination of the response data.

Condition Codes

E$OK	0000H	No exceptional conditions.
E$DISCONNECTED	00E9H	The port has no default socket and a zero was specified for the socket parameter.
E$EXIST	0006H	Either the port$tkn, control$ptr, data$ptr parameter does not point to an existing object.
E$HOST$ID	00E2H	The host$id portion of the socket parameter does not refer to an agent (board) that is currently in the message space.
ENOREMOTE$BUFFER	00E3H	The receiving agent could not allocate a buffer to hold the message.

ENOTCONFIGURED	0008H	This system call is not part of the present configuration.
ENUCBAD$BUF	80E2H	The info$ptr is invalid or points to a buffer that is not large enough.
E$PROTOCOL	80E0H	The port specified in the port$tkn parameter is of the signal type, not the data communication type.
E$RESOURCE$LIMIT	00E6H	Either the number of simultaneous messages, or simultaneous transactions, has been reached. These fields are set during system configuration.
E$TRANSMISSION	000BH	A NACK (negative acknowledgment), timeout, bus or agent error, or retry expiration occurred during the transmission of the message.
E$TRANS$LIMIT	00EAH	A transmission resource limitation has been encountered. An insufficient number of transaction buffers was specified during system configuration (The Max No. of Simultaneous Transactions parameter in the Interactive Configuration Utility's Nucleus Communication screen.).
E$TYPE	8002H	The port$tkn parameter refers to an object that is not a port.

The RQ$SEND$REPLY system call is sent in response to the RQ$SEND$RSVP system call.

```
trans$id = RQ$SEND$REPLY (port$tkn, socket, rsvp$trans$id,
                          control$ptr, data$ptr, data$length, flags,
                          except$ptr);
```

Input Parameters

port$tkn	The TOKEN that identifies the port sending the REPLY message.
socket	A DWORD (host$id:port$id) that identifies the remote destination. If the sending port has a default socket, this parameter is ignored.
rsvp$trans$id	This is the trans$id parameter from the SEND$RSVP call that is being answered. This WORD is used at the destination to identify the transaction that is being answered.
control$ptr	A POINTER to the control portion of the message. If data$ptr = NULL or data$length = 0, then the control message is 20 bytes in length. Otherwise, the control message is 16 bytes in length.
data$ptr	A POINTER to a data message. If NULL, then a control message is sent. If not NULL, then it is a POINTER to either a contiguous buffer, or a data chain block.
data$length	A DWORD that is the length of the data message.
flags	A WORD whose value is interpreted as follows:

Bits	Name
0-3	data$type
4-7	mode
8	Reserved (Set to zero)
9	EOT
10-15	Reserved (Set to zero)

where:

data$type	Describes the format in which the data is to be sent. If set (0001B), the data will be sent as a data$chain and data$ptr is a POINTER to the data chain block. If not set (0000B), the data is a single logical segment and data$ptr is a POINTER to a buffer.
mode	Defines the transmission mode. If set (1), the transmission is asynchronous. If not set (0), the transmission is synchronous.
EOT	Defines a SEND$REPLY end-of-transaction option. If EOT is not set, 000B, then this message is the last fragment of a response. Otherwise, more fragments will be sent.

Output Parameters

trans$id	A WORD that is used to identify this particular message transmission. If no data is being sent, data$ptr = NULL, then the value returned is zero.
except$ptr	A POINTER to a WORD to which the Operating System will return the condition code generated for this system call.

Description

The RQ$SEND$REPLY system call is an answer to a previous SEND$RSVP system call. The reply message may be sent as a single message or as a series of message fragments, as controlled by the EOT flag.

Condition Codes

E$OK	0000H	No exceptional conditions.
E$DISCONNECTED	00E9H	The port sending the message has previously issued an RQ$CONNECT to a remote port. The board on which the remote port is located has been reset.
E$EXIST	0006H	The port$tkn parameter does not point to an existing object.
E$HOST$ID	00E2H	The host$id portion of the socket parameter does not refer to an agent (board) that is currently in the message space.
ENOREMOTE$BUFFER	00E3H	The receiving agent could not allocate a buffer to hold the message.

ENOTCONFIGURED	0008H	This system call is not part of the present configuration.
ENUCBAD$BUF	80E2H	The info$ptr is invalid or points to a buffer that is not large enough.
E$PARAM	8004H	The flags parameter is illegally specified.
E$PROTOCOL	80E0H	The port specified in the port$tkn parameter is of the signal type, not the data communication type.
E$RESOURCE$LIMIT	00E6H	Either the number of simultaneous messages, or simultaneous transactions, has been reached. These fields are set during system configuration.
E$TRANSMISSION	000BH	A NACK (negative acknowledgment), timeout, bus or agent error, or retry expiration occurred during the transmission of the message.
E$TRANS$LIMIT	00EAH	A transmission resource limitation has been encountered.
E$TYPE	8002H	The port$tkn parameter refers to an object that is not a port

The RQ$SEND$SIGNAL system call sends a MULTIBUS II signal (dataless message) to a remote agent (board) through the specified port.

```
CALL RQ$SEND$SIGNAL (port$tkn, except$ptr);
```

Input Parameter

port$tkn	A TOKEN for the port through which the signal will be sent.

Output Parameter

except$ptr	A POINTER to a WORD to which the Operating System will return the condition code generated for this system call.

Description

The SEND$SIGNAL system call sends a signal (dataless message) to a remote agent through the specified port. If a bus timeout or other bus error occurs, the calling task receives an E$TRANSMISSION exceptional condition.

Condition Codes

E$OK	0000H	No exceptional conditions.
E$EXIST	0006H	The port parameter is not a token for an existing object.
ENOTCONFIGURED	0008H	This system call is not part of the present configuration.
E$PROTOCOL	80E0H	The port specified in the port$tkn parameter is of the data communication type, not the signal type.
E$TRANSMISSION	000BH	A NACK (negative acknowledgement), timeout, bus or agent error, or retry expiration occurred during the transmission of the signal
E$TYPE	8002H	The port parameter is a token for an object that is not a port.

The GET$INTERCONNECT system call retrieves the contents of the specified interconnect register.

```
value = RQ$GET$INTERCONNECT (slot$number, reg$number, except$ptr);
```

Input Parameters

slot$number

A BYTE that specifies the MULTIBUS II cardslot number of the board on which the specified interconnect register is located. Specify either the physical cardslot number or 31 decimal (the host processor board). The following decimal values are valid:

- If in the range 0-19 decimal, this parameter specifies Parallel System Bus (iPSB) slot numbers 0 to 19, respectively.
- If in the range 24-29 decimal, this parameter specifies iLBX II cardslot numbers 0 to 5 respectively on the host processor board's Local Bus Extension (iLBX II).
- If 31 decimal, this parameter specifies that you wish to retrieve the contents of an interconnect register from interconnect space on the calling task's processor board. In this case, do not specify the actual cardslot number; incorrect values could be returned.

Values 20-23, 30, and any decimal value greater than 31 are invalid.

reg$number

A WORD that specifies the interconnect register to read. This value must be in the range 0000H to 01FFH, the defined range of interconnect space. Refer to the hardware manual for your particular board to determine the proper register number.

Output Parameters

value

A BYTE in which this system call returns the contents of the interconnect register. If the cardslot is empty or if the actual cardslot number for the host processor board is specified (rather than 31), 0 is returned for iPSB cardslots and 0FFH for iLBX II cardslots.

except$ptr

A POINTER to a WORD to which the Operating System will return the condition code generated for this system call.

Description

The GET$INTERCONNECT system call returns the contents of the interconnect register specified by the slot$number and reg$number parameters.

CAUTION

The Nucleus checks the range validity of the cardslot and register numbers specified in the call, but it does not verify the existence of a board in any specific cardslot. Nor does it assign any meaning to the register being accessed.

Condition Codes

E$OK	0000H	No exceptional conditions.
ENOTCONFIGURED	0008H	This system call is not part of the present configuration.
E$PARAM	8004H	One or more of the input parameters has an illegal value.

The RQSETINTERCONNECT system call alters the contents of an interconnect register to a value specified in the call.

NOTE

The RQSETINTERCONNECT system call alters the contents of the specified interconnect register (if it is writeable). It is possible to corrupt the values in interconnect registers by specifying incorrect values.

```
CALL RQ$SET$INTERCONNECT (value, slot$number, reg$number,
                          except$ptr);
```

Input Parameters

value — A BYTE that contains the value to which the specified interconnect register is to be changed.

slot$number — A BYTE that specifies the MULTIBUS II cardslot number of the board on which the indicated interconnect register is located. The following decimal values are valid:

- If in the range 0 to 19, this parameter specifies the indicated iPSB cardslot number.
- If in the range 24 to 29, this parameter specifies an iLBX II cardslot number from 0 to 5, on the host processor board's iLBX II bus.
- If 31, this parameter indicates that you wish to program the contents of an interconnect register on the calling task's processor board. In this case, do not specify the actual cardslot number. Incorrect values could be returned.
- Values 20 to 23, 30, and any value greater than 31 are illegal.

reg$number — A WORD that specifies the interconnect register to which a value is to be written. This value must be in the range 0000H to 01FFH, the defined range of interconnect space. Refer to the board's hardware reference manual to determine the proper register number.

Output Parameter

except$ptr	A POINTER to a WORD to which the Operating System will return the condition code generated for this system call.

Description

The RQSETINTERCONNECT system call allows the contents of a specified interconnect register to be altered dynamically.

NOTE

The Nucleus checks the range validity of the cardslot and register numbers specified in the call, but it does not verify the existence of a board in the specified cardslot nor does it check the read/write permission of the register before it attempts to access the register.

Condition Codes

E$OK	0000H	No exceptional conditions.
ENOTCONFIGURED	0008H	This system call is not part of the present configuration.
E$PARAM	8004H	One or more of the input parameters has an illegal value.

Intel®

INDEX

A

C

D

E

E

F

G

S

T

U

V

W

intel

EXTENDED iRMX® II BASIC I/O SYSTEM CALLS REFERENCE MANUAL

Intel Corporation
3065 Bowers Avenue
Santa Clara, California 95051

PREFACE

This manual documents the system calls of the Basic I/O System, one of the subsystems of the extended iRMX II Operating System. The information provided in this manual is intended as a reference to the system calls and provides detailed descriptions of each call.

READER LEVEL

This manual is intended for programmers who are familiar with the concepts and terminology introduced in the *Extended iRMX II Nucleus User's Guide* and with the PL/M-286 programming language.

CONVENTIONS

System call names appear as headings on the outside upper corner of each page. The first appearance of each system call name is printed in ink; subsequent appearances are in black.

Throughout this manual, system calls are shown using a generic shorthand (such as A$CREATE$FILE instead of RQACREATE$FILE). This convention is used to allow easier alphabetic arrangement of the calls. The actual PL/M-286 external-procedure names must be used in all calling sequences.

You can also invoke the system calls from assembly language, but you must obey the PL/M-286 calling sequences when doing so. For more information on these calling sequences refer to the *Extended iRMX II Programming Techniques Reference Manual.*

Intel®

CONTENTS

Intel®

iRMX® II BASIC I/O SYSTEM CALLS

1.1 INTRODUCTION

The Basic I/O System Calls manual provides a detailed description of each Basic I/O System call, listed alphabetically.

The manual describes the PL/M-286 calling sequences to the Basic I/O System calls.

Basic I/O operations are declared as typed or untyped external procedures for PL/M-286. PL/M-286 programs perform I/O operations by making external procedure calls.

The information for each system call is organized in this order:

- A brief sketch of the effects of the call.
- The PL/M-286 calling sequence for the system call.
- Definitions of the input parameters, if any.
- Definitions of the output parameters, if any.
- A detailed description of the effects of the call.
- The condition codes that can result from using the call, with a description of the possible causes of each condition.

Throughout this manual, PL/M-286 data types, such as BYTE, WORD, and SELECTOR are used. In addition, the extended iRMX II data type TOKEN (always capitalized) is used. If your compiler supports the SELECTOR data type, a TOKEN can be declared literally as SELECTOR or WORD. Because TOKEN is not a PL/M-286 data type, you must declare it to be literally a SELECTOR or a WORD every place you use it. An asterisk (*) is used as a STRUCTURE and ARRAY size indicator. You must substitute a value for the asterisk in STRUCTURE and ARRAY declarations.

The Basic I/O System does not distinguish between upper and lowercase letters. For example, file "xyz" is equal to file "XYZ".

The system call dictionary on these next few pages lists system calls by function rather than alphabetically. This dictionary includes short descriptions and page numbers of the complete descriptions that follow.

1.2 SYSTEM CALL COMMAND DICTIONARY

This dictionary summarizes the Basic I/O System calls by function and, where applicable, indicates the file types to which they apply:

PF	Physical file
SF	Stream file
NF	Named data file
ND	Named directory file

The page reference listed with each call points to the detailed description for the call.

JOB-LEVEL SYSTEM CALLS

DEVICE-LEVEL SYSTEM CALLS

FILE/CONNECTION-LEVEL SYSTEM CALLS

System Call	Function	P F	S F	N F	N D	
A$ATTACH$FILE	Asynchronous attachfile.	*	*	*	*	6
A$CREATE$-DIRECTORY	Asynchronous directory file creation.				*	18
A$CREATE$FILE	Asynchronous data file creation.	*	*	*		23
A$DELETE$CON-NECTION	Asynchronous delete file connection.	*	*	*	*	29
A$DELETE$FILE	Asynchronous data or directory file deletion.			*	*	32

FILE-MODIFICATION SYSTEM CALLS

		P F	S F	N F	N D	
A$CHANGE$ACCESS	Asynchronous change access rights to file.				*	10
A$RENAME$FILE	Asynchronous rename file.				*	72
A$TRUNCATE	Asynchronous truncate file.				*	106

FILE INPUT/OUTPUT SYSTEM CALLS

		P F	S F	N F	N D	
A$CLOSE	Asynchronous close file.	*	*	*	*	16
A$OPEN	Asynchronous open file.	*	*	*	*	57
A$READ	Asynchronous read file.	*	*	*	*	68
A$SEEK	Asynchronous move file pointer.	*		*	*	77
A$UPDATE	Asynchronous finish writing to output device.	*		*	*	109

GET STATUS/ATTRIBUTE SYSTEM CALLS

USER OBJECT SYSTEM CALLS

EXTENSION DATA SYSTEM CALLS

TIME/DATE SYSTEM CALLS

CALLS FOR ACCESSING THE GLOBAL TIME-OF-DAY CLOCK

A$ATTACH$FILE creates a connection to an existing file.

```
CALL RQ$A$ATTACH$FILE(user, prefix, subpath$ptr, resp$mbox,
                      except$ptr);
```

Input Parameters

user	A TOKEN for the user object to be inspected in any access checking that takes place. A SELECTOR$OF(NIL) specifies the default user for the calling task's job. This parameter is ignored when attaching physical or stream files. Access checking does occur for named files.
prefix	A TOKEN for the connection object to be used as the path prefix. A SELECTOR$OF(NIL) specifies the default prefix for the calling task's job.
subpath$ptr	A POINTER to a STRING containing the subpath of the file to be attached. A null string indicates that the new connection is to the file designated by the prefix. The new connection will not be open, regardless of the open mode of the prefix. (This parameter is ignored for physical and stream files.)

Output Parameters

resp$mbox	A TOKEN for the mailbox into which the Basic I/O System places a token for the result object of the call. This result object is a new file connection if the call succeeds or an I/O result segment otherwise (see the *Extended iRMX II Basic I/O System User's Guide*, Appendix C). To ascertain the type of object returned, use the Nucleus system call GET$TYPE. If the object received is an I/O result segment, the calling task should call DELETE$SEGMENT to delete the segment after examining it.
except$ptr	A POINTER to a WORD where the sequential condition code will be returned.

Description

A$ATTACH$FILE creates a connection to an existing file. Once the connection is established, it remains in effect until the connection object is deleted, or until the creating job is deleted. Once attached, the file may be opened, closed, read, written, etc., as many times as desired. A$ATTACH$FILE has no effect on the owner ID or the access list for the file.

Condition Codes

A$ATTACH$FILE returns condition codes at two different times. The code returned to the calling task immediately after invocation of the system call is considered a sequential condition code. A code returned as a result of asynchronous processing is a concurrent condition code. A complete explanation of sequential and concurrent parts of system calls is in the *Extended iRMX II Basic I/O System User's Guide*.

The following list is divided into two parts--one for sequential codes, and one for concurrent codes.

Sequential Condition Codes

The Basic I/O System can return the following condition codes to the word specified by the except$ptr parameter of this system call.

E$OK	0000H	No exceptional conditions.
EDEVOFFLINE	002EH	The prefix parameter in this system call refers to a logical connection. One of the following is true of the device associated with the connection: • It has been physically attached but is now off-line. • It has never been physically attached. (See Appendix E in the *Extended iRMX II Basic I/O User's Guide* for a more detailed explanation.)
E$EXIST	0006H	One of the following is true: • One or more of the following parameters is not a token for an existing object: - The user parameter - The prefix parameter - The resp$mbox parameter • The prefix connection is being deleted.

		• The connection for a remote driver is no longer active.
E$LIMIT	0004H	Processing this call would cause one or more of these limits to be exceeded: • The object limit for this job. • The number of I/O operations that can be outstanding at one time for the user object specified in the call (255 decimal). • The number of I/O operations that can be outstanding at one time for the caller's job (255 decimal). • The number of outstanding I/O operations for a remote connection has been exceeded.
E$MEM	0002H	The memory available to the calling task's job is not sufficient to complete the call.
ENOPREFIX	8022H	The calling task specified a default prefix (prefix argument equals zero), but no default prefix can be found because of one of the following reasons: • When this job was created, a size of zero was specified for its object directory, so the job cannot catalog a default prefix. • The job's directory can have entries but a default prefix is not cataloged there.
ENOUSER	8021H	If the user parameter in this call is not SELECTOR$OF(NIL), the parameter is not a token for a user object.If the user parameter is SELECTOR$OF(NIL), it specifies a default user. But no default user can be found because of one of the following reasons: • When this job was created, a size of zero was specified for its object directory, so the job cannot catalog a default user. • The job's directory can have entries but a default user is not cataloged there. • The object that is cataloged with the name R?IOUSER is not a user object. The name R?IOUSER should be treated as a reserved word.

ENOTCONFIGURED	0008H	This system call is not part of the present configuration.
E$TYPE	8002H	One or more of the following conditions caused this exception: • The prefix parameter is a token for an object that is not of the correct type. It must be either a connection object or a logical device object. (Logical device objects are created by the Extended I/O System.) • The resp$mbox parameter in the call is a token for an object that is not a mailbox.

Concurrent Condition Codes

The Basic I/O System can return the following condition codes in an I/O result segment at the mailbox specified by resp$mbox. After examining the segment, you should delete it to return the memory to the memory pool.

E$OK	0000H	No exceptional conditions.
EDEVDETACHING	0039H	The file specified is on a device that the system is detaching.
E$FNEXIST	0021H	A file in the specified path, or the target file itself, does not exist or is marked for deletion.
E$FTYPE	0027H	The string pointed to by the subpath$ptr parameter contains a filename that should be the name of a directory, but is not. (Except for the last file, each file in a path must be a named directory.)
E$INVALID$FNODE	003DH	The fnode for the specified file is invalid. The file cannot be accessed; you should delete it.
E$IO	002BH	An I/O error occurred, which might have prevented the operation from completing. Examine the unit$status field of the I/O result segment for more information.
EIOMEM	0042H	The memory available to the Basic I/O System job is not sufficient to complete the call.

A$CHANGE$ACCESS changes the access rights to a named data or directory file.

```
CALL RQ$A$CHANGE$ACCESS(user, prefix, subpath$ptr, id, access,
                        resp$mbox, except$ptr);
```

Input Parameters

user	A TOKEN for the user object to be inspected in access checking. A value of SELECTOR$OF(NIL) specifies the default user for the calling task's job.
prefix	A TOKEN for the connection object to be used as the path prefix. A SELECTOR$OF(NIL) specifies the default prefix for the calling task's job.
subpath$ptr	A POINTER to a STRING giving the subpath of the file whose access is to be changed. A null string indicates that the prefix itself designates the desired file.
id	A WORD containing the ID number of the user whose access is to be changed. If this ID does not already exist in the ID-access mask list, it is added. This list may contain a total of three ID-access pairs.
access	A BYTE mask giving the new access rights for the ID. For each bit, a one grants access, and a zero denies it. (Bit 0 is the low-order bit.) For a named data file, the possible bit settings are:

Bit	Meaning
0	Delete
1	Read
2	Append
3	Update
4-7	Reserved (set to 0)

For a named directory file, the possible bit settings are:

Bit	Meaning
0	Delete
1	List
2	Add Entry
3	Change Entry
4-7	Reserved (set to 0)

If zero is specified for the access parameter (that is, no access), the ID specified in the id parameter is deleted from the file's ID-access list and the accessor count field is decremented.

Output Parameters

resp$mbox — A TOKEN for the mailbox that receives an I/O result segment indicating the result of the call (see *Extended iRMX II Basic I/O System User's Guide*, Appendix C). A value of SELECTOR$OF(NIL) means that you do not want to receive an I/O result segment.

If it receives an I/O result segment, the calling task should call DELETE$SEGMENT to delete the segment after examining it.

except$ptr — A POINTER to a WORD where the sequential condition code will be returned.

Description

A$CHANGE$ACCESS system call applies to named files only. This call has no effect on existing connections to the file. It is called to change the access rights to a named data or directory file. Depending on the contents of the "id" and "access" parameters specified in the system call, users may be added to or deleted from the file's ID-access mask list, or the access privileges granted to a particular user may be changed.

NOTE

The caller must be the owner of the file or must have change entry access to the file's parent directory. However, if the owner is "WORLD", that is, 0FFFFH, then any task may change the access mask of the file. If system manager support is configured, user 0 may change the access rights of any file regardless of which user is the owner.

Condition Codes

A$CHANGE$ACCESS returns condition codes at two different times. The code returned to the calling task immediately after invocation of the system call is considered a sequential condition code. A code returned as a result of asynchronous processing is a concurrent condition code. A complete explanation of sequential and concurrent parts of system calls is in the *Extended iRMX II Basic I/O System User's Guide*.

The following list is divided into two parts--one for sequential codes, and one for concurrent codes.

Sequential Condition Codes

The Basic I/O System can return the following condition codes to the word specified by the except$ptr parameter of this system call.

E$OK	0000H	No exceptional conditions.
EDEVOFFLINE	002EH	The prefix parameter in this system call refers to a logical connection. One of the following is true of the device that is associated with the connection: • It has been physically attached but is off-line. • It has never been physically attached. (For example, LOGICAL$ATTACH$DEVICE, an EIOS call, was used. This call does not cause the device to be physically attached until another EIOS call references the logical device object.)
E$EXIST	0006H	At least one of the following is true: • One or more of the following parameters is not a token for an existing object: - The user parameter - The prefix parameter - The response mailbox parameter • The prefix connection is being deleted. • The remote driver connection is no longer active.
E$IFDR	002FH	This system call applies only to named files, but the prefix and subpath parameters specify some other type of file.
E$LIMIT	0004H	Processing this call would cause one or more of these limits to be exceeded:

		• The object limit for this job. • The maximum number of outstanding I/O operations for the user object specified in the call (255 decimal). • The number of I/O operations that can be outstanding at one time for the caller's job (255 decimal). • The number of outstanding I/O operations for a remote file has been exceeded.
E$MEM	0002H	The memory available to the calling task's job is not sufficient to complete this call.
ENOPREFIX	8022H	The calling task specified a default prefix (prefix parameter equals SELECTOR$OF(NIL)). But no default prefix can be found because of one of the following: • When this job was created, a size of SELECTOR$OF(NIL) was specified for its object directory, so the job cannot catalog a default prefix. • The job's directory can have entries but no default prefix is cataloged there.
ENOUSER	8021H	If the user parameter in this call is not SELECTOR$OF(NIL), then the parameter is not a token for a user object.If the user parameter is SELECTOR$OF(NIL), it specifies a default user. But no default user can be found because of one of the following reasons: • When this job was created, a size of zero was specified for its object directory, so the job cannot catalog a default user. • The job's directory can have entries but no default user is cataloged there. • The object which is cataloged with the name R?IOUSER is not a user object. The name R?IOUSER should be treated as a reserved word.
ENOTCONFIGURED	0008H	This system call is not part of the present configuration.

E$PATHNAME$-SYNTAX	003EH	The specified pathname contains invalid characters.
E$SUPPORT	0023H	The connection was not created by this job.
E$TYPE	8002H	One or more of the following conditions caused this exception: • The user token designates a connection of the wrong type. • The prefix parameter is a token for an object that is not of the correct type. It must be either a connection object or a logical device object. (Logical device objects are created by the Extended I/O System.) • The resp$mbox parameter in the call is a token for an object that is not a mailbox.

Concurrent Condition Codes

The Basic I/O System can return the following condition codes in an I/O result segment at the mailbox specified by resp$mbox. After examining the segment, you should delete it.

E$OK	0000H	No exceptional conditions.
EDEVDETACHING	0039H	The file specified is on a device that the system is detaching.
E$FACCESS	0026H	The user object in the parameter list is not the owner of the specified file, nor does it have "change entry" access to the parent directory.
E$FNEXIST	0021H	A file in the specified path, or the target file itself, does not exist or is marked for deletion.
E$FTYPE	0027H	The string pointed to by the subpath$ptr parameter contains a filename which should be the name of a directory, but is not. (Except for the last file, each file in a path must be a named directory.)
E$INVALID$FNODE	003DH	The fnode for the specified file is invalid. The file cannot be accessed; you should delete it.
E$IO	002BH	An I/O error occurred which might have prevented the operation from completing. Examine the unit$status field of the I/O result segment for more information. For information on IORS structures, see the *Extended iRMX II Device Drivers User's Guide*.

EIOMEM	0042H	The memory available to the Basic I/O System job is not sufficient to complete this call.
E$SUPPORT	0023H	The call attempted to add another access ID to the list of access ID's. The access list already contained the limit of three such ID's.

A$CLOSE closes an open file connection.

```
CALL RQ$A$CLOSE(connection, resp$mbox, except$ptr);
```

Input Parameter

connection	A TOKEN for the file connection to be closed.

Output Parameters

resp$mbox	A TOKEN for the mailbox that receives an I/O result segment indicating the result of the call (see Appendix C in the *Extended iRMX II Basic I/O User's Guide*). A value of SELECTOR$OF(NIL) means that you do not want to receive an I/O result segment. If it receives an I/O result segment, the calling task should call DELETE$SEGMENT to delete the segment after examining it.
except$ptr	A POINTER to a WORD where the sequential condition code will be returned.

Description

The A$CLOSE system call closes an open file connection. It is called when the application needs to change the open mode or shared status of the connection. The Basic I/O System will not close the connection until all existing I/O requests for the connection have been satisfied. In addition, the Basic I/O System will not send a response to the response mailbox until the file is closed.

Condition Codes

A$CLOSE returns condition codes at two different times. The code returned to the calling task immediately after invocation of the system call is considered a sequential condition code. A code returned as a result of asynchronous processing is a concurrent condition code. A complete explanation of sequential and concurrent parts of system calls is in the *Extended iRMX II Basic I/O System User's Guide*.

The following list is divided into two parts--one for sequential codes, and one for concurrent codes.

Sequential Condition Codes

The Basic I/O System can return the following condition codes to the word specified by the except$ptr parameter of this system call.

E$OK	0000H	No exceptional conditions.
E$EXIST	0006H	At least one of the following is true: • One or more of the following parameters is not a token for an existing object: - The connection parameter - The resp$mbox parameter • The connection is being deleted. • The connection for a remote driver is no longer active.
E$LIMIT	0004H	At least one of the following is true. • The calling task's job has already reached its object limit. • The number of outstanding I/O operations for a remote connection has been exceeded.
E$MEM	0002H	The memory available to the calling task's job is not sufficient to complete this call.
ENOTCON-FIGURED	0008H	This system call is not part of the present configuration.
E$SUPPORT	0023H	The connection was not created by this job.
E$TYPE	8002H	At least one of the following is true: • The connection parameter is a token for an object that is not a connection. • The resp$mbox parameter is a token for an object that is not a mailbox.

Concurrent Condition Codes

The Basic I/O System can return the following condition codes in an I/O result segment at the mailbox specified by resp$mbox. After examining the segment, you should delete it.

E$OK	0000H	No exceptional conditions.
E$CONN$NOT$OPEN	0034H	The specified connection is not open.
E$IO	002BH	An I/O error occurred, but the operation was successful anyway.

A$CREATE$DIRECTORY creates a directory file.

```
CALL RQ$A$CREATE$DIRECTORY(user, prefix, subpath$ptr, access,
                           resp$mbox, except$ptr);
```

Input Parameters

user — A TOKEN for the user object of the new directory's owner. The user object is inspected to make sure the caller has proper access to the new directory's parent. A SELECTOR$OF(NIL) specifies the default user for the calling task's job.

prefix — A TOKEN for the connection to be used as the path prefix. A SELECTOR$OF(NIL) specifies the default prefix for the calling task's job.

subpath$ptr — A POINTER to a STRING containing the subpath of the directory to be created. The subpath string must not be null, and it must point to an unused location in the directory tree.

access — A BYTE mask giving the owner's initial access rights to the directory. For each bit in the mask, a one grants access and a zero denies it. The possible bit settings are:

Bit	Meaning
0	Delete
1	List
2	Add Entry
3	Change Entry
4-7	Reserved (set to 0)

Output Parameters

resp$mbox — A TOKEN for the mailbox that receives the result object of this call. This result object is a directory file connection if the call succeeded, or an I/O result segment otherwise (see Appendix C in the *Extended iRMX II Basic I/O System User's Guide*). To determine the type of object returned, use the Nucleus system call GET$TYPE. If the object received is an I/O result segment, the calling task should call DELETE$SEGMENT to delete the segment after examining it.

except$ptr — A POINTER to a WORD where the sequential condition code will be returned.

Description

The A$CREATE$DIRECTORY system call is applicable to named directory files only. When called, it creates a new directory file and returns a token for the new file connection. This system call cannot be used to create a connection to an existing directory. To attach to an existing file you should use the A$ATTACH$FILE system call.

NOTE

The caller must have add-entry access to the parent of the new directory.

Condition Codes

A$CREATE$DIRECTORY returns condition codes at two different times. The code returned to the calling task immediately after invocation of the system call is considered a sequential condition code. A code returned as a result of asynchronous processing is a concurrent condition code. A complete explanation of sequential and concurrent parts of system calls is in the *Extended iRMX II Basic I/O System User's Guide*.

The list on the following pages is divided into two parts--one for sequential codes, and one for concurrent codes.

Sequential Condition Codes

The Basic I/O System can return the following condition codes to the word specified by the except$ptr parameter of this system call.

E$OK	0000H	No exceptional conditions.
EDEVOFF$LINE	002EH	The prefix parameter in this system call refers to a logical connection. One of the following is true of the device that is associated with the connection: • It has been physically attached but is now off-line. • It has never been physically attached. (See *Extended iRMX II Basic I/O System User's Guide*, Appendix E for a more detailed explanation.)
E$EXIST	0006H	At least one of the following is true:

		• One or more of the following parameters is not a token for an existing object: - The user parameter - The prefix parameter - The response mailbox parameter • The prefix connection is being deleted. • The connection for a remote driver is no longer active.
E$IFDR	002FH	This system call applies only to named directory files, but the prefix and subpath parameters specify some other type of file.
E$LIMIT	0004H	Processing this call would cause one or more of these limits to be exceeded: • The object limit for this job. • The number of I/O operations that can be outstanding at one time for the user object specified in the call (255 decimal). • The number of I/O operations that can be outstanding at one time for the caller's job (255 decimal). • The number of outstanding I/O operations for a remote connection has been exceeded.
E$MEM	0002H	The memory available to the calling task's job is not sufficient to complete this call.
ENOPREFIX	8022H	The task specified a default prefix (prefix parameter equals SELECTOR$OF(NIL)). But no default prefix can be found because of one or more of the following reasons: • When this job was created, a size of zero was specified for its object directory, so the job cannot catalog a default prefix. • The job's directory can have entries but no default prefix is cataloged there.
ENOUSER	8021H	If the user parameter in this call is not SELECTOR$OF(NIL), then the parameter is not a user object. If the user parameter is SELECTOR$OF(NIL), it specifies a default user.

		But no default user can be found because of one of the following reasons: • When this job was created, a size of zero was specified for its object directory, so the job cannot catalog a default user. • The job's directory can have entries but no default user is cataloged there. • The object that is cataloged with the name R?IOUSER is not a user object. The name R?IOUSER should be treated as a reserved word.
ENOTCON-FIGURED	0008H	This system call is not part of the present configuration.
E$PATHNAME$-SYNTAX	003EH	The specified path name contains invalid characters.
E$TYPE	8002H	At least one of the following is true: • The prefix parameter is a token for an object that is not of the correct type. It must be either a connection object or a logical device object. (Logical device objects are created by the Extended I/O System.) • The resp$mbox parameter in the call is a token for an object that is not a mailbox.

Concurrent Condition Codes

The Basic I/O System can return the following condition codes in an I/O result segment at the mailbox specified by resp$mbox. After examining the segment, you should delete it.

E$OK	0000H	No exceptional conditions.
EDEVDETACHING	0039H	The file specified is on a device that the system is detaching.
E$FACCESS	0026H	The user object in the parameter list is not qualified for "add-entry" access to the parent directory.
E$FEXIST	0020H	A file with the specified path name already exists.
E$FNEXIST	0021H	A file in the specified path does not exist or is marked for deletion.
E$FNODE$LIMIT	003FH	The volume already contains the maximum number of files. No more fnodes are available for new files.

E$FTYPE	0027H	The string pointed to by the subpath$ptr parameter contains a filename which should be the name of a directory, but is not. (Except for the last file, each file in a path must be a named directory.)
E$INVALID$FNODE	003DH	The fnode for the specified file (or for a directory in the file's path) is invalid. The file with the invalid fnode cannot be accessed; you should delete it.
E$IO	002BH	An I/O error occurred which might have prevented the operation from completing. Examine the unit$status field of the I/O result segment for more information.
EIOMEM	0042H	The memory available to the Basic I/O System job is not sufficient to complete this call.
E$SPACE	0029H	The volume is full.
E$SUPPORT	0023H	The Basic I/O System is not configured to support space allocation.

A$CREATE$FILE creates a physical, stream, or named file.

```
CALL RQ$A$CREATE$FILE(user, prefix, subpath$ptr, access,
        granularity, size, must$create, resp$mbox, except$ptr);
```

Input Parameters

user

A TOKEN for the user object of the owner of the new file. It also furnishes the user ID for any access checking that might occur. A SELECTOR$OF(NIL) specifies the default user for the calling task's job. This parameter does not apply to physical or stream files.

prefix

A TOKEN for a device or file connection. The file created by this call is of the type (physical, stream, or named) that is associated with this parameter. A SELECTOR$OF(NIL) for this parameter specifies the default prefix for the job.

For stream files, if the prefix is a device connection, a new stream file is created. If the prefix is a file connection, a new file connection to the same stream file is created.

For named files, the prefix acts as the starting point in a directory tree scan.

subpath$ptr

A POINTER to a STRING containing the subpath for the named file being created. This parameter does not apply to physical and stream files.

Entering NIL for this parameter, when using a named file driver, causes an unnamed file to be created. This file is automatically deleted when the last connection to it is deleted.

access

A BYTE mask giving the owner's initial access rights to the new file. For each bit, a one grants access and a zero denies it. (Bit 0 is the low-order bit.)

Bit	Meaning
0	Delete
1	Read
2	Append
3	Update
4-7	Reserved (set to 0)

This parameter does not apply to physical or stream files.

granularity — A WORD giving the granularity of the file being created. This is the size (in bytes) of each logical block of volume space to be allocated to the file. The value specified in this parameter is rounded up, if necessary, to a multiple of the volume granularity. Note that a contiguous file can become noncontiguous when it is extended.

The granularity parameter can have the following values:

0	Same as volume granularity
FFFFH	The file must be contiguous
Other	Number of bytes per allocation

When a contiguous file is extended, space is allocated in volume-granularity units. If "Other" is specified, a multiple of 1024 bytes is recommended. This parameter is ignored for physical and stream files.

size — A DWORD giving the number of bytes initially reserved for the file. For stream files, this value must equal zero. If you make this value greater than zero, for stream files the reserved space may contain unknown data. For physical files, this parameter is ignored.

must$create — A BYTE with values of 1 for TRUE, or 0 for FALSE. Only the least significant bit is checked. This BYTE determines the handling of input paths designating an existing file (see following Description). This parameter applies only to named files.

Output Parameters

resp$mbox — A TOKEN for the mailbox that receives the result object of this call. This result object is a new file connection if the call succeeded; otherwise, it is an I/O result segment (see Appendix C in the *Extended iRMX II Basic I/O System User's Guide*). To determine the type of object returned, use the Nucleus system call GET$TYPE.

If the object received is an I/O result segment, the calling task should call DELETE$SEGMENT to delete the segment after examining it.

except$ptr — A POINTER to a WORD where the sequential condition code will be returned.

Description

The A$CREATE$FILE system call creates a physical, stream, or named data file and returns a token for the new file connection. If a named file designated by the prefix and subpath parameters already exists, one of the following occurs:

- Error: If the "must$create" parameter is TRUE (0FFH), an error condition code (E$FEXIST) is returned.
- Truncate File: If the "must$create" parameter is FALSE (0) and the path designates an existing data file, a new connection to that file is returned (that is, A$CREATE$FILE acts like A$ATTACH$FILE). In this case, the file is truncated or expanded according to the "size" parameter, so data in the file might be lost. As in the case of A$ATTACH$FILE, the file's owner ID and access list are unchanged.
- Temporary File Created: If the "must$create" parameter is FALSE (0), and the path designates an existing directory file or device, an unnamed temporary file is created on the corresponding device. This file is deleted automatically when the last connection to it is deleted. Because this file is created without a path, it can be accessed only through a connection.

  Any task can create a temporary file by referring to any directory. This is true because temporary files are not listed as ordinary entries in the directory, so no add-entry access is required.

Many of the parameters specified in the A$CREATE$FILE call do not apply to physical and stream files. In these cases, the parameter is ignored.

NOTE

The caller must have add-entry access to the parent directory of the new named file.

Condition Codes

A$CREATE$FILE returns condition codes at two different times. The code returned to the calling task immediately after invocation of the system call is considered a sequential condition code. A code returned as a result of asynchronous processing is a concurrent condition code. A complete explanation of sequential and concurrent parts of system calls is in the *Extended iRMX II Basic I/O System User's Guide*.

The following list is divided into two parts--one for sequential codes, and one for concurrent codes.

Sequential Condition Codes

The Basic I/O System can return the following condition codes to the word specified by the except$ptr parameter of this system call.

E$OK	0000H	No exceptional conditions.
EDEVOFF$LINE	002EH	The prefix parameter in this system call refers to a logical connection. One of the following is true of the device associated with the connection: • It has been physically attached but is now off-line. • It has never been physically attached. (See *Extended iRMX II Basic I/O System User's Guide*, Appendix E for a more detailed explanation.)
E$EXIST	0006H	At least one of the following is true: • One or more of the following parameters is not a token for an existing object: - The user parameter - The prefix parameter - The resp$mbox parameter • The prefix connection is being deleted. • The connection for a remote driver is no longer active.
E$LIMIT	0004H	The number of outstanding I/O operations for a remote connection has been exceeded.
E$MEM	0002H	The memory available to the calling task's job is not sufficient to complete this call.
ENOPREFIX	8022H	The call specified a default prefix (prefix argument equals SELECTOR$OF(NIL)). But no default prefix can be found because of one of the following reasons: • When this job was created, a size of SELECTOR$OF(NIL) was specified for its object directory, so the job cannot catalog a default prefix. • The job's directory can have entries but a default prefix is not cataloged there.

ENOUSER	8021H	If the user parameter in this call is not SELECTOR$OF(NIL), then the parameter is not a token for a user object. If the user parameter is SELECTOR$OF(NIL), it specifies a default user. But no default user can be found because of one of the following reasons: • When this job was created, a size of zero was specified for its object directory, so the job cannot catalog a default user. • The job's directory can have entries but a default user is not cataloged there. • The object that is cataloged with the name R?IOUSER is not a user object. Another task cataloged an object (not a user object) under the name R?IOUSER.
ENOTCON-FIGURED	0008H	This system call is not part of the present configuration.
E$PATHNAME$-SYNTAX	003EH	The specified path name contains invalid characters.
E$TYPE	8002H	At least one of the following is true: • The prefix parameter is a token for an object that is not of the correct type. It must be either a connection object or a logical device object. (Logical device objects are created by the Extended I/O System.) • The resp$mbox parameter in the call is a token for an object that is not a mailbox.

Concurrent Condition Codes

The Basic I/O System can return the following condition codes in an I/O result segment at the mailbox specified by resp$mbox. After examining the segment, you should delete it.

E$OK	0000H	No exceptional conditions.
EDEVDETACHING	0039H	The file specified is on a device that the system is detaching.
E$FACCESS	0026H	One of the following is true: • No file with the specified pathname exists, and the specified user object does not have "add-entry" access to the parent directory.

		• A file with the specified pathname exists, but the specified user object does not have "update" access to the file.
E$FEXIST	0020H	The "must$create" parameter in the call is TRUE, and the file already exists. (See the Description section.)
E$FNEXIST	0021H	A file in the specified path does not exist or is marked for deletion.
E$FNODE$LIMIT	003FH	The volume already contains the maximum number of files. No more fnodes are available for new files.
E$FTYPE	0027H	The string pointed to by the subpath$ptr parameter contains a filename which should be the name of a directory, but is not. (Except for the last file, each file in a path must be a named directory.)
E$INVALID$FNODE	003DH	The fnode for the specified file (or for a directory in the file's path) is invalid. The file with the invalid fnode cannot be accessed; you should delete it.
E$IO	002BH	An I/O error occurred which might have prevented the operation from completing. Examine the unit$status field of the I/O result segment for more information.
EIOMEM	0042H	The memory available to the Basic I/O System job is not sufficient to complete this call.
E$SHARE	0028H	The file this call is attempting to create already exists and is open. It was opened with the characteristic "no share with writers." (See the A$OPEN call in this manual.)
E$SPACE	0029H	The volume is full.
E$SUPPORT	0023H	One of the following is true: • The file exists and the must$create parameter is FALSE. When the Basic I/O System was configured, an option was chosen that prevented this combination, so that files could not be automatically truncated to zero size. See the Description section. • The Basic I/O System is not configured to allow space allocation on volumes.

A$DELETE$CONNECTION deletes a named file connection created by A$CREATE$FILE, A$CREATE$DIRECTORY, or A$ATTACH$FILE.

```
CALL RQ$A$DELETE$CONNECTION(connection, resp$mbox, except$ptr);
```

Input Parameter

connection	A TOKEN for the file connection to be deleted.

Output Parameters

resp$mbox	A TOKEN for the mailbox that receives an I/O result segment indicating the result of the call (see Appendix C in the *Extended iRMX II Basic I/O System User's Guide*). A value of SELECTOR$OF(NIL) means that you do not want to receive an I/O result segment. If it receives an I/O result segment, the calling task should call DELETE$SEGMENT to delete the segment after examining it.
except$ptr	A POINTER to a WORD where the sequential condition code will be returned.

Description

The A$DELETE$CONNECTION system call deletes a connection object. It also deletes the associated file if both of the following are true:

- The file is already marked for deletion (by a previous A$DELETE$FILE call) or is an unnamed file.
- The specified connection is the only connection to the file.

If a connection is open when A$DELETE$CONNECTION is called, it is closed before being deleted.

NOTE

Connections should be deleted when no longer needed.

Condition Codes

A$DELETE$CONNECTION returns condition codes at two different times. The code returned to the calling task immediately after invocation of the system call is considered a sequential condition code. A code returned as a result of asynchronous processing is a concurrent condition code. A complete explanation of sequential and concurrent parts of system calls is in the *Extended iRMX II Basic I/O System User's Guide*.

The following list is divided into two parts--one for sequential codes, and one for concurrent codes.

Sequential Condition Codes

The Basic I/O System can return the following condition codes to the word specified by the except$ptr parameter of this system call.

	E$OK	0000H No exceptional conditions.
E$EXIST	0006H	At least one of the following is true: • One or more of the following parameters is not a token for an existing object: - The connection parameter - The resp$mbox parameter • The connection is being deleted. • The connection for a remote driver is no longer active.
E$LIMIT	0004H	The calling task's job has already reached its object limit.
E$MEM	0002H	The memory available to the calling task's job is not sufficient to complete this call.
ENOTCON-FIGURED	0008H	This system call is not part of the present configuration.
ENOTFILE$CONN	0032H	The connection parameter is a device connection, not a file connection.
E$SUPPORT	0023H	The specified connection was not created by this job.
E$TYPE	8002H	One or more of the following is a token for an object that is not of the correct type: • The connection parameter. • The resp$mbox parameter.

Concurrent Condition Codes

The Basic I/O System can return the following condition codes in an I/O result segment at the mailbox specified by resp$mbox. After examining the segment, you should delete it.

E$OK	0000H	No exceptional conditions.
E$IO	002BH	An I/O error occurred, however the connection was deleted.

A$DELETE$FILE marks a file for deletion.

```
CALL RQ$A$DELETE$FILE(user, prefix, subpath$ptr, resp$mbox,
                      except$ptr);
```

Input Parameters

user	A TOKEN for the user object to be inspected in access checking. A SELECTOR$OF(NIL) specifies the default user for the calling task's job. This parameter does not apply to stream files.
prefix	A TOKEN for the connection object to be used as the path prefix. A SELECTOR$OF(NIL) specifies the default prefix for the calling task's job.
subpath$ptr	A POINTER to a STRING giving the subpath for the file being deleted. A null string indicates that the prefix itself designates the desired file. In this instance, the user parameter is ignored, since access checking was already performed when the file was attached. This parameter does not apply to stream files.

Output Parameters

resp$mbox	A TOKEN for a mailbox that receives an I/O result segment (see *Extended iRMX II Basic I/O System User's Guide*, Appendix C) when the file is marked for deletion. The file will not actually be deleted until all connections to the file are deleted, as explained under the Description below. A value of SELECTOR$OF(NIL) means that you do not want to receive an I/O result segment. If it receives an I/O result segment, the calling task should call DELETE$SEGMENT to delete the segment after examining it.
except$ptr	A POINTER to a WORD where the sequential condition code will be returned.

Description

The A$DELETE$FILE system call applies to stream and named files only. When called, it marks the designated file for deletion and removes the file's entry from the parent directory. The entry is removed immediately, but the file is not actually deleted until all connections to the file have been severed (by A$DELETE$CONNECTION calls). Directory files cannot be deleted unless they are empty.

NOTE

The caller must have delete access to the file.

Condition Codes

A$DELETE$FILE returns condition codes at two different times. The code returned to the calling task immediately after invocation of the system call is considered a sequential condition code. A code returned as a result of asynchronous processing is a concurrent condition code. A complete explanation of sequential and concurrent parts of system calls is in the *Extended iRMX II Basic I/O System User's Guide*.

The following list is divided into two parts--one for sequential codes, and one for concurrent codes.

Sequential Condition Codes

The Basic I/O System can return the following condition codes to the word specified by the except$ptr parameter of this system call.

E$OK	0000H	No exceptional conditions.
EDEVOFF$LINE	002EH	The prefix parameter in this system call refers to a logical connection. One of the following is true of the device that is associated with the connection: • It has been physically attached but is now off-line. • It has never been physically attached. (See Appendix E in the *Extended iRMX II Basic I/O System User's Guide* for a more detailed explanation.)
E$EXIST	0006H	At least one of the following is true: • One or more of the following parameters is not a token for an existing object: - The user parameter - The prefix parameter - The response mailbox parameter • The prefix connection is being deleted. • The connection for a remote driver is no longer active.

E$IFDR	002FH	This system call applies only to named or stream files, but the prefix and subpath parameters specified a physical file.
E$LIMIT	0004H	Processing this call would exceed one or more of the following limits: • The object limit for this job. • The number of I/O operations that can be outstanding at one time for the user object specified in the call (255 decimal). • The number of I/O operations that can be outstanding at one time for the caller's job (255 decimal). • The number of outstanding I/O operations for a remote connection has been exceeded.
E$MEM	0002H	The memory available to the calling task's job is not sufficient to complete this call.
ENOPREFIX	8022H	The call specified a default prefix (prefix argument equals SELECTOR$OF(NIL)). But no default prefix can be found because of one of the following reasons: • When this job was created, a size of SELECTOR$OF(NIL) was specified for its object directory, so the job cannot catalog a default prefix. • The job's directory can have entries but no default prefix is cataloged there.
ENOUSER	8021H	If the user parameter in this call is not SELECTOR$OF(NIL), then the problem is that the parameter is not a token for a user object. If the user parameter is SELECTOR$OF(NIL), it specifies a default user. But no default user can be found because of one of the following reasons: • When this job was created, a size of zero was specified for its object directory, so the job cannot catalog a default user. • The job's directory can have entries but no default user is cataloged there.

		• The object that is cataloged with the name R?IOUSER is not a user object. The name R?IOUSER should be treated as a reserved word.
ENOTCON-FIGURED	0008H	This system call is not part of the present configuration.
E$PATHNAME$ SYNTAX	003EH	The specified path name contains invalid characters.
E$SUPPORT	0023H	The specified connection was not created by this job.
E$TYPE	8002H	At least one of the following is true: • The prefix parameter is a token for an object that is not of the correct type. It must be either a connection object or a logical device object. (Logical device objects are created by the Extended I/O System.) • The resp$mbox parameter in the call is a token for an object that is not a mailbox.

Concurrent Condition Codes

The Basic I/O System can return the following condition codes in an I/O result segment at the mailbox specified by resp$mbox. After examining the segment, you should delete it.

E$OK	0000H	No exceptional conditions.
EDEVDETACHING	0039H	The file specified is on a device that the system is detaching.
EDIRNOT$EMPTY	0031H	The call is attempting to delete a directory containing entries.
E$FACCESS	0026H	At least one of the following is true: • The user object does not have delete access to the file. • The call attempted to delete the root directory or a bit map file.
E$FNEXIST	0021H	A file in the specified path, or the target file itself, does not exist or is marked for deletion.
E$FTYPE	0027H	The string pointed to by the subpath$ptr parameter contains a string that should be the name of a directory, but is not. (Except for the last file, each file in a pathname must be a named directory.)

E$IO	002BH	An I/O error occurred which might have prevented the operation from completing. Examine the unit$status field of the I/O result segment for more information.
EIOMEM	0042H	The memory available to the Basic I/O System is not sufficient to complete the call.

AGETCONNECTION$STATUS returns information about a file connection.

```
CALL RQ$A$GET$CONNECTION$STATUS(connection, resp$mbox, except$ptr);
```

Input Parameter

connection — A TOKEN for the file connection whose status is desired.

Output Parameters

resp$mbox — A TOKEN for the mailbox that is to receive a connection-status segment. The calling task is responsible for deleting the connection-status segment after examining it.

The information in this segment is structured as follows:

```
DECLARE conn$status STRUCTURE(
                    status          WORD,
                    file$driver     BYTE,
                    flags           BYTE,
                    open$mode       BYTE,
                    share$mode      BYTE,
                    file$ptr        DWORD,
                    access          BYTE);
```

These fields are interpreted as follows:

status — A condition code giving the outcome of the status-fetch operation. If this code is not E$OK, the remaining fields must be considered invalid.

file$driver — Tells the type of file driver to which this connection is attached. Possible values are:

Value	Type
1	Physical
2	Stream
4	Named
5	Remote

flags — Contains two flag bits. If bit 1 is set to one, this connection is active and can be opened. If bit 2 is set, this connection is a device connection. (Bit 0 is the low-order bit.)

open$mode — The mode established when this connection was opened. Possible values are:

- 0 Connection is closed
- 1 Open for reading
- 2 Open for writing
- 3 Open for reading and writing

share$mode — The sharing mode established when this connection was opened. Possible values are:

- 0 Private use only
- 1 Share with readers only
- 2 Share with writers only
- 3 Share with all users

file$ptr — The current byte location of the file pointer for this connection.

access — The access rights for this connection. For each bit, a one grants access and a zero denies it. (Bit 0 is the low-order bit.)

Bit	Data File	Directory
0	Delete	Delete
1	Read	List
2	Append	Add Entry
3	Update	Change Entry
4-7	Reserved	Reserved

except$ptr — A POINTER to a WORD where the sequential condition code will be returned.

Description

The AGETCONNECTION$STATUS system call returns a segment containing status information about a file connection.

Condition Codes

AGETCONNECTION$STATUS returns condition codes at two different times. The code returned to the calling task immediately after invocation of the system call is considered a sequential condition code. A code returned as a result of asynchronous processing is a concurrent condition code. A complete explanation of sequential and concurrent parts of system calls is in the *Extended iRMX II Basic I/O System User's Guide*.

The following list is divided into two parts--one for sequential codes, and one for concurrent codes.

Sequential Condition Codes

The Basic I/O System can return the following condition codes to the word specified by the except$ptr parameter of this system call.

E$OK	0000H	No exceptional conditions.
E$EXIST	0006H	At least one of the following is true: • One or more of the following parameters is not a token for an existing object: - The connection parameter - The resp$mbox parameter • The connection is being deleted. • The connection for a remote driver is no longer active.
E$LIMIT	0004H	At least one of the following is true: • The calling task's job has already reached its object limit. • The number of outstanding I/O operations for a remote connection has been exceeded.
E$MEM	0002H	The memory available to the calling task's job is not sufficient to complete the call.
ENOTCON-FIGURED	0008H	This system call is not part of the present configuration.
E$SUPPORT	0023H	The specified connection parameter is not valid in this system call because the connection was not created by this job.
E$TYPE	8002H	At least one of the following is true: • The connection parameter is a token for an object that is not a connection. • The resp$mbox parameter is a token for an object that is not a mailbox.

Concurrent Condition Codes

The Basic I/O System returns one of the following condition codes in an I/O result segment at the mailbox specified by resp$mbox. You are responsible for deleting this segment.

E$OK	0000H	No exceptional conditions.

E$IO	002BH	An I/O error occurred, which might or might not have prevented the operation from being completed. Examine the unit$status field of the I/O result segment for more information.

AGETDIRECTORY$ENTRY returns the file name associated with a named directory file entry.

```
CALL RQ$A$GET$DIRECTORY$ENTRY(connection, entry$num, resp$mbox,
                              except$ptr);
```

Input Parameters

connection — A TOKEN for the directory file with the desired entry.

entry$num — A WORD giving the entry number of the desired file name. Entries within a directory are numbered sequentially starting from zero. The E$EMPTY$ENTRY condition code will be returned if there is no entry associated with this number.

Output Parameters

resp$mbox — A TOKEN for the mailbox that will receive a directory-entry segment. The task making the AGETDIRECTORY$ENTRY call is responsible for deleting this segment after examining it.

Information in this segment is structured as follows:

```
DECLARE dir$entry$info  STRUCTURE(
                                  status  WORD,
                                  name (14)   BYTE);
```

where

status — Indicates how the operation was completed. EOK, EEMPTY$ENTRY, and E$DIR$END condition codes all indicate successful completion.

name — File name contained in the specified entry, padded with blanks. This field is valid only if status = E$OK.

except$ptr — A POINTER to a WORD where the sequential condition code will be returned.

AGETDIRECTORY$ENTRY

Description

The AGETDIRECTORY$ENTRY system call applies to named files only. When called, it returns the file name associated with a specified directory entry. This name is a single subpath component for a file whose parent is the designated directory. As an alternative to using this system call, an application task can open and read a directory file.

NOTE

The caller must have display access to the designated directory.

Condition Codes

AGETDIRECTORY$ENTRY returns condition codes at two different times. The code returned to the calling task immediately after invocation of the system call is considered a sequential condition code. A code returned as a result of asynchronous processing is a concurrent condition code. A complete explanation of sequential and concurrent parts of system calls is in the *Extended iRMX II Basic I/O System User's Guide*.

The following list is divided into two parts--one for sequential codes, and one for concurrent codes.

Sequential Condition Codes

The Basic I/O System can return the following condition codes to the word specified by the except$ptr parameter of this system call.

E$OK	0000H	No exceptional conditions.
E$EXIST	0006H	At least one of the following is true: • One or more of the following parameters is not a token for an existing object: - The connection parameter - The resp$mbox parameter • The connection is being deleted.
E$IFDR	002FH	This system call applies only to named directories, but the connection parameter specifies another type of file.
E$LIMIT	0004H	The calling task's job has already reached its object limit.
E$MEM	0002H	The memory available to the calling task's job is not sufficient to complete this call.

ENOTCON-FIGURED	0008H	This system call is not part of the present configuration.
E$SUPPORT	0023H	The specified connection was not created by this job.
E$TYPE	8002H	At least one of the following is true: • The connection parameter is a token for an object that is not a connection. • The resp$mbox parameter is a token for an object that is not a mailbox.

Concurrent Condition Codes

The Basic I/O System can return the following condition codes in an I/O result segment at the mailbox specified by resp$mbox. After examining the segment, you should delete it.

E$OK	0000H	No exceptional conditions.
EDIREND	0025H	The entry$num parameter is greater than the number of entries in the directory.
E$EMPTY$ENTRY	0024H	The file entry designated in the call is empty.
E$FACCESS	0026H	The specified connection is not qualified for "display" access to the directory.
E$FTYPE	0027H	The specified connection does not refer to a directory.
E$IO	002BH	An I/O error occurred which might have prevented the operation from completing. Examine the unit$status field of the I/O result segment for more information.

The AGETEXTENSION$DATA system call returns extension data stored with a Basic I/O System file.

```
CALL RQ$A$GET$EXTENSION$DATA(connection, resp$mbox, except$ptr);
```

Input Parameters

connection — A TOKEN of a connection to a file whose extension data is desired.

resp$mbox — A TOKEN for the mailbox that will receive a segment containing the named file-status information. The calling task is responsible for deleting this segment after examining it.

Structure of the named file-status information is as follows:

```
DECLARE ext$data$seg STRUCTURE (
                        status      WORD,
                        count       BYTE,
                        info(*)     BYTE);
```

These fields are interpreted as follows:

status — A condition code indicating the outcome of the status-fetch operation. If this code is not E$OK, the remaining fields must be considered invalid.

count — A number (from 0 to 255 decimal) indicating the number of bytes returned.

info — The extension data.

Output Parameter

except$ptr — A POINTER to a WORD where the sequential condition code will be returned.

Description

Associated with each file created through the Basic I/O System is a file descriptor containing information about the file. Some of that information is used by the Basic I/O System and can be accessed by tasks through the AGETFILE$STATUS system call. Up to 255 additional bytes of the file descriptor, known as extension data, are available for use by Operating System extensions. OS extensions can write extension data by using ASETEXTENSION$DATA and they can read extension data by using AGETEXTENSION$DATA.

When a task calls AGETEXTENSION$DATA, it specifies a response mailbox to which the system returns a segment with the extension data. The information is located in the low-memory portion of the segment. AGETEXTENSION$DATA can only be applied to connections created via the named file driver.

Condition Codes

AGETEXTENSION$DATA can return condition codes at two different times. The code returned to the calling task immediately after invocation of the system call is considered a sequential condition code. A code returned as a result of asynchronous processing is a concurrent condition code. A complete explanation of sequential and concurrent parts of system calls is in the *Extended iRMX II Basic I/O System User's Guide*.

The following list is divided into two parts--one for sequential codes and one for concurrent codes.

Sequential Condition Codes

The Basic I/O System can return the following condition codes to the word specified by the except$ptr parameter of this system call.

E$OK	0000H	No exceptional conditions.
E$EXIST	0006H	At least one of the following is true: • One or more of the following parameters is not a token for an existing object: - The connection parameter - The resp$mbox parameter • The connection is being deleted. • The connection for a remote driver is no longer active.
E$IFDR	002FH	This system call applies only to named files, but the prefix and subpath parameters specify another type of file.

E$LIMIT	0004H	At least one of the following is true: • The calling task's job has already reached its object limit. • The number of outstanding I/O operations for a remote connection has been exceeded.
E$MEM	0002H	The memory available to the calling task's job is not sufficient to complete the call.
ENOTCON-FIGURED	0008H	This system call is not part of the present configuration.
E$SUPPORT	0023H	The specified connection was not created by this job.
E$TYPE	8002H	At least one of the following is true: • The connection parameter is a token for an object that is not a connection. • The resp$mbox parameter is a token for an object that is not a mailbox.

Concurrent Condition Codes

The Basic I/O System will return one of the following codes in an I/O result segment at the mailbox specified by resp$mbox. After examining the segment, you should delete it.

E$OK	0000H	No exceptional conditions.
E$IO	002BH	An I/O error occurred which might have prevented the operation from completing. Examine the unit$status field of the I/O result segment for more information.

AGETFILE$STATUS returns status and attribute information about a file.

```
CALL RQ$A$GET$FILE$STATUS(connection, resp$mbox, except$ptr);
```

Input Parameter

connection — A TOKEN for a connection to the file whose status is sought.

Output Parameters

resp$mbox — A TOKEN for the mailbox that receives a segment containing a data structure with the status information for the specified file. The information in the first part of this structure--down to the dev$conn field--is returned for any file (physical, stream, or named), but information from the file$id field down to the end of the structure is provided only for named files. The contents of the named$file field indicates whether the file is a named file.

```
DECLARE file$info STRUCTURE(
                    status          WORD,
                    num$conn        WORD,
                    num$reader      WORD,
                    num$writer      WORD,
                    share           BYTE,
                    named$file      BYTE,
                    dev$name(14)    BYTE,
                    file$drivers    WORD,
                    functs          BYTE,
                    flags           BYTE,
                    dev$gran        WORD,
                    dev$size        DWORD,
                    dev$conn        WORD,
```

Output Parameters

Information from this point on is returned only if the file is a named file.

```
file$id             WORD,
file$type           BYTE,
file$gran           BYTE,
owner$id            WORD,
create$time         DWORD,
access$time         DWORD,
modify$time         DWORD,
file$size           DWORD,
file$blocks         DWORD,
vol$name(6)         BYTE,
vol$gran            WORD,
vol$size            DWORD,
accessor$count      WORD,
first$access        BYTE,
first$ID            WORD,
second$access       BYTE,
second$ID           WORD,
third$access        BYTE,
third$ID            WORD,
vol$flags           BYTE);
```

These fields are interpreted as follows:

status	A condition code indicating how the get file status operation was completed. If this code is not E$OK, the remaining fields must be considered invalid.
num$conn	The number of connections to the file.
num$reader	The number of connections currently open for reading.
num$writer	The number of connections currently open for writing.
share	The current shared status of the file; possible values are:AGETFILE$STATUS:share values; 0 Private use only 1 Share with readers only 2 Share with writers only 3 Share with all users
named$file	Tells whether this structure contains any information beyond the dev$conn field. 0FFH means yes and 0 means no.

dev$name — The name of the physical device where this file resides (same name as in the DUIB). This name is padded with blanks.

file$drivers — A bit map that tells what kinds of files can reside on this device. If bit n is on, then file driver n+1 can be used. Bit 0 is the low-order bit.

Bit	Driver No.	Driver
0	1	Physical file
1	2	Stream file
2	3	reserved
3	4	Named file

functs — A bit map that describes the functions supported by the device where this file resides. A bit set to one indicates the corresponding function is supported. Bit 0 is the low-order bit.

Bit	Function
0	F$READ
1	F$WRITE
2	F$SEEK
3	F$SPECIAL
4	F$ATTACH$DEV
5	F$DETACH$DEV
6	F$OPEN
7	F$CLOSE

flags — Meaningful only for diskette drives. This field is interpreted as follows. (Bit 0 is the low-order bit.)

Bit	Meaning
0	0 = bits 1-7 are not significant 1 = bits 1-7 are significant
1	0 = single density 1 = double density
2	0 = single sided 1 = double sided
3	0 = 8-inch diskette 1 = 5 1/4-inch diskette
4	0 = standard diskette, meaning that track 0 is single-density with 128-byte sectors

1 = a non-standard diskette or not a diskette

5-7 reserved

dev$gran	The device granularity, in bytes, of the device where this file resides.
dev$size	The storage capacity of the device, in bytes.
dev$conn	The number of connections to the device.

The information from here to the end of the structure is returned only for named files, as indicated by a value of 0FFH in the named$file field.

file$id	A number that distinguishes this file from all other files on the same device. The Disk Verification Utility refers to this number as an FNODE. For information on the disk verify utility, see *Extended iRMX II Disk Verification Utility Reference Manual.*
file$type	Indicates the type of the file: 6 means directory file; and 8 means data file.
file$gran	The file granularity, as a multiple of vol$gran. For example, if file$gran is 2 and vol$gran is 256, then the file's granularity is 512.
owner$id	The first ID in the user object that was presented to the Basic I/O System when the file was created.
create$time	The time and date when the file was created. Whether the Basic I/O System maintains this field is a configuration option.
access$time	The time and date when the file was last accessed. Whether the Basic I/O System maintains this field is a configuration option.
modify$time	The time and date when the file was last modified. Whether the Basic I/O System maintains this field is a configuration option.
file$size	The total size of the file, in bytes.
file$blocks	The number of volume blocks allocated to this file. A volume block is a contiguous area of storage that contains vol$gran bytes of data.

vol$name — The left-adjusted, null-padded ASCII name for the volume containing this file.

vol$gran — The volume granularity, in bytes.

vol$size — The storage capacity, in bytes, of the volume on which this file is stored.

accessor$count — The number of IDs in the file's accessor list. (This may have been added after file creation.)

first$access
second$access
third$access — Access masks for as many ID's as are indicated by accessor$count. The bits of the access masks are defined in the following table. An access right is granted if the appropriate bit is set to 1; otherwise, that right is denied. Bit 0 is the low-order bit.

Bit	Data File	Directory File
0	Delete	Delete
1	Read	Display
2	Append	Add Entry
3	Update	Change Entry
4-7	Reserved	Reserved

first$ID
second$ID
third$ID — ID values for the accessors.

vol$flags — Contains flags for general volume information. The following flags are defined:

Flag	Bit	Meaning
vf$integerity	0	0 = The Volume has been properly shut down. 1 = Indicates possible disk corruption (The volume was attached but was not subsequently shut down).

except$ptr — A POINTER to a WORD where the sequential condition code will be returned.

AGETFILE$STATUS

Description

The AGETFILE$STATUS system call returns status and attribute information about the designated file. Certain information is returned for all file driver types. Additional information is returned for named files.

Note that this call returns device-dependent information.

Condition Codes

AGETFILE$STATUS returns condition codes at two different times. The code returned to the calling task immediately after invocation of the system call is considered a sequential condition code. A code returned as a result of asynchronous processing is a concurrent condition code. A complete explanation of sequential and concurrent parts of system calls is in the *Extended iRMX II Basic I/O System User's Guide*.

The following list is divided into two parts--one for sequential codes, and one for concurrent codes.

Sequential Condition Codes

The Basic I/O System can return the following condition codes to the word specified by the except$ptr parameter of this system call.

E$OK	0000H	No exceptional conditions.
E$EXIST	0006H	At least one of the following is true: • One or more of the following parameters is not a token for an existing object: - The resp$mbox parameter • The connection is being deleted. • The connection for a remote driver is no longer active.
E$LIMIT	0004H	At least one of the following is true: • The calling task's job has already reached its object limit. • The number of outstanding I/O operations for a remote connection has been exceeded.
E$MEM	0002H	The memory available to the calling task's job is not sufficient to complete the call.
ENOTCON-FIGURED	0008H	This system call is not part of the present configuration.

intel

EXTENDED iRMX® II EXTENDED I/O SYSTEM CALLS REFERENCE MANUAL

Intel Corporation
3065 Bowers Avenue
Santa Clara, California 95051

PREFACE

This manual documents the system calls of the Extended I/O System, a subsystem of the Extended iRMX II Operating System. The information provided in this manual is intended as a reference to the system calls and provides detailed descriptions of each call.

READER LEVEL

This manual is intended for programmers who are familiar with the concepts and terminology introduced in the *Extended iRMX II Nucleus User's Guide* and with the PL/M-286 programming language.

CONVENTIONS

System call names appear as headings on the outside upper corner of each page. The first appearance of each system call name is printed in ink; subsequent appearances are in black.

Throughout this manual, system calls are shown using a generic shorthand (such as S$CREATE$FILE instead of RQSCREATE$FILE). This convention is used to allow easier alphabetic arrangement of the calls. The actual PL/M-286 external-procedure names must be used in all calling sequences.

You can also invoke the system calls from assembly language, but you must obey the PL/M-286 calling sequences when doing so. For more information on these calling sequences refer to the *Extended iRMX II Programming Techniques Reference Manual.*

Intel®

CONTENTS

Intel®

iRMX® II EXTENDED I/O SYSTEM CALLS

1.1 INTRODUCTION

This manual describes the system calls provided by the Extended I/O System. The manual contains

- A brief explanation of condition codes.
- A system call dictionary listing the system calls by function.
- Complete descriptions of each system call.

Throughout this manual, PL/M-286 data types, such as BYTE, WORD, and SELECTOR are used. In addition, the extended iRMX II data type TOKEN is used. These words are always capitalized. If your compiler supports the SELECTOR data type, a TOKEN can be declared literally as SELECTOR. Because TOKEN is not a PL/M-286 data type, you must declare it to be literally a SELECTOR every place you use it. Definitions of both PL/M-286 and extended iRMX II data types are given in the *Extended iRMX II Extended I/O System User's Guide*. The word "token" in lowercase refers to a value that the extended iRMX II Operating System returns to a TOKEN (the data type) when it creates the object.

In each description of a system call, you will find a list of possible condition codes. This list is intended to help you debug your application system.

1.2 SYSTEM CALL DICTIONARY

The system call dictionary on the next few pages lists system calls by function rather than alphabetically.

The following abbreviations identify types of files for which a particular system call can be used:

PF means physical file
SF means stream file
NF means named file
ND means named directory

SYSTEM CALLS FOR I/O JOBS

SYSTEM CALLS RELATING TO LOGICAL NAMES

SYSTEM CALLS RELATING TO LOGICAL NAMES (continued)

SYSTEM CALLS FOR CREATING FILES AND CONNECTIONS

SYSTEM CALLS FOR CHANGING ACCESS AND RENAMING

SYSTEM CALLS TO MANIPULATE DATA IN FILES

SYSTEM CALL RELATING DIRECTLY TO DEVICES

SYSTEM CALLS FOR OBTAINING STATUS

SYSTEM CALLS TO DELETE FILES AND CONNECTIONS

SYSTEM CALLS RELATING TO USERS

CREATEIOJOB creates an I/O job containing one task.

```
io$job = RQ$CREATE$IO$JOB(  pool$min, pool$max, except$handler,
                    job$flags, task$priority, start$address,
                    data$seg, stack$ptr, stack$size, task$flags,
                    msg$mbox, except$ptr);
```

Input Parameters

pool$min

A DWORD containing the minimum allowable size of the new job's pool, in 16-byte paragraphs. For example, a value of 35 indicates thirty-five 16-byte paragraphs. The Extended I/O System also uses this value as the initial size of the memory pool for the new job.

You must not assign pool$min a value less than 32. Furthermore, if the base of the stack$ptr parameter is equal to zero, you should ensure that pool$min is no less than 32 + (number of 16-byte paragraphs required to contain the stack). If you set pool$min to a value smaller than these minimums, the Extended I/O System will return an E$PARAM exceptional condition.

The purpose of the pool$min parameter in this system call is identical to the purpose of the pool$min parameter of the CREATE$JOB system call provided by the Nucleus. For information regarding memory pools, refer to the *Extended iRMX II Nucleus User's Guide*.

pool$max

A DWORD containing the maximum allowable size of the new job's pool, in 16-byte paragraphs. For example, a value of 40 indicates forty 16-byte paragraphs.

You must set pool$max to a value no less than pool$min, or the Extended I/O System will return an E$PARAM exceptional condition.

The purpose of the pool$max parameter in this system call is identical to the purpose of the pool$max parameter of the CREATE$JOB system call provided by the extended iRMX II Nucleus. For more information about memory pools, refer to the *Extended iRMX II Nucleus Reference Manual*.

except$handler

A POINTER to a structure of the following form:

```
DECLARE handler          STRUCTURE(
            exception$handler$offset    WORD,
            exception$handler$base      SELECTOR,
            exception$mode              BYTE)
```

The Extended I/O System expects you to designate an exception handler to be used as the new job's default exception handler. If you wish to designate the system default exception handler, you can do so by setting exception$handler$base to SELECTOR$OF(NIL). If you set the base to any other value, then the Extended I/O System assumes that the first two words of this structure point to the first instruction of your exception handler.

Set the exception$mode to tell the Extended I/O System when to pass control to the new task's exception handler. Encode the mode as follows:

Value	When Control Passes To Exception Handler
0	Control never passes to handler
1	On programmer errors only
2	On environmental conditions only
3	On all exceptional conditions

For more information regarding exception handlers and exception modes, refer to the *Extended iRMX II Nucleus Reference Manual.*

job$flags

A WORD that tells the Nucleus whether to check the validity of objects used as parameters in system calls. If bit 1 (where bit 0 is the low-order bit) is zero, the Nucleus will validate objects.

All bits other than bit 1 must be set to zero. This parameter serves precisely the same purpose as the job$flags parameter of the CREATE$JOB system call provided by the Nucleus. Refer to the *Extended iRMX II Nucleus Reference Manual* for more information.

task$priority

A BYTE which establishes the priority of the initial task in the new job.

- If equal to zero, specifies that the new job's initial task is to have a priority equal to the the maximum priority of the initial job of the Extended I/O System. For more information about the initial job of the Extended I/O System, refer to the chapter of the *Extended iRMX II Interactive Configuration Utility Reference Manual* relating to the Extended I/O System.

- If not equal to zero, contains the priority of the initial task of the new job. If this priority is higher than (numerically less than) the maximum priority of the initial job of the Extended I/O System, an E$PARAM error occurs.

start$address — A POINTER to the first instruction of the code segment for the new job's initial task. This code segment can be, but is not required to be, an extended iRMX II segment.

data$seg — A SELECTOR which,

- if SELECTOR$OF(NIL), indicates one of two things. Either the new job's initial task uses no data segment, or it creates one for itself. Tasks can create their own data segments only under special circumstances. To find out more about these circumstances, refer to the Extended I/O System parameters section of the *Extended iRMX II Interactive Configuration Utility Reference Manual*.
- if not SELECTOR$OF(NIL), contains the base address of the data segment of the new job's initial task. This data segment can be, but is not required to be, an extended iRMX II segment.

stack$ptr — A POINTER which,

- if the stack pointer is NIL, specifies that the Nucleus should allocate a stack for the new job's initial task. The length of the allocated stack is determined by the stack$size parameter of this system call. Be aware that this stack is not an extended iRMX II segment.
- if the stack pointer is not equal to NIL, points to the base of the stack for the new job's initial task. Because the Nucleus does not allocate this stack, you must allocate it during the configuration process, or your application code must allocate it while the system is running.

stack$size — A WORD containing the size, in bytes, of the stack for the new job's initial task. If you specify less than 200, the Extended I/O System will increase the size to 200. For information regarding the amount of stack to allocate, refer to the chapter of the *Extended iRMX II Programming Techniques Reference Manual* that discusses stack sizes.

If you are allocating the stack during configuration, or if the application code is allocating the stack while the system is running, the value of this parameter will be the precise amount of stack that the system can use. However, if the Nucleus is allocating the stack for you, it might allocate as many as 15 additional bytes in order to make the stack occupy whole 16-byte paragraphs.

task$flags — A WORD in which all bits except the two low-order bits must be set to zero. The upper 14 bits are reserved for Intel's use.

Bit Zero: Use the low-order bit (bit 0) to tell the operating system whether the new job's initial task uses floating-point instructions. A value of 1 indicates the presence of floating-point instructions, while a zero indicates the absence of floating-point instructions.

Bit One: Bit 1 indicates whether the initial task in the job should run immediately, or whether it should wait until a STARTIOJOB system call is issued to start it. Set bit 1 to zero if the task is to be made ready to run; set bit 1 to one if the task is to wait until the STARTIOJOB call is issued.

msg$mbox — A TOKEN for a mailbox. When a task exits (by invoking EXITIOJOB), the Extended I/O System sends a message to this mailbox. If you desire no such message, assign msg$mbox a value of SELECTOR$OF(NIL).

The format of the message is as follows:

```
DECLARE message STRUCTURE(
            termination$code    WORD,
            user$fault$code     WORD,
            job$token           TOKEN,
            return$data$len     BYTE,
            return$data(*)      BYTE)
```

where

termination$code — A WORD that indicates why an I/O job terminated, as follows:

CODE	MEANING
0	Some task within the job--the terminating task--invoked the EXITIOJOB system call, and indicated with this code that no problem caused the termination. The job has not yet been deleted, and some of its tasks might still be ready.
1	The job was deleted because some task invoked the DELETE$JOB system call.

any other code	Some task within the new job invoked the other EXITIOJOB system call and indicated that the job was terminated because some problem occurred. The job has not yet been deleted and some of its tasks might still be ready.
user$fault$code	A WORD that contains an encoded reason for termination of the new job. Whenever the termination$code has a value other than 0 or 1, this parameter contains an error code that the terminating task specified when invoking the EXITIOJOB system call. The precise meaning of this code is provided by the terminating task, not by the operating system.
job$token	A TOKEN for the job that was terminated.
return$data$len	A BYTE that specifies the length (in bytes) of the return$data parameter described below. The maximum length is 89 (decimal) bytes.
return$data	A sequence of BYTES that contain data specified by the terminating task when it invoked the EXITIOJOB system call.

Output Parameters

io$job	A TOKEN that represents the newly created job. The operating system returns a valid token only if the Extended I/O System returns an E$OK condition code.
except$ptr	A POINTER to a WORD where the Extended I/O System returns the condition code.

Description

This system call creates a job whose tasks can invoke the system calls provided by the Extended I/O System. Such jobs are called I/O jobs, and they differ from other jobs in these ways:

- Job parameter defaults: Many of the parameters required by the Nucleus's CREATE$JOB system call are not required by the CREATEIOJOB system call. These parameters include

  directory$size
  param$object
  max$objects
  max$tasks
  max$priority

  The Extended I/O System allows you to specify values for some of these parameters during the system configuration process. The precise instructions for defining these values are provided in the *Extended iRMX II Interactive Configuration Utility Reference Manual.*

- Default job attributes: The CREATEIOJOB system call provides default values for the following I/O job attributes:

  global job
  default user
  default prefix

  The values for these attributes are passed from parent job to child job. For instance, if Job A uses the CREATEIOJOB system call to spawn Job B, then the Extended I/O System copies the values of the Job A attributes into the Job B attributes. Be aware that if you change the Job A attributes after Job B has been created, the changed values are not copied into Job B.

  You can set the values for these attributes for the "first parent" job during the process of configuring your system.

- Notification of job termination: The CREATEIOJOB system call provides a mechanism for notifying the parent job of the termination of the I/O job. The Extended I/O System implements this mechanism by sending a termination message to a mailbox of your choice whenever a task in the I/O job terminates (calls EXITIOJOB). You specify the mailbox by using the msg$mbox parameter of this system call.

The CREATEIOJOB system call can be called only from another I/O job. You can set up one or more initial I/O jobs while configuring the operating system. For more information about configuration, refer to Chapter 7 of the *Extended iRMX II Extended I/O System User's Guide.*

Do not delete a task in an I/O job if the task is using a connection (that is, if the connection has not been deleted). If you do so, the connection will not be available to any other task.

Condition Codes

E$OK	0000H	No exceptional conditions.
E$CONTEXT	0005H	The calling task's job is not an I/O job.
E$EXIST	0006H	At least one of the following is true: • The token cataloged under the name RQGLOBAL (the global job) is not a token for an existing object. (See the *Extended iRMX II Basic I/O System User's Guide* for information on the global object directory.) • The value assigned to the msg$mbox parameter is not a token for an existing mailbox. • The user TOKEN is not valid.
E$MEM	0002H	The memory available to the calling task's job is not sufficient to complete the call.
ENOTCONFIGURED	0008H	This system call is not part of the present configuration.
E$NOUSER	8021H	The calling task's job does not have a default user, or the object cataloged under the logical name R?IOUSER is not a user object. (See the *Extended iRMX II Basic I/O System User's Guide* for information on R?IOUSER.)
E$PARAM	8004H	At least one of the following is true: • The value assigned to the pool$min parameter is less than 32 decimal, or it is greater than the value assigned to the pool$max parameter. • The value assigned to task$priority is not zero and is greater than (numerically less than) the maximum priority of the calling I/O job. • The value assigned to the exception$mode parameter is outside the range 0-3, inclusive. • Either the name or password contains invalid characters.
EIOJOB	0047H	The calling task's job is not an I/O job.

RQE$CREATE$IO$JOB creates an I/O job containing one task with a maximum of 16M bytes of memory pool.

```
io$job = RQE$CREATE$IO$JOB(pool$min, pool$max, except$handler,
                           job$flags, task$priority, start$address,
                           data$seg, stack$ptr, stack$size, task$flags,
                           msg$mbox, except$ptr);
```

Input Parameters

pool$min

A DWORD containing the minimum allowable size of the new job's pool, in 16-byte paragraphs. For example, a value of 35 indicates thirty-five 16-byte paragraphs. The Extended I/O System also uses this value as the initial size of the memory pool for the new job. The memory initially allocated is always contiguous. If additional memory is requested, it is not necessarily contiguous.

You must not assign pool$min a value less than 32. Furthermore, if the base of the stack$ptr parameter is equal to zero, you should ensure that pool$min is no less than 32 + (number of 16-byte paragraphs required to contain the stack). If you set pool$min to a value smaller than these minimums, the Extended I/O System will return an E$PARAM exceptional condition.

The purpose of the pool$min parameter in this system call is identical to the purpose of the pool$min parameter of the CREATE$JOB system call provided by the Nucleus. For information regarding memory pools, refer to the *Extended iRMX II Nucleus Reference Manual.*

pool$max

A DWORD containing the maximum allowable size of the new job's pool, in 16-byte paragraphs. For example, a value of 40 indicates forty 16-byte paragraphs.

You must set pool$max to a value no less than pool$min, or the Extended I/O System will return an E$PARAM exceptional condition.

The purpose of the pool$max parameter in this system call is identical to the purpose of the pool$max parameter of the CREATE$JOB system call provided by the extended iRMX II Nucleus. For more information about memory pools, refer to the *Extended iRMX II Nucleus Reference Manual.*

except$handler

A POINTER to a structure of the following form:

```
DECLARE handler     STRUCTURE(
        exception$handler$offset    WORD,
        exception$handler$base      SELECTOR,
        exception$mode              BYTE)
```

The Extended I/O System expects you to designate an exception handler to be used as the new job's default exception handler. If you wish to designate the system default exception handler, you can do so by setting exception$handler$base to SELECTOR$OF(NIL). If you set the base to any other value, then the Extended I/O System assumes that the first two words of this structure point to the first instruction of your exception handler.

Set the exception$mode to tell the Extended I/O System when to pass control to the new task's exception handler. Encode the mode as follows:

Value	When Control Passes To Exception Handler
0	Control never passes to handler
1	On programmer errors only
2	On environmental conditions only
3	On all exceptional conditions

For more information regarding exception handlers and exception modes, refer to the *Extended iRMX II Nucleus Reference Manual.*

job$flags

A WORD that tells the Nucleus whether to check the validity of objects used as parameters in system calls. If bit 1 (where bit 0 is the low-order bit) is zero, the Nucleus will validate objects.

All bits other than bit 1 must be set to zero. This parameter serves precisely the same purpose as the job$flags parameter of the CREATE$JOB system call provided by the Nucleus. Refer to the *Extended iRMX II Nucleus Reference Manual* for more information.

task$priority

A BYTE which establishes the priority of the initial task in the new job.

- If equal to zero, specifies that the new job's initial task is to have a priority equal to the the maximum priority of the initial job of the Extended I/O System. For more information about the initial job of the Extended I/O System, refer to the chapter of the *Extended iRMX II Interactive Configuration Utility Reference Manual* relating to the Extended I/O System.

- If not equal to zero, contains the priority of the initial task of the new job. If this priority is higher than (numerically less than) the maximum priority of the initial job of the Extended I/O System, an E$PARAM error occurs.

start$address — A POINTER to the first instruction of the code segment for the new job's initial task. This code segment can be, but is not required to be, an extended iRMX II segment.

data$seg — A SELECTOR which,

- if SELECTOR$OF(NIL), indicates one of two things. Either the new job's initial task uses no data segment, or it creates one for itself. Tasks can create their own data segments only under special circumstances. To find out more about these circumstances, refer to the Extended I/O System parameters section of the *Extended iRMX II Interactive Configuration Utility Reference Manual.*
- if not SELECTOR$OF(NIL), contains the base address of the data segment of the new job's initial task. This data segment can be, but is not required to be, an extended iRMX II segment.

stack$ptr — A POINTER which,

- if the stack pointer is NIL, specifies that the Nucleus should allocate a stack for the new job's initial task. The length of the allocated stack is determined by the stack$size parameter of this system call. Be aware that this stack is not an extended iRMX II segment.
- if the stack pointer is not equal to NIL, points to the base of the stack for the new job's initial task. Because the Nucleus does not allocate this stack, you must allocate it during the configuration process, or your application code must allocate it while the system is running.

stack$size — A WORD containing the size, in bytes, of the stack for the new job's initial task. If you specify less than 200, the Extended I/O System will increase the size to 200. For information regarding the amount of stack to allocate, refer to the chapter of the *Extended iRMX II Programming Techniques* manual that discusses stack sizes.

If you are allocating the stack during configuration, or if the application code is allocating the stack while the system is running, the value of this parameter will be the precise amount of stack that the system can use. However, if the Nucleus is allocating the stack for you, it might allocate as many as 15 additional bytes in order to make the stack occupy whole 16-byte paragraphs.

task$flags

A WORD in which all bits except the two low-order bits are set to zero.

Bit Zero: Use the low-order bit (bit 0) to tell the operating system whether the new job's initial task uses floating-point instructions. A value of 1 indicates the presence of floating-point instructions, while a zero indicates the absence of floating-point instructions.

Bit One: Bit 1 indicates whether the initial task in the job should run immediately, or whether it should wait until a STARTIOJOB system call is issued to start it. Set bit 1 to zero if the task is to be made ready to run; set bit 1 to one if the task is to wait until the STARTIOJOB call is issued.

msg$mbox

A TOKEN for a mailbox. When a task exits (by invoking EXITIOJOB), the Extended I/O System sends a message to this mailbox. If you desire no such message, assign msg$mbox a value of zero.

The format of the message is as follows:

```
DECLARE message     STRUCTURE(
                  termination$code    WORD,
                  user$fault$code     WORD,
                  job$token           WORD,
                  return$data$len     BYTE,
                  return$data(*)      BYTE)
```

where

termination$code — A WORD that indicates why an I/O job terminated, as follows:

CODE	MEANING
0	Some task within the job--the terminating task--invoked the EXITIOJOB system call, and indicated with this code that no problem caused the termination. The job has not yet been deleted, and some of its tasks might still be ready.
1	The job was deleted because some task invoked the DELETE$JOB system call.

any other code	Some task within the new job invoked the EXITIOJOB system call and indicated that the job was terminated because some problem occurred. The job has not yet been deleted and some of its tasks might still be ready.
user$fault$code	A WORD that contains an encoded reason for termination of the new job. Whenever the termination$code has a value other than 0 or 1, this parameter contains an error code that the terminating task specified when invoking the EXITIOJOB system call. The precise meaning of this code is provided by the terminating task, not by the operating system.
job$token	A TOKEN for the job that was terminated.
return$data$len	A BYTE that specifies the length (in bytes) of the return$data parameter described below. The maximum length is 89 (decimal) bytes.
return$data	A sequence of BYTES that contain data specified by the terminating task when it invoked the EXITIOJOB system call.

Output Parameters

io$job	The TOKEN that represents the newly created job. The operating system returns a valid token only if the Extended I/O System returns an E$OK condition code.
except$ptr	A POINTER to a WORD where the Extended I/O System returns the condition code.

Description

This system call creates a job whose tasks can invoke the system calls provided by the Extended I/O System. Such jobs are called I/O jobs, and they differ from other jobs in these ways:

- Job parameter defaults: Many of the parameters required by the Nucleus's CREATE$JOB system call are not required by the CREATEIOJOB system call. These parameters include

  directory$size
  param$object
  max$objects
  max$tasks
  max$priority

The Extended I/O System allows you to specify values for some of these parameters during the system configuration process. The precise instructions for defining these values are provided in the *Extended iRMX II Interactive Configuration Utility Reference Manual*.

- Default job attributes: The CREATEIOJOB system call provides default values for the following I/O job attributes:

  global job
  default user
  default prefix

  The values for these attributes are passed from parent job to child job. For instance, if Job A uses the E$CREATE$IO$JOB system call to spawn Job B, then the Extended I/O System copies the values of the Job A attributes into the Job B attributes. Be aware that if you change the Job A attributes after Job B has been created, the changed values are not copied into Job B.

  You can set the values for these attributes for the "first parent" job during the process of configuring your system.

- Notification of job termination: The CREATEIOJOB system call provides a mechanism for notifying the parent job of the termination of the I/O job. The Extended I/O System implements this mechanism by sending a termination message to a mailbox of your choice whenever a task in the I/O job terminates (calls EXITIOJOB). You specify the mailbox by using the msg$mbox parameter of this system call.

The E$CREATE$IO$JOB system call can be called only from another I/O job. You can set up one or more initial I/O jobs while configuring the operating system. For more information about configuration, refer to Chapter 7 of the *Extended iRMX II Extended I/O System User's Guide*.

Do not delete a task in an I/O job if the task is using a connection (that is, if the connection has not been deleted). If you do so, the connection will not be available to any other task.

Condition Codes

E$OK	0000H	No exceptional conditions.
E$CONTEXT	0005H	The calling task's job is not an I/O job.

E$EXIST	0006H	At least one of the following is true: • The token cataloged under the name RQGLOBAL (the global job) is not a token for an existing object. (See the EXTENDED iRMX II BASIC I/O SYSTEM USER'S GUIDE for information on the global object directory.) • The value assigned to the msg$mbox parameter is not a token for an existing mailbox.
E$MEM	0002H	The memory available to the calling task's job is not sufficient to complete the call.
ENOTCONFIGURED	0008H	This system call is not part of the present configuration.
E$NOUSER	8021H	The calling task's job does not have a default user, or the object cataloged under the logical name R?IOUSER is not a user object. (See the EXTENDED iRMX II BASIC I/O SYSTEM USER'S GUIDE for information on R?IOUSER.)
E$PARAM	8004H	At least one of the following is true: • The value assigned to the pool$min parameter is less than 32 decimal, or it is greater than the value assigned to the pool$max parameter. • The value assigned to task$priority is not zero and is greater than (numerically less than) the maximum priority of the calling I/O job. • The value assigned to the exception$mode parameter is outside the range 0-3, inclusive.
EIOJOB	0047H	The calling task's job is not an I/O job.

EXITIOJOB sends a message to a previously designated mailbox and deletes the calling task.

```
CALL RQ$EXIT$IO$JOB(user$fault$code, return$data$ptr, except$ptr);
```

Input Parameters

user$fault$code	A WORD containing the encoded reason for terminating the job. If you terminate the job under normal circumstances, you should enter a value of zero. If you terminate the job because of a problem, you should enter an error code that identifies the problem. The Extended I/O System sends a structure containing the value you enter to the mailbox specified in the E$CREATE$IO$JOB system call.
return$data$ptr	A POINTER to a buffer containing a STRING containing data (provided by the calling task) to be returned to the message mailbox specified in the CREATEIOJOB system call. If you enter NIL, no data is returned. If the string is longer than 89 (decimal) bytes, only the first 89 bytes are returned.

Output Parameter

except$ptr	A POINTER to a WORD where the Extended I/O System returns the condition code.

Description

The EXITIOJOB system call complements the CREATEIOJOB system call. Using the EXITIOJOB system call, a task can delete itself and have the Extended I/O System notify the parent job of the deletion.

When a task in an I/O job (a job created by the CREATEIOJOB system call) invokes the EXITIOJOB system call, two things happen:

- The Extended I/O System deletes the task (but not the job containing the task) that invoked the EXITIOJOB system call.
- The Extended I/O System sends a termination message to the mailbox specified in the CREATEIOJOB system call.

Special Circumstances

Your application code can use this system call to bring about an orderly deletion of an I/O job. To do this, have a task within the I/O job invoke this system call. Then have a task in the parent job receive the message and delete the I/O job. Under certain circumstances, this system call does not delete the calling task or does not send a termination message.

Calling Task Not Deleted

Although the EXITIOJOB system call generally deletes the calling task, this deletion does not occur in the following circumstances:

- If the DELETE$TASK system call (which the Extended I/O System calls) returns an exception code to the Extended I/O System.
- If the calling task is an interrupt task.

In both cases, the Extended I/O System returns control to the calling task and issues an exceptional condition code to indicate the nature of the problem. Under any other circumstance, the Extended I/O System deletes the calling task.

Even if it fails to delete the task, the Extended I/O System sends the termination message if one has been requested, except for the following circumstances:

- If the msg$mbox parameter of the CREATEIOJOB was set to SELECTOR$OF(NIL).
- If the mailbox specified in the msg$mbox parameter of the CREATEIOJOB system call no longer exists.

Condition Codes

E$CONTEXT	0005H	The task invoking the EXITIOJOB system call is an interrupt task and cannot be deleted.
ENOTCON-FIGURED	0008H	This system call is not part of the present configuration.

The GET$LOGICAL$DEVICE$STATUS system call provides status information about a logical device.

```
CALL RQ$GET$LOGICAL$DEVICE$STATUS( log$name$ptr, dev$info$ptr,
                                   except$ptr);
```

Input Parameter

log$name$ptr — A POINTER to a STRING containing the logical name under which the logical device object is cataloged in the root object directory.

Output Parameters

dev$info$ptr — A POINTER to a structure in which the Extended I/O System returns the status information. You can allocate memory for this structure by requesting an extended iRMX II segment or by reserving the memory in your code. The structure must have the following form:

```
DECLARE dev$info    STRUCTURE(
                        device$name(15)     BYTE,
                        file$driver         BYTE,
                        num$conns           WORD,
                        owner$id            WORD)
```

where

device$name — A STRING containing the physical name associated with the device. This is the name established during Basic I/O System configuration.

file$driver — The file driver associated with the device. Possible values include

value	file driver
1	physical
2	stream
4	named
5	remote

	num$conns	The current number of connections to the device.
	owner$id	The owner ID for this device. This ID is the first ID listed in the default user object of the attaching task's job.
except$ptr	A POINTER to a WORD in which the Extended I/O System returns the condition code.	

Description

The GET$LOGICAL$DEVICE$STATUS system call allows a task to obtain status information about logical names that represent devices. The Extended I/O System does not check access before returning status information.

Condition Codes

E$OK	0000H	No exceptional conditions.
E$EXIST	0006H	The device connection corresponding to the logical name is being deleted.
E$LIMIT	0004H	Either the user object or the calling task's job is already involved in 255 (decimal) I/O operations.
ELOGNAME$-NEXIST	0045H	The logical name was not found in the root object directory.
ELOGNAME$-SYNTAX	0040H	The syntax of the specified logical name is incorrect because at least one of the following conditions is true: • The name was missing matching colons (:). • The STRING pointed to by the log$name$ptr parameter is of zero length or has a length greater than 12 (not including colons (:). • The logical name contains invalid characters.
ENOTCON-FIGURED	0008H	This system call is not part of the present configuration.
ENOTDEVICE	8041H	The specified logical name does not represent a valid device connection.

The GET$USER$IDS system call returns the user ID(s) associated with a USER defined in the User Definition File (UDF).

```
CALL RQ$GET$USER$IDS(name$ptr, ids$ptr, except$ptr);
```

Input Parameter

name$ptr — A POINTER to a STRING containing the user name. (Only the first eight characters are significant.)

Output Parameters

ids$ptr — A POINTER to a structure where the ID(s) associated with the user name will be placed. The structure has the following form:

```
DECLARE ids     STRUCTURE (
                    length      WORD,
                    count       WORD,
                    id(*)       WORD );
```

where

length — Should be set by the caller to the maximum number of ID(s) desired.

count — Will contain the number of valid IDs in the ID array after GET$USER$IDS has returned to the caller. This value will never be greater than the ids.length. The user does not need to initialize this value.

id — Is an array of IDs obtained from the UDF. The length of this array is contained in ids.count. The user does not need to initialize this value.

except$ptr — A POINTER to a WORD where the Extended I/O System returns a condition code.

GET$USER$IDS

Description

This system call returns the user ID(s) associated with a user name defined in the User Definition File (UDF). It searches the file :CONFIG:UDF for the user name pointed to by the name$ptr parameter and if found, returns that user's ID(s). Refer to the section on configuration in the *Extended iRMX II Extended I/O System User's Guide* for details.

Condition Codes

E$OK	0000H	No exceptional conditions.
EBADCALL	8005H	A task wrote over the interface library or over the EIOS job.
E$CONTEXT	0005H	The calling job is not an I/O job.
EDEVDETACHING	0039H	An I/O operation could not be performed on the device (:SD:) because it was being detached.
E$DEVFD	0022H	The device (:SD:) cannot be used with the file driver as specified in the preceding logical attach operation.
EUDFFORMAT	0048H	The UDF is not in the correct format.
E$FACCESS	0026H	The user does not have the proper access rights for the requested operation.
E$FLUSHING	002CH	The device (:SD:) is being detached.
E$FNEXIST	0021H	At least one of the following is true: • The file or a file in its path does not exist. • The specified physical device was not found.
E$FTYPE	0027H	A path component is not a directory file.
E$ILLVOL	002DH	The file driver given in the volume label conflicts with the file driver specified in the preceding logical attach operation.
E$INVALID$FNODE	003DH	The fnode associated with a file is either marked not allocated, or the fnode number is out of range. This file should be deleted.
EIOHARD	0052H	A hard error occurred; the BIOS cannot retry the request.
EIOMEM	0042H	The BIOS job did not have enough memory to perform the requested function.
EIOOPRINT	0053H	The device is off-line; operator intervention is required.

EIOSOFT	0051H	A soft error occurred and the BIOS has retried the operation and has failed; a retry is not possible.
EIOUNCLASS	0050H	An unclassified I/O error occurred.
EIOWR$PROT	0054H	The volume specified in this call is write protected.
E$LIMIT	0004H	The root job object directory is full.
ELOGNAME$-NEXIST	0045H	The logical name was not found in the caller's object directory, the global job object directory, or the root job object directory.
E$MEDIA	0044H	The device associated with the system call is off-line.
E$NAME$NEXIST	0049H	The name specified in this call is not defined.
E$NOPREFIX	8022H	The caller's job does not have a default prefix, or it is invalid.
E$NOUSER	8021H	The caller's job does not have a default user or it is invalid.
ENOTCON-FIGURED	0008H	This system call is not part of the present configuration.
E$PARAM	8004H	At least one of the following is true: • The name$ptr parameter is equal to NIL. • The length field of the ids structure is equal to zero. • The name contains invalid characters.
E$SHARE	0028H	The file is not sharable with the requested access.

The HYBRID$DETACH$DEVICE system call removes the correspondence between a logical name and a physical device without removing the logical name from the root object directory.

```
CALL RQ$HYBRID$DETACH$DEVICE(log$name$ptr, except$ptr);
```

Input Parameter

log$name$ptr	A POINTER to a STRING containing the logical name under which the logical device object is cataloged in the root object directory.

Output Parameter

except$ptr	A POINTER to a WORD where the Extended I/O System returns the condition code.

Description

HYBRID$DETACH$DEVICE severs an association created by a call to LOGICAL$ATTACH$DEVICE without deleting the corresponding entry in the root object directory. When a task calls HYBRID$DETACH$DEVICE, the Extended I/O System detaches the device by issuing the Basic I/O System A$PHYSICAL$DETACH$DEVICE call. In so doing, the Extended I/O System specifies the hard detach option which deletes all connections to files on the device.

A device detached using HYBRID$DETACH$DEVICE can be reattached in one of two ways:

- A task can issue the Basic I/O System A$PHYSICAL$ATTACH$DEVICE system call.
- A task can use the device's logical name (which is still cataloged in the root object directory) as the prefix portion of a pathname when issuing an Extended I/O System call. In this case, the Extended I/O System physically attaches the device using the parameters originally specified when the logical name was established (via LOGICAL$ATTACH$DEVICE).

A task cannot use LOGICAL$ATTACH$DEVICE to reattach a device that HYBRID$DETACH$DEVICE detached. Before reattaching a device with LOGICAL$ATTACH$DEVICE, a task must first issue LOGICAL$DETACH$DEVICE.

The HYBRID$DETACH$DEVICE system call is particularly useful for tasks that must temporarily detach a device and attach it in a different way (for example, attaching a disk as a physical device when formatting a volume). These tasks can call HYBRID$DETACH$DEVICE to detach the device, attach the device using A$PHYSICAL$ATTACH$DEVICE, perform the special processing on the device, and detach the device using A$PHYSICAL$DETACH$DEVICE. Later, when a task includes the device's logical name in an Extended I/O System call, the Extended I/O System automatically reattaches the device in the previous manner.

The HYBRID$DETACH$DEVICE system call can be issued as follows:

- By the task ("attaching task") that created the logical name by issuing LOGICAL$ATTACH$DEVICE, or by some other task in the same job as the attaching task.
- By any task in a job whose default user object contains the file's owner ID in its ID list.
- By the System Manager.

Condition Codes

E$OK	0000H	No exceptional conditions.
E$EXIST	0006H	The device connection corresponding to the logical name is being deleted.
E$LIMIT	0004H	Either the user object or the calling task's job is already involved in 255 (decimal) I/O operations.
ELOGNAME$-NEXIST	0045H	The logical name was not found in the root object directory.
ELOGNAME$-SYNTAX	0040H	The syntax of the specified logical name is incorrect because at least one of the following conditions is true: • The STRING pointed to by the log$name$ptr parameter is of zero length, has a length greater than 12 not including colons (:), or is missing matching colons. • The logical name contains invalid characters.
ENOTCON-FIGURED	0008H	This system call is not part of the present configuration.
ENOTDEVICE	8041H	The specified logical name does not represent a valid device connection.
ENOTOWNER	0046H	The user (specified by the default user object) is not the user that attached the device.

The LOGICAL$ATTACH$DEVICE system call assigns a logical name to a physical device.

CAUTION

Any task that uses this system call loses its device independence. To maintain as much device independence as possible in your application, a few selected tasks should perform all attaching and detaching of devices.

```
CALL RQ$LOGICAL$ATTACH$DEVICE(log$name$ptr, dev$name, file$driver,
                              except$ptr);
```

Input Parameters

log$name$ptr — A POINTER to a STRING (of 1 to 12 characters) containing the logical name to be assigned to a device. The name can be delimited with colons (:). The operating system removes the colons so that a logical name with colons is the same as one without (e.g., :F0: is effectively the same as F0), and colons do not count in the length of the name. If you intend to use this logical name as part of a pathname in other system calls, enclose it in colons.

dev$name — A POINTER to a STRING containing the name of the device to which the logical name is assigned. This device name is the name of a Device-Unit Information Block (DUIB) specified during Basic I/O System configuration.

file$driver — A BYTE specifying which Basic I/O System file driver to use with the device. Possible values are as follows:

value	file driver
1	physical
2	stream
4	named
5	remote

Output Parameter

except$ptr — A POINTER to a WORD where the Extended I/O System returns the condition code.

Description

LOGICAL$ATTACH$DEVICE assigns a logical name to a physical device. This system call creates a Logical Device Object that corresponds to a physical device. This Logical Device Object is cataloged in the root object directory under the logical name pointed to by log$name$ptr. The Logical Device Object must be cataloged before the Extended I/O System can make connections to files on the device.

The first Extended I/O System call that uses the logical name as a prefix in a path name causes the physical device to be attached. (The Extended I/O System uses the Basic I/O System call A$PHYSICAL$ATTACH$DEVICE.) The logical name can be used as a prefix in other system calls and can be deleted by LOGICAL$DETACH$DEVICE.

Because of the nature of LOGICAL$ATTACH$DEVICE, some exception codes that result because of errors in this system call are not returned until the Extended I/O System tries to attach the device with A$PHYSICAL$ATTACH$DEVICE.

Condition Codes

E$OK	0000H	No exceptional conditions.
E$CONTEXT	0005H	The root object directory already contains an entry with the name pointed to by the log$name$ptr parameter.
E$LIMIT	0004H	At least one of the following is true: • The calling task's job object directory is full. • The root object directory is full. • The calling task's job is not an I/O job.
E$MEM	0002H	The memory available to the calling task's job is not sufficient to complete this call.
ELOGNAME$-SYNTAX	0040H	The specified logical name is syntactically incorrect because at least one of the following conditions is true: • The STRING pointed to by the log$name$ptr parameter is of zero length or has a length of greater than 12 (including the colons). • The logical name contains invalid characters.
ENOTCON-FIGURED	0008H	This system call is not part of the present configuration.

The LOGICAL$DETACH$DEVICE system call removes the correspondence between a logical name and a physical device, and removes the logical name from the root object directory.

```
CALL RQ$LOGICAL$DETACH$DEVICE(log$name$ptr, except$ptr);
```

Input Parameter

log$name$ptr	A POINTER to a STRING containing the logical name under which the logical device object is catalogued in the root object directory.

Output Parameter

except$ptr	A POINTER to a WORD where the Extended I/O System returns the condition code.

Description

LOGICAL$DETACH$DEVICE severs an association created by a call to LOGICAL$ATTACH$DEVICE and deletes the corresponding entry in the root object directory. After LOGICAL$DETACH$DEVICE is issued, users cannot create new connections using the logical name as a prefix. When the last file connection on the physical device is severed, the Extended I/O System detaches the device (issues the Basic I/O System call A$PHYSICAL$DETACH$DEVICE).

The LOGICAL$DETACH$DEVICE system call can be issued as follows:

- By the task ("attaching task") that created the logical name by issuing LOGICAL$ATTACH$DEVICE, or by some other task in the same job as the attaching task.
- By another job having the same owner ID in its default user object.
- By the System Manager.

Condition Codes

E$OK	0000H	No exceptional conditions.
E$EXIST	0006H	The device connection corresponding to this logical name is being deleted.

E$LIMIT	0004H	One of the following is true: • The job has reached the object limit of the calling task's object directory. • Either the user object or the calling task's job is already involved in 255 (decimal) I/O operations. • The calling task's job is not an I/O job.
ELOGNAME$-NEXIST	0045H	The logical name was not found in the root object directory.
ELOGNAME$-SYNTAX	0040H	The syntax of the specified logical name is incorrect because at least one of the following conditions is true: • The STRING pointed to by the log$name$ptr parameter is of zero length or has a length greater than 12 (not including colons (:)). • The logical name contains invalid characters.
E$MEM	0002H	The memory available to the calling task's job is not sufficient to complete the call.
ENOTCON-FIGURED	0008H	This system call is not part of the present configuration.
ENOTDEVICE	8041H	The specified logical name does not represent a valid device connection.
ENOTOWNER	0046H	The user (specified by the default user object) is not the user that attached the device.

STARTIOJOB starts the execution of a task in an I/O job. The task was not started when the I/O job was created.

```
CALL RQ$START$IO$JOB(io$job, except$ptr);
```

Input Parameter

io$job	TOKEN for the I/O job to be started. This is the TOKEN that was returned by the call to CREATEIOJOB.

Output Parameter

except$ptr	A POINTER to a WORD where the Extended I/O System returns the condition code.

Description

When you call RQE$CREATE$IO$JOB you can specify (with the task$flags parameter) that the task in the new job not start running until the STARTIOJOB call is issued. In this way you can initialize any items that need to be set before the initial task in the new job starts running. For example, you can create a job, catalog a logical name in the new job's object directory, and then issue STARTIOJOB.

Condition Codes

E$OK	0000H	No exceptional conditions.
ENOTCON-FIGURED	0008H	This system call is not part of the present configuration.
E$TIME	0001H	The job cannot be started yet, probably because the operating system has not finished processing the CREATEIOJOB call that created this job.

The S$ATTACH$FILE system call creates a connection to an existing file.

```
connection = RQ$S$ATTACH$FILE(path$ptr, except$ptr);
```

Input Parameter

path$ptr	A POINTER to a STRING containing the pathname of the file to be attached.

Output Parameters

connection	The TOKEN that represents the new connection to the file.
except$ptr	A POINTER to a WORD where the Extended I/O System returns the condition code.

Description

This system call allows a task to obtain a connection to any named, physical, or stream file.

The Extended I/O System allows any task to attach any file. However, if the file being attached is a named file, the Extended I/O System computes access rights for the connection. These access rights are based on the file's access list and the user IDs in the default user object of the calling task's job. (Refer to extended iRMX II Operating System user guides for more information.) If the file's access list allows no access to the users listed in the default user object, the call creates the connection, but it allows no access.

Condition Codes

E$OK	0000H	No exceptional conditions.
E$ALREADY$-ATTACHED	0038H	The Extended I/O System cannot attach the device containing the file because the Basic I/O system has already attached the device.
E$CONTEXT	0005H	The calling task's job is not an I/O job.
EDEVDETACHING	0039H	The device containing the specified file is in the process of being detached.

E$DEVFD	0022H	The Extended I/O System attempted the physical attachment of a device that had formerly been only logically attached. In the process, it found that the device and the device driver specified in the logical attachment were incompatible.
E$EXIST	0006H	The device connection TOKEN is invalid.
E$FACCESS	0026H	The default user object is not allowed access to the file. See the Description section for more information.
E$FNEXIST	0021H	A file in the specified path, or the target file itself, does not exist or is marked for deletion.
E$FTYPE	0027H	The specified path is attempting to use a data file as a directory.
E$ILLVOL	002DH	The Extended I/O System attempted the physical attachment of a device that had formerly been only logically attached. During this process, it examined the volume label and found that the volume does not contain named files. This prevented the Extended I/O System from completing physical attachment because the named file driver was requested during logical attachment.
E$INVALID$FNODE	003DH	The fnode for the specified file is invalid. The file cannot be accessed; you should delete it.
EIOHARD	0052H	A hard I/O error occurred. A retry is probably useless.
EIOMEM	0042H	The BIOS job did not have enough memory to perform the requested function.
EIOOPRINT	0053H	The device was off-line. Operator intervention is required.
EIOSOFT	0051H	A soft I/O error occurred. The Basic I/O System tried to perform the operation a number of times and failed (the number of retries is a configuration parameter). Another retry might still be successful.
EIOUNCLASS	0050H	An unknown type of I/O error occurred.
E$LIMIT	0004H	At least one of the following is true: • The calling task reached the object limit. • The user object or the calling task's job is already involved in 255 (decimal) I/O operations.

		• The calling task's job is not an I/O job.
ELOGNAME$-NEXIST	0045H	The specified path contains an explicit logical name, but the call was unable to find this name in the object directories of the calling task's local job, the global job, or the root job.
ELOGNAME$-SYNTAX	0040H	The specified logical name contains at least one of the following syntax errors: • The specified path starts with a colon (:), indicating that it contains a logical name. But the call was unable to find a second colon to delimit the logical name. • The specified path contains a logical name that is either longer than 12 characters (including colons), has no characters, or contains invalid characters.
E$MEDIA	0044H	The device containing the specified file is not on-line.
E$MEM	0002H	The memory available to the calling task's job is not sufficient to complete the call.
ENOPREFIX	8022H	You did not specify an explicit prefix (logical name), and the default prefix for the calling task's job is either undefined, or it is not a valid device connection or file connection.
ENOTCON-FIGURED	0008H	This system call is not part of the present configuration.
ENOTLOG$NAME	8040H	The specified path contains a logical name that represents an object that is neither a device connection nor a file connection.
ENOUSER	8021H	The calling task's job does not have a default user, or its default user is not a user object.
E$PARAM	8004H	The Extended I/O System attempted the physical attachment of a device that had formerly been only logically attached. The logical attachment referred to a file driver (named, physical, or stream) that is not configured into your system, so physical attachment is not possible.
E$PATHNAME$-SYNTAX	003EH	The specified pathname contains invalid characters.

The S$CATALOG$CONNECTION system call creates a logical name for a connection by cataloging the connection in the object directory of a specific job.

```
CALL RQ$S$CATALOG$CONNECTION(job, connection, log$name$ptr,
                             except$ptr);
```

Input Parameters

job	A TOKEN for the job in whose object directory the logical name is to be cataloged. If the value of this parameter is SELECTOR$OF(NIL), the Extended I/O System catalogs the connection in the object directory of the calling task's job.
connection	A TOKEN for the connection to be assigned the logical name. If the value of this parameter is SELECTOR$OF(NIL), the Extended I/O System obtains the connection by looking up the name in the object directory of the calling task's job.
log$name$ptr	A POINTER to a buffer containing the logical name, which must be a STRING of 12 or fewer characters. The name can be delimited with colons (:). The operating system removes the colons so that a logical name with colons is the same as one without (e.g., :F0: is effectively the same as F0); colons do not count in the length of the name. If you expect to use this logical name in other Extended I/O System calls, delimit the name with colons.

Output Parameter

except$ptr	A POINTER to a WORD where the Extended I/O System returns the condition code.

Description

The Extended I/O System converts the characters in the log$name$ptr STRING to uppercase and catalogs the connection in the object directory of the specified job. However, two special situations affect the outcome of this system call:

- If the job's object directory already contains the logical name, the new connection replaces the existing object in the directory. The Extended I/O System considers this to be a normal circumstance and, consequently, does not return an exception code.

- If your task sets the connection parameter to SELECTOR$OF(NIL), the Extended I/O System looks up the logical name in the object directory of the calling task's job. The system then copies the logical name and its definition into the object directory of the specified job.

Condition Codes

E$OK	0000H	No exceptional conditions.
E$CONTEXT	0005H	The job in which your task is attempting to catalog the connection has an object directory that is zero bytes long.
E$EXIST	0006H	The job or connection parameter is not a token for an existing object.
E$LIMIT	0004H	At least one of the following is true: • The object directory for the specified job is already full. • The calling task's job is not an I/O job.
ELOGNAME$NEXIST	0045H	The Extended I/O System was unable to find the specified logical name in the object directory of the calling task's job.
ELOGNAME$-SYNTAX	0040H	The specified logical name contains at least one of the following syntax errors: • The specified path starts with a colon (:), indicating that it contains a logical name. But the call was unable to find a second colon to delimit the logical name. • The specified path contains a logical name that is either longer than 12 characters, has no characters, or contains invalid characters.
E$MEM	0002H	The memory available to the calling task's job is not sufficient to complete the call.
ENOTCON-FIGURED	0008H	This system call is not part of the present configuration.
ENOTCON-NECTION	8042H	The connection parameter is a token for an object that is not a connection.
E$TYPE	8002H	The job parameter is a token for an object that is not a job.

The S$CHANGE$ACCESS system call changes the access list for a named file. This system call can be used for either data or directory files.

```
CALL RQ$S$CHANGE$ACCESS(path$ptr, id, access, except$ptr);
```

Input Parameters

path$ptr — A POINTER to a STRING containing a path to the file whose access is to be changed.

id — A WORD containing the ID of the user whose access to the file is to be changed. This value can differ from the owner ID of the calling task's default user object. If the file's access list contains the ID, the Extended I/O System changes the ID's current access. If the access list does not contain the ID, the Extended I/O System adds the ID to the file's access list, unless the access list is full (contains three entries). If the access parameter described in the next paragraph is zero, this call removes the ID from the access list.

access — A BYTE defining the new access rights to be assigned to the specified user. If the entire BYTE is set to zero, the Extended I/O System removes the specified ID from the access list of the file. If the BYTE is nonzero, the meaning of the various bit settings depend upon whether the file is a data file or a directory file. The following two tables correlate the bit position and the kind of access. (System calls that start with "A$", like A$READ, are part of the Basic I/O System.)

If the bit is set to 1, access is to be granted. If the bit is set to 0, access is to be denied. (Bit 0 is low-order bit.)

DATA FILE ACCESS RIGHTS

Bit	Access
0	Delete--permission to delete the entire file by using the S$DELETE$FILE or A$DELETE$FILE system calls. Also allows changing the name of the file by using the S$RENAME$FILE or A$RENAME$FILE system call.
1	Read--permission to read data from the file by using the S$READ$MOVE or A$READ system call.

2	Append--permission to write information only at the end of the file by using the S$WRITE$MOVE or A$WRITE system call. This does not include permission to write over information already in the file or permission to truncate the file.
3	Update--permission to write over any information in the file by using the S$WRITE$MOVE or A$WRITE system calls, and permission to truncate the file using the S$TRUNCATE$FILE or A$TRUNCATE system call. This does not include permission to add information to the end of the file.
4-7	Reserved. Set to zero.

DIRECTORY ACCESS RIGHTS

Bits	Access
0	Delete--permission to delete the directory by using the A$DELETE$FILE or S$DELETE$FILE system call. Also allows changing the name of the directory by using the A$RENAME$FILE or S$RENAME$FILE system call.
1	Display--permission to read information from the directory by using the A$READ, A$GET$DIRECTORY$ENTRY, or S$READ$MOVE system call.
2	Add entry--permission to add files to the directory by using the A$CREATE$FILE, A$CREATE$DIRECTORY, A$RENAME$FILE, S$CREATE$FILE, S$CREATE$DIRECTORY, or S$RENAME$FILE system call. This does not include permission to change existing entries.
3	Change entry--permission to change the access list associated with a file contained in the directory. In other words, permission to use the A$CHANGE$ACCESS or S$CHANGE$ACCESS system call. This does not include permission to add new entries or change the access list of the directory in which the file is cataloged.

4-7	Reserved. Set to zero.

Output Parameter

except$ptr	A POINTER to a WORD where the Extended I/O System returns the condition code.

Description

The S$CHANGE$ACCESS system call allows a task to change the access rights associated with named data or directory files. This system call can be used on any named file, including those created by the Basic I/O System.

For a task to be able to change the access rights associated with a file, the task's job must meet at least one of the following criteria:

- One of the IDs in the job's default user object is the owner of the file, or is the System Manager ID.
- One of the IDs in the job's default user object has change-entry access to the parent directory of the file.

For more information about owners, access rights, and default user objects, refer to Chapter 4 of the *Extended iRMX II Estended I/O System User's Guide*.

Condition Codes

E$OK	0000H	No exceptional conditions.
E$ALREADY$-ATTACHED	0038H	The Extended I/O System cannot attach the device containing the file because the Basic I/O System has already attached the device.
E$CONTEXT	0005H	The calling task's job is not an I/O job.
EDEVDETACHING	0039H	The device containing the specified file is being detached.
E$DEVFD	0022H	The Extended I/O System attempted the physical attachment of a device that had formerly been only logically attached. In the process, it found that the device and the device driver specified in the logical attachment were incompatible.
E$FACCESS	0026H	The job containing the calling task meets none of the prerequisites for using this system call. None of the IDs in the job's default user object is the owner of the file, nor does any have change-entry access to the file's parent directory.

E$FNEXIST	0021H	One of the following conditions is true: • A file in the specified path, or the target file itself, does not exist or is marked for deletion. • The physical device was not found. The device was specified by the original call to A$PHYSICAL$ATTACH$DEVICE and is indicated in this call by the path$ptr parameter.
E$FTYPE	0027H	The specified path is attempting to use a data file as a directory.
E$IFDR	002FH	The file driver associated with this connection is the physical or stream file driver. However, the call is compatible with the named file driver only.
E$ILLVOL	002DH	The Extended I/O System attempted the physical attachment of a device that had formerly been only logically attached. In the process, it examined the volume label and found that the volume does not contain named files. This prevented the call from completing physical attachment because the named file driver was requested during logical attachment.
E$INVALID$FNODE	003DH	The fnode for the specified file is invalid. The file cannot be accessed; you should delete it.
EIOHARD	0052H	A hard I/O error occurred. A retry is probably useless.
EIOOPRINT	0053H	The device was off-line. Operator intervention is required.
EIOSOFT	0051H	A soft I/O error occurred. The I/O System tried to perform the operation a number of times and failed (the number of retries is a configuration parameter). Another retry might still be successful.
EIOUNCLASS	0050H	An unknown type of I/O error occurred.
EIOWRPROT	0054H	The volume is write-protected.
EIOMEM	0042H	The Basic I/O System job does not currently have a block of memory large enough to allow this system call to run to completion.
E$LIMIT	0004H	At least one of the following is true: • The user object or the calling task's job is already involved in 255 (decimal) I/O operations.

		• The calling task's job is not an I/O job.
ELOGNAME$NEXIST		0045H The specified path contains an explicit logical name, but the call was unable to find this name in the object directories of the calling task's local job, the global job, or the root job.
ELOGNAME$-SYNTAX	0040H	The specified logical name contains at least one of the following syntax errors: • The specified path starts with a colon (:), indicating that it contains a logical name. But the call was unable to find a second colon to delimit the logical name. • The specified path contains a logical name that is either longer than 12 characters (including colons), has no characters, or contains invalid characters.
E$MEDIA	0044H	The device containing the specified file is not on-line.
E$MEM	0002H	The memory available to the calling task's job is not sufficient to complete the call.
ENOPREFIX	8022H	You did not specify an explicit prefix (logical name), and the default prefix for the calling task's job is either undefined, or it is not a valid device connection or file connection.
ENOTCONFIGURED	0008H	This system call is not part of the present configuration.
ENOTLOG$NAME	8040H	The specified path contains a logical name that refers to an object that is neither a device connection nor a file connection.
ENOUSER	8021H	The calling task's job does not have a default user, or its default user is not a user object.
E$PARAM	8004H	The Extended I/O System attempted the physical attachment of a device that had formerly been only logically attached. The logical attachment referred to a file driver (named, physical, or stream) that is not configured into your system. Therefore, physical attachment is not possible.
E$PATHNAME$-SYNTAX	003EH	The specified pathname contains invalid characters.

E$SUPPORT	0023H	At least one of the following is true: • The calling task attempted to change access for a file other than a named file. • The calling task attempted to add another user ID to the file's access list, but the list already contains three entries. The task must delete an entry before it can add another. • The connection specified in the call is not contained in the job making the call.

The S$CLOSE system call closes an open connection to a named, physical, or stream file.

```
CALL RQ$S$CLOSE(connection, except$ptr);
```

Input Parameter

connection	A TOKEN for a file connection that is currently open and was opened by the S$OPEN system call.

Output Parameter

except$ptr	A POINTER to a WORD where the Extended I/O System returns the condition code.

Description

The S$CLOSE system call closes a connection that has been opened by the S$OPEN system call. It performs the following steps:

1. It waits until all currently running I/O operations for the file are completed.
2. It ensures that any information in a partially filled output buffer is written to the file.
3. It releases any buffers associated with the file.
4. It closes the connection to the file, deleting neither the file nor the connection.

The Extended I/O System performs no access checking before closing the connection.

The S$CLOSE system call cannot be used to close connections that were opened by the Basic I/O System. If your task attempts to do this, the Extended I/O System returns an E$CONN$NOT$OPEN exception code.

Condition Codes

E$OK	0000H	No exceptional conditions.
E$CANNOT$CLOSE	0041H	An error occurred while flushing data from EIOS buffers to an output device.

E$CONN$NOT$OPEN	0034H	One of the following conditions is true: • The connection is not open. • The connection was opened by A$OPEN rather than S$OPEN.
E$EXIST	0006H	The connection parameter is not a token for an existing object.
EIOHARD	0052H	A hard I/O error occurred. A retry is probably useless.
EIOMODE	0056H	One of the following is true: • A tape drive attempted to perform a read operation before the previous write operation completed. • A tape drive attempted to perform a write operation before the previous read operation completed.
EIONO$DATA	0055H	A tape drive attempted to read the next record, but it found no data.
EIOOPRINT	0053H	The device was off-line. Operator intervention is required.
EIOSOFT	0051H	A soft I/O error occurred. The I/O System tried to perform the operation a number of times and failed (the number of retries is a configuration parameter). Another retry might still be successful.
EIOUNCLASS	0050H	An unknown type of I/O error occurred.
EIOWRPROT	0054H	The volume is write-protected.
E$LIMIT	0004H	At least one of the following is true: • The calling task's job is not an I/O job. • The calling task's job is already involved in 255 (decimal) I/O operations.
E$MEM	0002H	The memory available to the calling task's job is not sufficient to complete the call.
ENOTCONFIGURED	0008H	This system call is not part of the present configuration.
ENOTCONNECTION	8042H	The connection parameter is a token for an object that is not a connection.
E$SUPPORT	0023H	The specified connection was not created by a task in the calling task's job.

The S$CREATE$DIRECTORY system call creates a new directory file.

```
connection = RQS$CREATE$DIRECTORY(path$ptr, except$ptr);
```

Input Parameter

path$ptr	A POINTER to a STRING containing the pathname of the new directory.

Output Parameters

connection	A TOKEN that represents a connection to the new directory. You can use this TOKEN as a parameter in system calls that access the directory.
except$ptr	A POINTER to a WORD where the Extended I/O System returns the condition code.

Description

A task invokes this system call to create a new named-file directory. After creation, the new directory contains no entries. This system call automatically adds a new entry to the parent directory. The new directory is compatible with directories created by the Basic I/O System.

Positioning the Directory

The calling task must use the path$ptr parameter to specify the location of the new directory within the named file structure. The location indicated by the path must not be occupied. In other words, this system call can be used only to obtain connections to new, rather than existing, directories.

The default user object for the calling task's job must have add-entry access to the parent of the new directory. If the creation is successful, the first ID in the job's default user object (the owner ID) becomes the owner of the file.

The entry in the parent directory for the newly created directory provides the owner of the new directory with full access (the ability to Delete, List, Add, and Change entries) to the new directory.

Condition Codes

E$OK	0000H	No exceptional conditions.
E$ALREADY$-ATTACHED	0038H	The Extended I/O System cannot attach the device containing the file because the Basic I/O System has already attached the device.
E$CONTEXT	0005H	The calling task's job is not an I/O job.
EDEVDETACHING	0039H	The device containing the specified file is in the process of being detached.
E$DEVFD	0022H	The Extended I/O System attempted to physically attach a device that had formerly been only logically attached. In the process, it found that the device and the device driver specified in the logical attachment were incompatible.
E$FACCESS	0026H	The user object associated with the calling task's job does not have add-entry access to the parent directory.
E$FEXIST	0020H	The file already exists.
E$FNEXIST	0021H	At least one of the following is true: • A file in the specified path does not exist or is marked for deletion. • The device specified in the call is not part of the current configuration.
E$FNODE$LIMIT	003FH	The volume already contains the maximum number of files. No more fnodes are available for new files.
E$FTYPE	0027H	The specified path is attempting to use a data file as a directory.
E$ILLVOL	002DH	The Extended I/O System attempted to physically attach a device that had formerly been only logically attached, and found that the volume does not contain named files. This prevented the call from completing physical attachment because the named file driver was requested during logical attachment.
E$INVALID$FNODE	003DH	The fnode for a directory in the specified pathname is invalid. The file cannot be accessed; you should delete it.
EIOHARD	0052H	A hard I/O error occurred. This means that a retry is probably useless.

EIOOPRINT	0053H	The device was off-line. Operator intervention is required.
EIOSOFT	0051H	A soft I/O error occurred. The I/O System tried to perform the operation a number of times and failed (the number of retries is a configuration parameter). Another retry might still be successful.
EIOUNCLASS	0050H	An unknown type of I/O error occurred.
EIOWRPROT	0054H	The volume is write-protected.
EIOMEM	0042H	The Basic I/O System job does not currently have a block of memory large enough to allow this system call to run to completion.
E$LIMIT	0004H	At least one of the following is true: • The user object or the calling task's job is already involved in 255 (decimal) I/O operations. • The calling task's job is not an I/O job.
ELOGNAME$-NEXIST	0045H	The specified path contains an explicit logical name, but the call was unable to find this name in the object directories of the calling task's local job, the global job, or the root job.
ELOGNAME$-SYNTAX	0040H	The specified logical name contains at least one of the following syntax errors: • The specified path starts with a colon (:), indicating that it contains a logical name. But the call was unable to find a second colon to delimit the logical name. • The specified path contains a logical name that is either longer than 12 characters (excluding colons), has a length of zero characters, or contains invalid characters.
E$MEDIA	0044H	The device containing the specified file is not on-line.
E$MEM	0002H	The memory available to the calling task's job is not sufficient to complete the call.
ENOPREFIX	8022H	You did not specify an explicit prefix (logical name), and the default prefix for the calling task's job is either undefined, or it is not a valid device connection or file connection.

ENOTCONFIGURED	0008H	This system call is not part of the present configuration.
ENOTLOG$NAME	8040H	The specified path contains a logical name that refers to an object that is neither a device connection nor a file connection.
ENOUSER	8021H	The calling task's job does not have a default user, or its default user is not a user object.
E$PARAM	8004H	The Extended I/O System attempted the physical attachment of a device that had formerly been only logically attached. The logical attachment referred to a file driver (named, physical, stream, or remote) that is not configured into your system, so physical attachment is not possible.
E$PATHNAME$-SYNTAX	003EH	The specified pathname contains invalid characters.
E$SUPPORT	0023H	The NO ALLOCATE option is configured into the BIOS. You cannot create any directories on this volume.
E$SPACE	0029H	The volume is full.

The S$CREATE$FILE system call creates a new physical, stream, or named data file. It cannot create a named directory file.

```
connection = RQ$S$CREATE$FILE(path$ptr, except$ptr);
```

Input Parameter

path$ptr	A POINTER to a STRING that contains the pathname of the file to be created. The format of this pathname depends on the kind of file being created. Refer to Chapter 4 of the *Extended iRMX II Extended I/O System User's Guide* for a discussion of named file paths, to Chapter 5 for physical files, and to Chapter 6 for stream file paths.

Output Parameters

connection	The TOKEN that represents the connection to the new file.
except$ptr	A POINTER to a WORD where the Extended I/O System returns a condition code.

Description

A task invokes this system call to create a physical, stream, or named data file, or to attach an existing file. This system call cannot be used to create or to attach a directory. (The Extended I/O System provides the S$CREATE$DIRECTORY system call for that purpose.) The file created by this system call is compatible with files created by the Basic I/O System.

If the file specified by the path$ptr parameter already exists, the Extended I/O System attempts to truncate the file to zero length and return a connection to the empty file. That is, S$CREATE$FILE acts exactly as an A$ATTACH$FILE followed by a call to S$TRUNCATE$FILE. The owner and the accessor list for the file remain unchanged.

If the file already exists, the call succeeds only if both of the following conditions are true:

- All connections to the file that are currently open allow sharing with writers.
- An ID in the default user object of the calling task's job has update access to the existing file. (This requirement applies to named files only.)

If you wish to prevent the file from being truncated accidentally, use the S$ATTACH$FILE system call; if the call to S$ATTACH$FILE returns an exception code indicating the file does not exist, you can safely use S$CREATE$FILE.

Specifying the Kind of File to be Created

The path$ptr parameter does more than simply indicate the path of the file being created. It also tells the Extended I/O System what kind of file (stream, physical, or named data) to create. The correlation between file paths and the kinds of files is discussed in detail in Chapters 4, 5, and 6 of the *Extended iRMX II Extended I/O System User's Guide*.

Special Considerations for Named Files

These special considerations relate to named files:

- Your task must tell the Extended I/O System which directory is to be the parent of the new named file.
- To create a named file, an ID in the default user object for the calling task's job must have add-entry access to the parent directory.
- The first ID in the default user object of the calling task's job becomes the owner of the new file. The owner has full access (the owner can delete, read, append, and update the file).

Temporary Named Files

If your task invokes this system call with the path of an existing directory file, the Extended I/O System creates a temporary named data file on the device that contains the directory file. This temporary file differs from other named data files in two ways. First, the file is automatically marked for deletion, so that the Extended I/O System deletes the file as soon as your application code deletes all connections to the file. Second, the file is created without a path, so it can be accessed only through a connection.

Two access considerations apply to temporary files:

- First, any task can create a temporary file by referring to any directory. This is true because the temporary files are not listed as ordinary entries in the directory, so no add-entry access is required for the directory.
- Second, the owner of the temporary file (the first ID in the default user object of the calling task's job) has full access to the file.

Device Considerations

Every file, regardless of kind, has an associated device. Even stream files, which have no physical devices, use the device connection to the stream file pseudo-device.

Before any file can be created, its associated device must be attached to the system.

There are two ways to attach devices to the system. One is to specify the attachment during configuration. (For more information, refer to extended iRMX II Operating System user guides).

The second way is to attach a device while the system is running using the LOGICAL$ATTACH$DEVICE system call.

Condition Codes

E$OK	0000H	No exceptional conditions.
E$ALREADY$-ATTACHED	0038H	The Extended I/O System cannot attach the device containing the file because the Basic I/O System has already attached the device.
E$CONTEXT	0005H	The calling task's job is not an I/O job.
EDEVDETACHING	0039H	The device containing the specified file is being detached.
E$DEVFD	0022H	The Extended I/O System attempted to physically attach a device that had formerly been only logically attached, and found that the device and the device driver specified in the logical attachment were incompatible.
E$FACCESS	0026H	At least one of the following is true: • The default user object associated with the calling task's job does not have add-entry access to the parent directory. • The default user object associated with the calling task's job does not have update access to the existing file with the specified pathname.
E$FNEXIST	0021H	At least one of the following is true: • A file in the specified path does not exist or is marked for deletion. • The physical device specified in the call was not found.
E$FNODE$LIMIT	003FH	The volume already contains the maximum number of files. No more fnodes are available for new files.
E$FTYPE	0027H	The specified path is attempting to use a data file as a directory.

E$ILLVOL	002DH	The Extended I/O System attempted to physically attach a device that had formerly been only logically attached, and found that the volume does not contain named files. This prevented the call from completing physical attachment.
E$INVALID$FNODE	003DH	The fnode for a directory in the specified pathname is invalid. The file cannot be accessed; you should delete it.
EIOHARD	0052H	A hard I/O error occurred. A retry is probably useless.
EIOOPRINT	0053H	The device was off-line. Operator intervention is required.
EIOSOFT	0051H	A soft I/O error occurred. The I/O System tried to perform the operation a number of times and failed (the number of retries is a configuration parameter). Another retry might still be successful.
EIOUNCLASS	0050H	An unknown type of I/O error occurred.
EIOWRPROT	0054H	The volume is write-protected.
EIOMEM	0042H	The Basic I/O System job does not currently have a block of memory large enough to allow this system call to run to completion.
E$LIMIT	0004H	At least one of the following is true: • The calling task has reached the object's limit. • The user object or the calling task's job is already involved in 255 (decimal) I/O operations. • The calling task's job is not an I/O job.
ELOGNAME$-NEXIST	0045H	The specified path contains an explicit logical name, but the call was unable to find this name in the object directories of the calling task's local job, the global job, or the root job.
ELOGNAME$-SYNTAX	0040H	The specified logical name contains at least one of the following syntax errors: • The specified path starts with a colon (:), indicating that it contains a logical name. But the call was unable to find a second colon to delimit the logical name.

		• The specified path contains a logical name that is either longer than 12 characters (including colons), does not contain at least one character, or contains invalid characters.
E$MEDIA	0044H	The device containing the specified file is not on-line. The media maybe inserted incorrectly (upside down).
E$MEM	0002H	The memory available to the calling task's job is not sufficient to complete the call.
ENOPREFIX	8022H	You did not specify an explicit prefix (logical name), and the default prefix for the calling task's job is either undefined, or it is not a valid device connection or file connection.
ENOTCONFIGURED	0008H	This system call is not part of the present configuration.
ENOTLOG$NAME	8040H	The specified path contains a logical name that refers to an object that is neither a device connection nor a file connection.
ENOUSER	8021H	The calling task's job does not have a default user object, or the object cataloged in R?IOUSER is not a user object.
E$PARAM	8004H	The Extended I/O System attempted to physically attach a device that had formerly been only logically attached. The logical attachment referred to a file driver (named, physical, or stream) that is not configured into your system, so the physical attachment is not possible.
E$PATHNAME$-SYNTAX	003EH	The specified pathname contains invalid characters.
E$SHARE	0028H	You are trying to create a file that already exists. The Extended I/O System must truncate the existing file to zero length to do the create. This truncate to zero length failed for one or more of the following reasons: • Another open connection does not allow sharing with writers. • The default user for the calling task's job does not have update access to the file.
E$SPACE	0029H	The volume is full.

E$SUPPORT	0023H	One of the following is true: • The NO CREATE FALSE option is configured into the BIOS. • The NO TRUNCATE option is configured into the BIOS.

The S$DELETE$CONNECTION system call deletes a file connection. It cannot delete a device connection.

```
CALL RQ$S$DELETE$CONNECTION(connection, except$ptr);
```

Input Parameter

connection	A TOKEN for the file connection to be deleted.

Output Parameter

except$ptr	A POINTER to a WORD where the Extended I/O System returns the condition code.

Description

This system call deletes a file connection, but it cannot delete a device connection. If the connection is open, the S$DELETE$CONNECTION system call automatically closes it before deleting it.

If the file has been marked for deletion (by a previous system call) and there are no more connections to the file, then S$DELETE$CONNECTION deletes the file.

The Extended I/O System does not check access before deleting a connection.

The S$DELETE$CONNECTION system call can be used with connections that were created by the Basic I/O System as long as the connections meet the requirements discussed in the *Extended iRMX II Basic I/O System User's Guide*, Appendix E.

Condition Codes

E$OK	0000H	No exceptional conditions.
E$EXIST	0006H	The connection parameter is not a token for an existing object.
EIOHARD	0052H	A hard I/O error occurred. A retry is probably useless.

EIOMODE	0056H	One of the following is true: • A tape drive attempted to perform a read operation before the previous write operation completed. • A tape drive attempted to perform a write operation before the previous read operation completed.
EIONO$DATA	0055H	A tape drive attempted to read the next record, but it found no data.
EIOOPRINT	0053H	The device was off-line. Operator intervention is required.
EIOSOFT	0051H	A soft I/O error occurred. The I/O System tried to perform the operation a number of times and failed (the number of retries is a configuration parameter). Another retry might still be successful.
EIOUNCLASS	0050H	An unknown type of I/O error occurred.
EIOWRPROT	0054H	The volume is write-protected.
E$LIMIT	0004H	At least one of the following is true: • The associated job or the job's default user object is already involved in 255 (decimal) I/O operations. • The calling task's job is not an I/O job.
E$MEM	0002H	The memory available to the calling task's job is not sufficient to complete the call.
ENOTCONFIGURED	0008H	This system call is not part of the present configuration.
ENOTCONNECTION	8042H	The connection parameter is a token for an object that is not a file connection.
E$SUPPORT	0023H	The specified connection was not created by a task in this job.

The S$DELETE$FILE system call deletes a stream, named data, or named directory file. This system call cannot delete a physical file.

```
CALL RQ$S$DELETE$FILE(path$ptr, except$ptr);
```

Input Parameter

path$ptr — A POINTER to a STRING that specifies the path for the file to be deleted. The form of the path depends upon the kind of file. (See the EXTENDED iRMX II BASIC I/O SYSTEM USER'S GUIDE for information on path syntax.)

Output Parameter

except$ptr — A POINTER to a WORD where the Extended I/O System returns a condition code.

Description

A task can use this system call whenever the task needs to delete a stream, named data, or named directory file. This system call marks the specified file for deletion, but the Extended I/O System postpones deletion until the following conditions are met:

- For stream and named data files, there is only one condition. The deletion occurs as soon as no connections to the file remain. Your tasks can use the S$DELETE$CONNECTION system call to delete connections.
- For named directories there are two conditions. The directory must be empty, and no connections to the directory can remain. The Extended I/O System deletes marked directories as soon as both of these conditions are met.

This system call can delete files created by the Basic I/O System as well as those created by the Extended I/O System. Refer to the *Extended iRMX II Basic I/O User's Guide*, Appendix E for a general discussion of compatibility between the Extended and Basic I/O Systems.

If the task attempts to delete a named data or directory file, the default user object of the task's job must have deletion access to the file.

Condition Codes

E$OK	0000H	No exceptional conditions.
E$ALREADY$-ATTACHED	0038H	The specified device is already attached.
E$CONTEXT	0005H	The calling task's job is not an I/O job.
EDEVDETACHING	0039H	The device containing the specified file is in the process of being detached.
E$DEVFD	0022H	The Extended I/O System attempted to physically attach a device that had formerly been only logically attached, and found that the device and the device driver specified in the logical attachment were incompatible.
EDIRNOT$EMPTY	0031H	Your task is attempting to delete a directory that is not empty.
E$FACCESS	0026H	At least one of the following is true: • The default user object associated with the calling task's job does not have delete access to the specified file. • The call is attempting to delete a bit map file or the root directory.
E$FNEXIST	0021H	At least one of the following is true: • A file in the specified path, or the target file itself, does not exist or is marked for deletion. • The physical device was not found. The device was specified by the original call to A$PHYSICAL$ATTACH$DEVICE and is indicated in this call by the path$ptr parameter.
E$FTYPE	0027H	The specified path contains a file name that should be the name of a directory, but is not. (Except for the last file, each file in a path must be a directory.)
E$ILLVOL	002DH	The Extended I/O System attempted to physically attach a device that had formerly been only logically attached, and found that the volume does not contain named files. This prevented the call from completing physical attachment because the named file driver was requested during logical attachment.
E$IFDR	002FH	The specified file is a physical file.

E$INVALID$FNODE	003DH	The fnode associated with a file is either marked not allocated, or the fnode number is out of range. This file should be deleted.
EIOHARD	0052H	A hard I/O error occurred. A retry is probably useless.
EIOOPRINT	0053H	The device was off-line. Operator intervention is required.
EIOSOFT	0051H	A soft I/O error occurred. The I/O System tried to perform the operation a number of times and failed (the number of retries is a configuration parameter). Another retry might still be successful.
EIOUNCLASS	0050H	An unknown type of I/O error occurred.
EIOWRPROT	0054H	The volume is write-protected.
EIOMEM	0042H	The Basic I/O System job does not currently have a block of memory large enough to allow this system call to run to completion.
E$LIMIT	0004H	At least one of the following is true: • Either the user object or the calling task's job is already involved in 255 (decimal) I/O operations. • The calling task's job is not an I/O job.
ELOGNAME$-NEXIST	0045H	The specified path contains an explicit logical name, but the call was unable to find this name in the object directories of the calling task's local job, global job, or the root job.
ELOGNAME$-SYNTAX	0040H	The specified logical name contains at least one of the following syntax errors: • The specified path starts with a colon (:), indicating that it contains a logical name. But the call was unable to find a second colon to delimit the logical name. • The specified path contains a logical name that is either longer than 12 characters (including colons), contains no characters, or contains invalid characters.
E$MEDIA	0044H	The device containing the specified file is off-line.
E$MEM	0002H	The memory available to the calling task's job is not sufficient to complete the call.

ENOPREFIX	8022H	You did not specify an explicit prefix (logical name), and the default prefix for the calling task's job is either undefined, or it is not a valid device connection or file connection.
ENOTCONFIGURED	0008H	This system call is not part of the present configuration.
ENOTLOG$NAME	8040H	The specified path contains a logical name that refers to an object that is neither a device connection nor a file connection.
ENOUSER	8021H	The calling task's job does not have a default user object, or the object cataloged in R?IOUSER is not a user object.
E$PARAM	8004H	The Extended I/O System attempted to physically attach a device that is logically attached. That logical attachment refers to a file driver (named, physical, or stream) that is not configured into your system. Therefore, physical attachment is not possible.
E$PATHNAME$-SYNTAX	003EH	The specified pathname contains invalid characters.
E$SUPPORT	0023H	The task is attempting to delete a physical file.

The SGETCONNECTION$STATUS system call provides status information about file and device connections.

```
CALL RQ$S$GET$CONNECTION$STATUS(connection, info$ptr, except$ptr);
```

Input Parameter

connection — A TOKEN for the connection whose status is desired.

Output Parameters

info$ptr — A POINTER to a structure in which the Extended I/O System places the status information. You can provide the memory for this structure by requesting an extended iRMX II segment, or by reserving it in your code. The structure must have the following form:

```
DECLARE connection$info STRUCTURE(
                file$driver         BYTE,
                flags               BYTE,
                open$mode           BYTE,
                share$mode          BYTE,
                file$pointer        DWORD,
                access              BYTE,
                number$buffers      BYTE,
                buffer$size         WORD,
                seek                BOOLEAN)
```

where

file$driver — Identifies the kind of file associated with the connection.

1 physical file
2 stream file
4 named file
5 remote

flags — Indicates the kind of connection this is. If Bit 1 is one, the connection is capable of being opened. If Bit 2 is one, the connection is a device connection. (Bit zero is the low-order bit.)

open$mode — Indicates the purpose for which the connection was opened. This applies only to file connections.

0	closed
1	open for reading only
2	open for writing only
3	open for both reading and writing

share$mode — Indicates who can share the connection. Applies to both device and file connections.

0	cannot be shared
1	share with readers only
2	share with writers only
3	share with anybody

file$pointer — A 32-bit offset from the beginning of the file where the next I/O operation will be performed.

access — The access rights that were computed when the connection was created. This information applies only to connections for named files, and the interpretation of the information depends upon whether the file is a data file or a directory. Access is represented as a bit mask. In the following tables, access is granted if a bit is set to one (bit zero is the low-order bit.).

Bit	Data File	Directory
0	Delete	Delete
1	Read	List
2	Append	Add Entry
3	Update	Change Entry
4-7	Reserved	Reserved

number$buffers — The number of buffers used with this connection. This applies only to file connections.

buffer$size — The size, in bytes, of each buffer used with the connection.

seek — Tells whether or not the SEEK function can be used with this connection. Zero means no, and 0FFh means yes.

except$ptr — A POINTER to a WORD where the Extended I/O System returns the condition code.

Description

The SGETCONNECTION$STATUS system call allows a task to obtain status information about file connections and device connections that were created by either the Basic I/O System or the Extended I/O System. The nature of the returned information depends upon whether the connection is for a file or a device. Some of the information also depends on the kind of file associated with the connection.

The Extended I/O System does not check access before returning status information.

Although you can use this system call with connections created by the Basic I/O System, you must adhere to the restrictions described in the *Extended iRMX II Basic I/O User's Guide*, Appendix E.

Condition Codes

E$OK	0000H	No exceptional conditions.
E$CONN$NOT$OPEN	0034H	The connection was opened by the A$OPEN system call rather than the S$OPEN system call.
E$EXIST	0006H	The connection parameter is not a token for an existing job.
E$IFDR	002FH	An invalid file driver request occurred.
E$LIMIT	0004H	At least one of the following is true: • The calling task has reached its object limit. • Either the calling task's job, or the job's default user object, is already involved in 255 (decimal) I/O operations. • The calling task's job is not an I/O job.
E$MEM	0002H	The memory available to the calling task's job is not sufficient to complete the call.
ENOTCON-FIGURED	0008H	This system call is not part of the present configuration.
ENOTCONNECTION	8042H	The connection parameter is a token for an object that is not a connection.
E$SUPPORT	0023H	The specified connection was not created by a task in the calling task's job.

The RQSGET$DIRECTORY$ENTRY system call returns a directory entry name to the caller. A directory entry name is a single path component for a file whose parent is the directory.

```
CALL RQ$S$GET$DIRECTORY$ENTRY(dir$name$ptr, entry$num, name$ptr,
                              except$pt);
```

Input Parameters

dir$name$ptr	A POINTER to a STRING containing the directory pathname. This pathname can be up to 255 characters long.
entry$num	A WORD giving the entry number of the desired file name. Entries in a directory are numbered sequentially starting from zero. The E$EMPTY$ENTRY condition code will be returned if there is no directory entry associated with the number.

Output Parameter

name$ptr	A POINTER to a buffer where the system will return the entry name. This name, a maximum length of 14 BYTES, corresponds to the entry number given by the user in the entry$num parameter.
except$ptr	A POINTER to a WORD where the condition code will be returned.

Description

The SGETDIRECTORY$ENTRY system call applies to named files only. When called, it returns the file name associated with a specified directory entry. This name is a single subpath component for a file whose parent is the designated directory. As an alternative to using this system call, an application task can open and read a directory file.

NOTE

The caller must have List access to the designated directory.

Condition Codes

E$OK	0000H	No exceptional conditions.
EDIREND	0025H	The entry$num parameter is greater than the number of entries in the directory.
E$EMPTY$ENTRY	0024H	The file entry designated in the call is empty.
E$FACCESS	0026H	The specified connection is not qualified for list access to the directory.
E$FTYPE	0027H	The specified connection does not refer to a directory.
E$IFDR	002FH	This system call applies only to named directories, but the STRING pointed to by dir$name$ptr specifies another type of file.
E$IO	002BH	An I/O error occurred that might have prevented the operation from completing.
E$LIMIT	0004H	The calling task's job has already reached its object limit.
E$MEM	0002H	The memory available to the calling task's job is not sufficient to complete this call.
ENOTCONFIGURED	0008H	This system call is not part of the present configuration.

The SGETFILE$STATUS system call allows a task to obtain information about a physical, stream, or named file.

```
CALL RQ$S$GET$FILE$STATUS(path$ptr, info$ptr, except$ptr);
```

Input Parameter

path$ptr — A POINTER to a STRING that contains the path for the file. The format of this path varies from one kind of file to another. Refer to Chapters 4, 5, or 6 of the *Extended iRMX II Extended I/O User's Guide* for path syntax.

Output Parameters

info$ptr — A POINTER to a structure where the Extended I/O System returns the status information. You must allocate this memory, either in your program code space or as an extended iRMX II segment. The structure has the form described here.

The information in the first part of this structure--down to the device$connections field--is returned for any file (physical, stream, or named), but information from the file$id field to the end of the structure is present only for named files. The contents of the named$file field indicate whether the file is a named file.

```
DECLARE file$info STRUCTURE(
            device$share            WORD,
            number$connections      WORD,
            number$reader           WORD,
            number$writer           WORD,
            share                   BYTE,
            named$file              BYTE,
            device$name(14)         BYTE,
            file$drivers            WORD,
            functions               BYTE,
            flags                   BYTE,
            device$granularity      WORD,
            device$size             DWORD,
            device$connections      WORD,
```

Information from this point on is returned only if the file is a named file.

```
file$id              WORD,
file$type            BYTE,
file$granularity     BYTE,
owner$id             WORD,
create$time          DWORD,
access$time          DWORD,
modify$time          DWORD
file$size            DWORD,
file$blocks          DWORD,
volume$name(6)       BYTE,
volume$granularity   WORD,
volume$size          DWORD,
accessor$count       WORD,
owner$access         BYTE;
```

The meanings of these fields are

device$share	Indicates whether or not the device can be shared. Currently, this word is always set to 1, indicating that all devices can be shared.
number$connections-	The number of connections to the file.
number$reader	The number of connections currently open for reading.
number$writer	The number of connections currently open for writing.
share	The current shared status of the file; possible values are 0 Private use only 1 Share with readers only 2 Share with writers only 3 Share with all users
named$file	Tells whether this structure contains any information beyond the device$connections field. 0FFh means yes and 0 means no.
device$name	The name of the physical device where this file resides. This name is padded with blanks. To ensure the uniqueness of device names, they should not be more than 14 characters in length.

file$drivers

A bit map that tells what kinds of files can reside on this device. If bit n is on, then file driver n + 1 can be used. Bit 0 is the low-order bit.

Bit	Driver No.	Driver
0	1	Physical file
1	2	Stream file
2	3	Reserved
3	4	Named file
4	5	Remote file

functions

A bit map that describes the functions supported by the device where this file resides. A bit set to one indicates the corresponding function is supported. Bit 0 is the low-order bit.

Bit	Function
0	F$READ
1	F$WRITE
2	F$SEEK
3	F$SPECIAL
4	F$ATTACH$DEV
5	F$DETACH$DEV
7	F$CLOSE

flags

Meaningful only for diskette drives. This field is interpreted as follows. (Bit 0 is the low-order bit.)

Bit	Meaning
0	0 = bits 1-7 not significant 1 = bits 1-7 are significant
1	0 = single density 1 = double density
2	0 = single sided 1 = double sided
3	0 = 8-inch diskette 1 = 5 1/4-inch diskette
4	0 = standard diskette, meaning that track 0 is single-density with 128 byte sectors 1 = a nonstandard diskette or not a diskette
5-7	reserved

device$granularity

The granularity, in bytes, of the device where this file resides.

device$size

The storage capacity of the device, in bytes.

device$connections	The number of connections to the device.

The information from here to the end of the structure is returned only for named files, as indicated by a value of 0FFh in the named$file field.

file$id	A number that distinguishes this file from all other files on the same device.
file$type	The file type: 6 means directory file and 8 means data file.
file$granularity	The file granularity, as a multiple of volume$granularity. For example, if file$granularity is 2 and volume$granularity is 256, then the file's granularity is 512.
owner$id	The first ID in the creating task's default user object.
create$time	The time and date when the file was created. Whether the operating system maintains this field is a configuration option.
access$time	The time and date when the file was last accessed. Whether the operating system maintains this field is a configuration option.
modify$time	The time and date when the file was last modified. Whether the operating system maintains this field is a configuration option.
file$size	The total size of the file, in bytes.
file$blocks	The number of volume blocks allocated to this file. A volume block is a contiguous area of storage that contains volume$granularity bytes of data.
volume$name	The left-adjusted, null-padded ASCII name for the volume containing this file.
volume$granularity	The volume granularity, in bytes.
volume$size	The storage capacity, in bytes, of the volume on which this file is stored.
accessor$count	The number of IDs in the creating task's default user object.
owner$access	The access rights to this file that are currently held by the owner. The access rights are encoded in a bit mask that you can interpret by using the following table. Remember that Bit 0 is the low-order bit, and that access is granted if the corresponding bit is set to 1.

Bit	Data File	Directory
0	Delete	Delete
1	Read	List
2	Append	Add Entry
3	Update	Change Entry
4-7	Reserved	Reserved

except$ptr — A POINTER to a WORD where the Extended I/O System returns the condition code.

Description

This system call provides the calling task with information about the status of a file. Fields through the device$connections field are always returned if the call is successful. Fields following the device$connections field are returned only when the file being referred to is a named file, as indicated by the named$file field being 0FFh.

The Extended I/O System does not check access before returning file status information.

This system call can be used with any file, including those created by the Basic I/O System. However, because of the asynchronous nature of some of the Basic I/O System calls, there is some chance that the information returned might be inaccurate. For instance, if your application code invokes the SGETFILE$STATUS system call while the Basic I/O System is processing an A$WRITE for the same file, the values returned in the file size fields might be incorrect. Refer to the *Extended iRMX II Basic I/O User's Guide*, Appendix E for a more general discussion of compatibility between the Extended and Basic I/O Systems.

Condition Codes

E$OK	0000H	No exceptional conditions.
E$ALREADY$-ATTACHED	0038H	The Extended I/O System is unable to attach the device containing the file because the Basic I/O System has already attached the device.
E$CONTEXT	0005H	The calling task's job is not an I/O job.
EDEVDETACHING	0039H	The device containing the specified file is in the process of being detached.
E$DEVFD	0022H	The Extended I/O System attempted the physical attachment of a device that had formerly been only logically attached. In the process, it found that the device and the device driver specified in the logical attachment were incompatible.

E$FNEXIST	0021H	At least one of the following is true: • A file in the specified path, or the target file itself, does not exist or is marked for deletion. • The physical device specified in the call was not found.
E$FTYPE	0027H	The specified path contains a file name that should be the name of a directory, but is not. (Except for the last file, each file in a path must be a directory.)
E$ILLVOL	002DH	The Extended I/O System attempted the physical attachment of a device that had formerly been only logically attached. In the process, it examined the volume label and found that the volume does not contain named files. This prevented the Extended I/O System from completing physical attachment because the named file driver was requested during logical attachment.
E$INVALID$FNODE	003DH	The fnode for the specified file is invalid. The file cannot be accessed; you should delete it.
EIOHARD	0052H	A hard I/O error occurred. A retry is probably useless.
EIOMODE	0056H	One of the following is true: • A tape drive attempted to perform a read operation before the previous write operation completed. • A tape drive attempted to perform a write operation before the previous read operation completed.
EIONO$DATA	0055H	A tape drive attempted to read the next record, but it found no data.
EIOOPRINT	0053H	The device was off-line. Operator intervention is required.
EIOSOFT	0051H	A soft I/O error occurred. The I/O System tried to perform the operation a number of times and failed (the number of retries is a configuration parameter). Another retry might still be successful.
EIOUNCLASS	0050H	An unknown type of I/O error occurred.
EIOMEM	0042H	The Basic I/O System job does not currently have a block of memory large enough to allow this system call to run to completion.

E$LIMIT	0004H	At least one of the following is true: • The user object or the calling task's job is already involved in 255 (decimal) I/O operations. • The calling task's job is not an I/O job. • The calling task's object limit has been reached.
ELOGNAME$-NEXIST	0045H	The specified path contains an explicit logical name, but the call was unable to find this name in the object directories of the calling task's local job, the global job, or the root job.
ELOGNAME$-SYNTAX	0040H	The specified logical name contains at least one of the following syntax errors: • The specified path starts with a colon (:), indicating that it contains a logical name. But the call was unable to find a second colon to delimit the logical name. • contains a logical name that is either longer than 12 characters (including colons), has no characters, or contains invalid characters.
E$MEDIA	0044H	The device containing the specified file is not on-line.
E$MEM	0002H	The memory available to the calling task's job is not sufficient to complete the call.
ENOPREFIX	8022H	You did not specify an explicit prefix (logical name), and the default prefix for the calling task's job is either undefined, or it is not a valid device connection or file connection.
ENOTCON-FIGURED	0008H	This system call is not part of the present configuration.
ENOTLOG$NAME	8040H	The specified path contains a logical name that refers to an object that is neither a device connection nor a file connection.
ENOUSER	8021H	The calling task's job does not have a default user, or its default user is not a user object.

E$PARAM	8004H	The Extended I/O System attempted the physical attachment of a device that had formerly been only logically attached. In the process, it found that the logical attachment referred to a file driver (named, physical, or stream) that is not configured into your system. Therefore the physical attachment is not possible.
E$PATHNAME$-SYNTAX	003EH	The specified pathname contains invalid characters.

SGETPATH$COMPONENT returns the name of a named file as the file is known in its parent directory.

```
CALL RQ$S$GET$PATH$COMPONENT(connection, name$ptr, except$ptr);
```

Input Parameters

connection	A TOKEN for the file connection whose name is desired.

Output Parameter

name$ptr	A POINTER to a STRING where the system returns the path component. The maximum length of the STRING is 14 BYTES.
except$ptr	A POINTER to a WORD where the Extended I/O System returns condition codes.

Description

The format of the component returned by this call is dependent on the type of file driver employed by the call. A null string is returned under the following circumstances:

- If the file driver is Named or Remote and the connection is to the root directory of a volume.
- If the file driver's connection accesses either Stream or Physical files.

Condition Codes

E$OK	0000H	No exceptional conditions.
E$CONTEXT	0005H	The name$ptr parameter is equal to NIL.
E$FNEXIST	0021H	The file is marked for deletion. (In this case, the string is undefined.)
E$INVALID$FNODE	003DH	The fnode for the specified file is invalid. The file cannot be accessed; you should delete it.
E$IO	002BH	An I/O error occurred that might have prevented the operation from completing.
EIOMEM	0042H	The memory available to the EIOS is not sufficient to complete the call.

The S$LOOK$UP$CONNECTION system call accepts a logical name from the calling task and returns a token for the connection associated with the logical name.

```
connection = RQ$S$LOOK$UP$CONNECTION(log$name$ptr, except$ptr);
```

Input Parameter

log$name$ptr	A POINTER to a STRING (of 1 to 12 characters) containing the logical name to be looked up. The name can be delimited with colons (:). The operating system removes the colons so that a logical name with colons is the same as one without (e.g., :F0: is effectively the same as F0). Colons do not count in the length of the name.

Output Parameters

connection	The TOKEN that represents the connection associated with the logical name.
except$ptr	A POINTER to a WORD where the Extended I/O System returns the condition code.

Description

After converting any lowercase letters in the logical name to uppercase, the Extended I/O System searches for the logical name. It first checks the object directory of the local job, then the global job, and finally the root job. (This progressively more global search sequence is described more completely in Chapter 3 of the *Extended iRMX II Extended I/O System User's Guide*.) When it finds the logical name, the Extended I/O System returns the token for the connection.

Your tasks can invoke this system call to look up logical names created by the Nucleus system call CATALOG$OBJECT. However, CATALOG$OBJECT does not convert from lowercase to uppercase. So if you desire compatibility, use uppercase characters when you use the CATALOG$OBJECT system call.

Condition Codes

E$OK	0000H	No exceptional conditions.
E$CONTEXT	0005H	The calling task's job is not an I/O job.
E$LIMIT	0004H	The calling task's job is not an I/O job.

ELOGNAME$-NEXIST	0045H	The specified path contains an explicit logical name, but the call was unable to find this name in the object directories of the calling task's local job, the global job, or the root job.
ELOGNAME$-SYNTAX	0040H	The specified logical name contains at least one of the following syntax errors: • The specified path starts with a colon (:), indicating that it contains a logical name. But the call was unable to find a second colon to delimit the logical name. • The specified path contains a logical name that is either longer than 12 characters, has no characters, or contains invalid characters.
E$MEM	0002H	The memory available to the calling task's job is not sufficient to complete the call.
ENOTCON-FIGURED	0008H	This system call is not part of the present configuration.
ENOTCONNECTION	8042H	The logical name refers to an object that is not a connection.
E$TIME	0001H	The calling task's job is not an I/O job.

The S$OPEN system call opens a file connection so that your tasks can access the file.

```
CALL RQ$S$OPEN(connection, mode, number$buffers, except$ptr);
```

Input Parameters

connection — A TOKEN for the file connection to be opened. The connection must have been created in the calling task's job. If the connection was created in a different job, use S$ATTACH$FILE to obtain a new connection.

mode — A BYTE telling how your task is going to use the connection and with whom it will share the connection. You should set the BYTE as follows:

Value	How Connection is Used
1H	For reading only; share with all.
2H	For writing only; share with all.
3H	For both reading and writing; share with all.
4H	For reading only; private use.
5H	For writing only; private use.
6H	For both reading and writing; private use.
7H	For reading only; share with readers.
8H	For writing only; share with readers.
9H	For both reading and writing; share with readers.
0AH	For reading only; share with writers.
0BH	For writing only; share with writers.
0CH	For both reading and writing; share with writers.

number$buffers — A BYTE containing the number of buffers that you want the Extended I/O System to allocate for this connection. This number must be between zero and a maximum value that you specified when you configured the Basic I/O System.

Output Parameter

except$ptr — A POINTER to a WORD where the Extended I/O System returns the condition code.

Description

This system call performs the following functions:

- It creates the number of buffers requested.
- It sets the connection's file pointer to NIL.
- It starts reading ahead if the number of buffers is greater than zero and the mode parameter includes reading.

Access Rights and Selecting a Mode

When you specify the mode, you must be accurate or err on the side of generosity. If you are not certain how the connection will be used, specify both reading and writing.

In the case of named files, the mode that you specify must match the access rights of the connection. (These are the access rights that the Extended I/O System assigned the connection when the connection was created.) For example, if your task attempts to open for reading a connection that has access for writing only, the Extended I/O System returns an E$FACCESS exception code.

Selecting the Number of Buffers

Deciding how many buffers to allocate for file I/O is based on two considerations--memory and performance. The amount of memory used for buffers is directly proportional to the number of buffers. So you can save memory by using fewer buffers.

The performance consideration is more complex. Up to a certain point, the more buffers you allocate, the faster your task can run. The actual break-even point, the point where more buffers don't improve performance, depends on many variables. Be aware that in order to overlap I/O with computation, you must specify at least two buffers.

If performance is important, and you have no idea how many buffers to specify, start with two. Once your task is running successfully, you can experiment, adding or removing buffers until you have found the optimum number of buffers.

If performance is not so important and memory is, use zero buffers.

Condition Codes

E$OK	0000H	No exceptional conditions.
E$CONN$OPEN	0035H	The connection is already open.
EDEVOFF$LINE	002EH	The device being accessed is now offline.
E$EXIST	0006H	The connection parameter is not a token for an existing object.
E$FACCESS	0026H	The access rights embedded in the connection prohibit opening the file in the specified mode.
EIOHARD	0052H	A hard I/O error occurred. A retry is probably useless.
EIOMODE	0056H	One of the following is true: • A tape drive attempted to perform a read operation before the previous write operation completed. • A tape drive attempted to perform a write operation before the previous read operation completed.
EIONO$DATA	0055H	A tape drive attempted to read the next record, but it found no data.
EIOOPRINT	0053H	The device was off-line. Operator intervention is required.
EIOSOFT	0051H	A soft I/O error occurred. The I/O System tried to perform the operation a number of times and failed (the number of retries is a configuration parameter). Another retry might still be successful.
EIOUNCLASS	0050H	An unknown type of I/O error occurred.
EIOWRPROT	0054H	The volume is write-protected.
E$LIMIT	0004H	At least one of the following is true: • The calling task's job is not an I/O job. • The calling task's job, or the job's default user object, is already involved in 255 (decimal) I/O operations.
E$MEM	0002H	The memory available to the calling task's job is not sufficient to complete the call.
ENOTCON-FIGURED	0008H	This system call is not part of the present configuration.

ENOTCONNECTION	8042H	The connection parameter is a token for an object that is not a file connection.
ENOTFILE$CONN	0032H	The connection is a device connection.
E$PARAM	8004H	The mode parameter is set to a value other than 1 through C hexadecimal.
E$SHARE	0028H	At least one of the following is true: • The call attempted to open a directory file or a bit-map file for writing. • The file's sharing attribute is currently not compatible with the mode specified in this call.
E$SUPPORT	0023H	The specified connection was not created by a task in the calling task's job.

The S$READ$MOVE reads a number of bytes from a file to a buffer.

```
bytes$read = RQ$S$READ$MOVE(connection, buffer$ptr, bytes$desired,
                            except$ptr);
```

Input Parameters

connection	A TOKEN for the connection to the file. This connection must be open for reading or for both reading and writing, and the file pointer of the connection must point to the first byte to be read.
bytes$desired	A WORD containing the maximum number of bytes you want to read from the file.

Output Parameters

bytes$read	A WORD containing the actual number of bytes that the Extended I/O System reads from the file.
buffer$ptr	A POINTER to a buffer that will receive the information that the Extended I/O System reads from the file.
except$ptr	A POINTER to a WORD where the Extended I/O System returns a condition code.

Description

This system call reads a collection of contiguous bytes from the file associated with the connection. These bytes are placed in a buffer specified by the calling task.

Creating the Buffer

The buffer$ptr parameter tells the Extended I/O System where to place the bytes after they are read. You must create this buffer because the Extended I/O System does not. To create the buffer, make an extended iRMX II segment, or create a buffer during the compilation of your program. You must ensure that the buffer is long enough.

If you use an extended iRMX II segment as your buffer, the 80286 microprocessor's built-in abilities will detect when a task attempts to write beyond a buffer. If you create a buffer at compilation time, the Extended I/O System will not sense when overwriting occurs. If your task attempts to read more bytes than the buffer is capable of holding, the information immediately following the buffer could be overwritten.

Number of Bytes Read

The number of bytes that your task requests (bytes$desired) is the maximum number of bytes that the Extended I/O System places in the buffer. However, there are two circumstances under which the Extended I/O System reads fewer bytes.

- First, if the Extended I/O System detects an end-of-file before reading the number of bytes requested, it returns only those bytes preceding the end-of-file. In this case, the bytes$read parameter can be less than the bytes$desired parameter without generating an exceptional condition.
- Second, if an exceptional condition occurs during the reading operation. In this case, the information in the buffer and the value of the bytes$read parameter are meaningless.

If your task performs random-access reads of the file, it must identify which bytes to read from the file by using the S$SEEK system call to position the connection's file pointer to the first byte that it wants to read.

In contrast, if your task reads from the file sequentially, the Extended I/O System maintains the connection's file pointer automatically.

Effects of Priority

The priority of the task invoking this system call can greatly affect the performance of the application system. For better performance, the priority of the invoking task should be equal to or lower than (numerically greater than) the priority of the task that attached the device with the Basic I/O System call A$PHYSICAL$ATTACH$DEVICE. If the device was attached with LOGICAL$ATTACH$DEVICE, the task that issues A$PHYSICAL$ATTACH$DEVICE is an Extended I/O System task created when the system is initialized. The priority of this task is set to 130 decimal. If the priority of the calling task is higher than the task that attached the device, the operating system cannot overlap the read operation with computation or with other I/O operations. (To find out how to set priorities for application tasks, refer to the *Extended iRMX II Nucleus User's Guide*.)

Condition Codes

E$OK	0000H	No exceptional conditions.
EBADBUFF	8023H	One of the following is true: • The specified memory buffer is not writeable. • The specified memory buffer crosses a segment boundary.

E$CONN$NOT$OPEN	0034H	At least one of the following is true: • The connection is not open for reading or for both reading and writing. • The connection is closed. • The connection was opened by the A$OPEN system call rather than the S$OPEN system call.
E$EXIST	0006H	The connection is not a token for an existing object.
E$FLUSHING	002CH	The specified device is being detached.
E$IDDR	002AH	This request is invalid for the device driver. For example, it is not valid to use this call with a line printer.
EIOHARD	0052H	A hard I/O error occurred. A retry is probably useless.
EIOMODE	0056H	One of the following is true: • A tape drive attempted to perform a read operation before the previous write operation completed. • A tape drive attempted to perform a write operation before the previous read operation completed.
EIONO$DATA	0055H	A tape drive attempted to read the next record, but it found no data.
EIOOPRINT	0053H	The device was off-line. Operator intervention is required.
EIOSOFT	0051H	A soft I/O error occurred. The I/O System tried to perform the operation a number of times and failed (the number of retries is a configuration parameter). Another retry might still be successful.
EIOUNCLASS	0050H	An unknown type of I/O error occurred.
E$LIMIT	0004H	At least one of the following is true: • The calling task's job, or the job's default user object, is already involved in 255 (decimal) I/O operations. • The calling task's job is not an I/O job.
E$MEM	0002H	The memory available to the calling task's job is not sufficient to complete the call.

ENOTCON-FIGURED	0008H	This system call is not part of the present configuration.
ENOTCONNECTION	8042H	The connection parameter is a token for an object that is not a file connection.
E$SPACE	0029H	At least one of the following is true: • This call attempted to read beyond the end of the volume. • Another task is writing to the file using the same connection and is attempting to write beyond the end of the volume or the end of the available space on the volume.
E$SUPPORT	0023H	The connection parameter was not created by a task in the calling task's job.

The S$RENAME$FILE system call changes the name of a directory or data file. It cannot be used for stream or physical files.

```
CALL RQ$S$RENAME$FILE(path$ptr, new$path$ptr, except$ptr);
```

Input Parameters

path$ptr	A POINTER to a STRING that specifies the current path for an existing file that is to be renamed. The syntax of this path is described in Chapter 4 of the *Extended iRMX II Estended I/O System User's Guide*.
new$path$ptr	A POINTER to a STRING that specifies the new path for the file. This path must comply with the syntax and semantics of paths for named files as discussed in Chapter 4 of the *Extended iRMX II Extended I/O System User's Guide*. Furthermore, this path cannot refer to an existing file.

Output Parameter

except$ptr	A POINTER to a WORD where the Extended I/O System returns a condition code.

Description

This system call, which can be used only with named files, allows your task to change the path for a file. You can rename directory files as well as data files.

NOTE

When you rename a directory, you change the paths for all files and other directories contained in the directory.

Restrictions

If your task is renaming a file, the task can change any aspect of the file's path so long as the file remains on the same volume. If you are renaming a directory, it must still have the same parent directory (the directory above the one being renamed).

To be able to rename a file, the default user object of the calling task's job must have two kinds of access:

- Deletion access to the original file
- Add-entry access to the file's new parent directory

Condition Codes

E$OK	0000H	No exceptional conditions.
E$ALREADY$-ATTACHED	0038H	The Extended I/O System is unable to attach the device containing the file because the Basic I/O System has already attached the device.
E$CONTEXT	0005H	The calling task's job is not an I/O job.
EDEVDETACHING	0039H	The device containing the specified file is in the process of being detached.
E$DEVFD	0022H	The Extended I/O System attempted to physically attach a device that had been only logically attached, and found that the device and the device driver specified in the logical attachment were incompatible.
E$FACCESS	0026H	At least one of the following is true: • The call is trying to rename a bit-map file or the root directory. • The default user object associated with the calling task's job does not have add-entry access to the parent directory of the new$path$ptr file. • The default user object associated with the calling task's job does not have delete access to the file to be renamed.
E$FEXIST	0020H	The new$path$ptr parameter refers to a file that already exists.
E$FNEXIST	0021H	A file in the specified path, or the file being renamed, does not exist or is marked for deletion.
E$FTYPE	0027H	The specified path contains a file name that should be the name of a directory, but is not. (Except for the last file, each file in a path must be a directory.)
E$IFDR	002FH	The specified file is a stream or physical file.
E$ILLOGICAL$-RENAME	003BH	The call attempted to rename a directory to a new path containing itself.

E$ILLVOL	002DH	The Extended I/O System attempted to physically attach a device that had formerly been only logically attached. In the process, it found that the volume does not contain named files. This prevented the Extended I/O System from completing physical attachment because the named file driver was requested during logical attachment.
E$INVALID$FNODE	003DH	The fnode for the specified file is invalid. The file cannot be accessed; you should delete it.
EIOHARD	0052H	A hard I/O error occurred. A retry is probably useless.
EIOOPRINT	0053H	The device was off-line. Operator intervention is required.
EIOSOFT	0051H	A soft I/O error occurred. The I/O System tried to perform the operation a number of times and failed (the number of retries is a configuration parameter). Another retry might be successful.
EIOUNCLASS	0050H	An unknown type of I/O error occurred.
EIOWRPROT	0054H	The volume is write-protected.
EIOMEM	0042H	The Basic I/O System job does not currently have a block of memory large enough to allow this system call to run to completion.
E$LIMIT	0004H	At least one of the following is true: • The user object or the calling task's job is already involved in 255 (decimal) I/O operations. • The calling task's job is not an I/O job. • The calling task's object limit has been reached.
ELOGNAME$-NEXIST	0045H	At least one of the specified paths contains an explicit logical name, but the call was unable to find this name in the object directories of the calling task's local job, the global job, or the root job.

ELOGNAME$-SYNTAX	0040H	At least one of the specified paths contain one or more of the following logical name syntax errors: • A path starts with a colon (:), indicating that it contains a logical name. But the call was unable to find a second colon to delimit the logical name. • A path contains a logical name that is either longer than 12 characters (including colons), has no characters, or contains invalid characters.
E$MEDIA	0044H	The device containing the specified file is not on-line.
E$MEM	0002H	The memory available to the calling task's job is not sufficient to complete the call.
ENOPREFIX	8022H	At least one of the specified paths contains no explicit prefix (no logical name), and the default prefix for the calling task's job is either undefined, or it is not a valid device connection or file connection.
ENOTCONFIGURED	0008H	This system call is not part of the present configuration.
ENOTLOG$NAME	8040H	At least one of the specified paths contains a logical name that refers to an object that is neither a device connection nor a file connection.
ENOTSAME$DEV	003AH	The two paths refer to different devices.
ENOUSER	8021H	The calling task's job does not have a default user object, or the object cataloged in R?IOUSER is not a user object.
E$PATHNAME$-SYNTAX	003EH	One or both of the specified pathnames contain invalid characters.
E$PARAM	8004H	The specified task$priority for an IO job is unequal to 0 and is greater than the max$priority of the IO job.
E$SPACE	0029H	The volume is full.
E$SUPPORT	0023H	The task attempted to rename a physical or stream file.

Using the S$SEEK system call, your tasks can move the file pointer for any open physical- or named-file connection. This system call cannot be used with stream files.

```
CALL RQ$S$SEEK(connection, mode, move$count, except$ptr);
```

Input Parameters

connection — A TOKEN for an open connection whose file pointer you wish to move.

mode — A BYTE containing a value that controls the nature of the movement of the file pointer. Any of the following values are valid:

Mode	Meaning
1	Move the pointer backward by the number of bytes specified in move$count. If the move count is large enough to position the pointer past the beginning of the file, the pointer moves to the first byte ???NIL?(position zero).
2	Set the pointer to the position specified by the move count. Position ?NIL?zero is the first position in the file. Moving the pointer beyond the end of the file is valid for named files only.
3	Move the file pointer forward by the specified amount. Moving the pointer beyond the end-of-file is valid for named files.
4	First move the pointer to the end of the file and then move it backward by the specified amount. If the value specified by move$count would position the pointer beyond the front of the file, the pointer moves to the first byte in the file ?NIL?(position zero).

move$count — A DWORD integer that tells the Extended I/O System how far, in bytes, to move the pointer.

Output Parameter

except$ptr	A POINTER to the WORD where the Extended I/O System returns the condition code.

Description

When performing random I/O, your tasks must use this system call to position the file pointer before using the S$READ$MOVE, S$TRUNCATE$FILE, and S$WRITE$MOVE system calls. The location of the file pointer tells the Extended I/O System where in the file to begin reading, truncating, or writing information.

If your tasks are performing sequential I/O on a file, they do not need to use this system call.

Access Control

Two requirements relate to access control. First, the connection must be open for reading only, writing only, or both reading and writing. If this is not the case, your task can use the S$OPEN system call to open the file.

The second access requirement is that the connection must have been created by a task within the calling task's job. If this is not the case, use the existing connection as a prefix, and have the calling task obtain a new connection by invoking the S$ATTACH$FILE system call. This newly created connection satisfies the second requirement.

Reading and Writing Beyond the End of File

It is legitimate to position the file pointer beyond the end-of-file for a named file. If your task does this and then invokes the S$READ$MOVE system call, the Extended I/O System behaves as though the reading operation began at the end-of-file.

Also, it is possible to invoke the S$WRITE$MOVE system call with the file pointer beyond the end of the file. If your task does this, the Extended I/O System attempts to expand the file. If the Extended I/O System does expand your file in this manner, the file contains random information between the old end-of-file and the new end-of-file.

Condition Codes

E$OK	0000H	No exceptional conditions.
EBADBUFF	8023H	One of the following is true: • The specified memory buffer is not writeable. • The specified memory buffer crosses a segment boundary.

E$CONN$NOT$OPEN	0034H	At least one of the following is true: • The connection is not open. • The connection was opened by an A$OPEN rather than an S$OPEN.
E$EXIST	0006H	The connection parameter is not a token for an existing object.
E$FLUSHING	002CH	The specified device is being detached.
E$IDDR	002AH	This request is invalid for the device driver. For example, it is not valid to use this call with a line printer.
E$IFDR	002FH	The call attempted to seek in a stream file. The S$SEEK system call can be used only with named and physical files.
EIOHARD	0052H	A hard I/O error occurred. A retry is probably useless.
EIOMODE	0056H	A tape drive attempted a read (write) operation before the previous write (read) completed.
EIONO$DATA	0055H	A tape drive attempted to read the next record, but it found no data.
EIOOPRINT	0053H	The device was off-line. Operator intervention is required.
EIOSOFT	0051H	A soft I/O error occurred. The I/O System tried to perform the operation a number of times and failed (the number of retries is a configuration parameter). Another retry might still be successful.
EIOUNCLASS	0050H	An unknown type of I/O error occurred.
E$LIMIT	0004H	At least one of the following is true: • Either the calling task's job, or the job's default user object, is already involved in 255 (decimal) I/O operations. • The calling task's job is not an I/O job.
E$MEM	0002H	The memory available to the calling task's job is not sufficient to complete the call.
ENOTCONFIGURED	0008H	This system call is not part of the present configuration.
ENOTCONNECTION	8042H	The connection parameter is a token for an object that is not a file connection.

E$PARAM	8004H	At least one of the following is true: • The value of the mode parameter is not 1, 2, 3, or 4. • The calling task was attempting to seek past the end of a physical file.
E$SPACE	0029H	This seek operation forced the Extended I/O System to attempt to empty the connection's buffer(s) by writing their contents to the volume. However, the volume is full.
E$SUPPORT	0023H	The connection parameter refers to a connection that was created by a task outside of the calling task's job.

The S$SPECIAL system call allows your tasks to perform functions that are peculiar to a specific device.

```
CALL RQ$S$SPECIAL(connection, function, data$ptr, iors$ptr,
                  except$ptr);
```

Input Parameters

connection — A TOKEN for a connection to the file for which the special function is to be performed.

function — A WORD that specifies the special function being requested. Each function is described in detail under the "Description" heading, but the following table summarizes the values to be assigned to this parameter.

Function Value	Type of file for connection	Effect of Function
0	Physical	Format disk track
0	Stream	Query
1	Stream	Satisfy
2	Physical or Named	Notify
3	Physical	Get disk special data
4	Physical	Get terminal data
5	Physical	Set terminal data
6	Physical	Set signal character
7	Physical	Rewind tape
8	Physical	Read tape file mark
9	Physical	Write tape file mark
10	Physical	Retention tape
11-32767		Reserved for other Intel products

data$ptr — A POINTER to a parameter block that your task uses to supply the Extended I/O System with information, or to receive information from the Extended I/O System. The contents and form of the parameter block depend upon the function being requested, so the form of the parameter block is described later, under the "Description" heading. If the function requires no parameter block, set data$ptr to NIL.

Output Parameters

iors$ptr

A POINTER to a structure of the form described below. The Extended I/O System uses this structure to return information that might be of use to the calling task. If you set this POINTER to NIL, the Extended I/O System does not return the information. Be aware that this is relatively obscure information that most applications do not need.

```
DECLARE iors$data STRUCTURE(
                    actual          WORD,
                    actual$fill     WORD,
                    device          WORD,
                    unit            BYTE,
                    funct           BYTE,
                    subfunct        WORD,
                    device$loc      DWORD,
                    buf$ptr         POINTER,
                    count           WORD,
                    count$fill      WORD,
                    aux$ptr         POINTER)
```

where

actual — Number of bytes that were actually transferred during the special function, if any.

actual$fill — Reserved for Intel's use.

device — Device number identifying the device. For an explanation of device numbers, refer to the *Extended iRMX II Interactive Configuration Utility Reference Manual.*

unit — Number of the unit that contains the file on which the special function is being performed.

funct — Code recognized by the driver, usually meaning that this is a special operation.

subfunct — Function code you code into the call.

device$loc — Location on the device where the operation was performed.

buf$ptr — POINTER to a buffer used for this operation, if any buffer is used.

count — Number of bytes transferred, if any were transferred.

count$fill — Reserved for future use.

aux$ptr — Same as data$ptr in the call to S$SPECIAL.

except$ptr — A POINTER to a WORD where the Extended I/O System returns the condition code.

Description

This system call allows your tasks to communicate with devices, device drivers, and the stream file driver to perform operations that are less device-independent than other Extended I/O System operations.

S$SPECIAL allows your task to perform several special functions. The Extended I/O System decides which function to perform by examining the function parameter and the kind of connection provided in the connection parameter. The following sections explain each function in detail.

Formatting a Track (Function Code 0)

To use the S$SPECIAL system call to format a track on a disk, the calling task must supply the following information:

connection — A TOKEN for a connection to a physical file. This connection must be open for reading, writing or both.

function — Must be set to zero.

data$ptr — Must point to a STRUCTURE of the following form:

```
DECLARE track$formatter STRUCTURE(
                    track$number        WORD,
                    interleave          WORD,
                    track$offset        WORD,
                    fill$char           BYTE)
```

where

track$number — Number of the track to be formatted. Acceptable values are 0 to one less than the number of tracks on the volume. Other values cause an E$SPACE exception code. When formatting a tape or a RAM-disk, you must place a zero value in this field.

interleave	The number of physical sectors between consecutive logical sectors. (This field does not apply to tapes or to RAM-disks.) If the interleave factor is zero or one, no physical sectors are skipped. If the specified interleave value is greater than the number of physical sectors on a track, the operating system divides that interleave value by the number of physical sectors and uses the remainder as the interleave factor. A remainder of zero has the same effect as an interleave of zero.
track$offset	Number of physical sectors to skip between the index mark and the first logical sector. (This field does not apply to tapes or to RAM-disks.)
fill$char	The character with which the sector will be written; some drivers ignore this field and fill the sectors with a character the driver establishes.

Also see the description of Function Code 3, Getting Special Disk Data.

Obtaining Information About Stream File Operations (Function Code 0)

Occasionally, a task using a stream file must find out what is being requested by another task using the same stream file. For example, the task reading a stream file might need to know how many bytes are being sent by a task writing to the same file. Tasks can obtain this kind of information by calling S$SPECIAL with the following information:

connection	A TOKEN for a connection to a stream file.
function	Zero.
data$ptr	Set to NIL.

If a task is reading from or writing to a stream file, the Extended I/O System returns information in the structure to which iors$ptr points. The following four fields contain valid information:

actual	The number of bytes already transferred.
count	The number of bytes remaining to be transferred.
buf$ptr	A POINTER to the memory location to be used for the next byte to be transferred.
funct	A value that indicates the purpose of the queued request. The value is zero for read requests and one for write requests.

If no task is reading from or writing to the stream file, the Extended I/O System queues the S$SPECIAL request. The request remains queued until a task issues a read or write request. If, before a read or write request is issued, another S$SPECIAL request arrives, the Extended I/O System cancels both S$SPECIAL requests and returns E$STREAM$SPECIAL exception codes to the tasks that issued the S$SPECIAL calls.

Satisfying Stream File Transactions (Function Code 1)

Stream files provide two tasks with the ability to communicate. When one task tries to read or write to a stream file, the task does not run again until the complementary task issues a matching request.

For example, suppose that Task A wants to read 512 bytes, but Task B writes only 256 bytes. Task A stops running until Task B issues one or more requests which supply at least 256 more bytes.

The S$SPECIAL system call enables tasks to force a stream file transaction to complete, even if the number of bytes written does not match the number of bytes read.

To force this completion, a task must invoke the S$SPECIAL system call with the parameters set as follows:

connection	A TOKEN for a connection to the stream file. This connection must be open for the operation that has not satisfied the matching requirement. For example, if the reading task wants to force the Extended I/O System to consider the transaction completed, the connection must be open for reading.
function	One.
data$ptr	Set to NIL.

After requesting this satisfy function, the only information that your task can obtain is the condition code returned by the Extended I/O System. If the task invoking the S$SPECIAL system call has already completed the transaction, the Extended I/O System returns an E$STREAM$SPECIAL condition code.

Requesting Notification that a Volume is Unavailable (Function Code 2)

This function applies to named and physical files only. When a person opens a door to a flexible disk drive or presses the STOP button on other mass storage drives, the volume mounted on that drive becomes unavailable. A task can request notification of such an event by calling S$SPECIAL. For flexible disk drives attached to an iSBC 208 or iSBC 218A controller, and for some 5-1/4" flexible disk drives, notification occurs when the Basic I/O System first tries to perform an operation on the unavailable volume. For most other drives, notification occurs immediately. The reason for this difference is that controller/drive combinations that include the iSBC 208 or iSBC 218A controller, or that include some 5-1/4" drives, cannot generate an interrupt when the drive ceases to be ready. In contrast, most other controller/drive combinations do.

On those drives where no notification occurs until the Basic I/O System attempts to access the drive, a dangerous situation occurs whenever you change a volume without first detaching the device. If you do not first detach the device and then reattach it, the Basic I/O System accesses the device using the directory information from the old volume. Unless the new volume is write-protected, this process corrupts the entire volume, rendering it useless. The correct sequence of events when changing volumes on one of these devices is as follows:

1. Detach the unit (via A$PHYSICAL$DETACH$DEVICE).
2. Remove the old volume.
3. Install the new volume.
4. Reattach the unit (via A$PHYSICAL$ATTACH$DEVICE).

For devices that can perform notification, a task requests notification by calling S$SPECIAL with a token for a device connection, with spec$func set to 2, and with data$ptr pointing to a structure of the following form:

```
DECLARE notify      STRUCTURE(
                        mailbox     TOKEN,
                        object      TOKEN);
```

where

mailbox — A TOKEN for a mailbox.

object — A TOKEN for an object. When the Basic I/O System detects that the implied volume is unavailable, the object is sent to the mailbox.

After a task has made a request for notification, the Basic I/O System remembers the object and mailbox tokens until either the volume is detected as being unavailable or until the device is detached by the A$PHYSICAL$DETACH$DEVICE system call. When the volume becomes unavailable, the object is sent to the mailbox. Note that this implies that some task should be dedicated to waiting at the mailbox.

If the volume is detected as being unavailable, the Basic I/O System will not execute I/O requests to the device on which the volume was mounted. Such requests are returned with the status field of the I/O result segment set to E$IO and the unit$status field set to IO$OPRINT (value = 3). The latter code means that operator intervention is required.

If any task issues a subsequent notification request for the same device connection, the Basic I/O System replaces the old mailbox and object values with the new ones specified. It does not return an exception code.

To restore the availability of a volume, perform the following steps:

1. Close the door of the diskette drive or restart the hard disk drive.
2. Call A$PHYSICAL$DETACH$DEVICE. It may be necessary to do a "hard" detach of the device.
3. Call A$PHYSICAL$ATTACH$DEVICE and reattach the device.
4. Create a new file connection.

To cancel a request for notification, make a dummy request using the same connection with a SELECTOR$OF(NIL) value in the mailbox parameter.

Getting Disk Special Data (Function Code 3)

You can write your own program to format a disk, rather than using the FORMAT command (part of the extended iRMX II Human Interface). If you do so, you must place some special device data into the last bytes of the label on the extended iRMX II named volume. Currently, this field in the label is eight (8) bytes long, although Intel reserves the right to add to its length later. (The structure of an extended iRMX II named file volume is described in the *Extended iRMX II Disk Verification Utility Reference Manual.*)

You can obtain the data in this field by issuing S$SPECIAL with a function code of three. You can then save the data and write it into the label field when you format the disk.

To use the S$SPECIAL system call to obtain the special data for the label, the calling task must supply the following information:

connection	A TOKEN for a connection to a physical file. This connection must be open for reading, for writing, or for both reading and writing.
function	Three.

data$ptr — A POINTER to a STRUCTURE of the following form:

```
DECLARE disk$label$data     STRUCTURE(
        label$data(8)               WORD);
```

Getting Terminal Characteristics (Function Code 4)
Setting Terminal Characteristics (Function Code 5)

These two functions are complements of each other. They use the same type of data structure with identical meanings for each field in the structure. A function code of four returns the current characteristics of a particular terminal; a function code of five allows you to set the characteristics of a terminal.

Intel recommends that before setting the terminal characteristics, you first invoke S$SPECIAL with function code 4 to get the current characteristics. Then, modify the returned structure to reflect your desired changes. Finally, invoke S$SPECIAL with function code 5 to set the characteristics, using your modified structure as input.

In this section, certain terms unique to terminal devices (for example, line editing, OSC sequence, translation) are described only briefly. If you are unfamiliar with these terms refer to the *Extended iRMX II Basic I/O System Calls Reference Manual* and the *Extended iRMX II Device Driver User's Guide*.

To use the S$SPECIAL system call to get or to set terminal characteristics, the calling task must supply the following information:

connection — A TOKEN for a connection to a terminal.

function — Four (get characteristics) or five (set characteristics).

data$ptr — A POINTER to a STRUCTURE of the following form:

```
DECLARE terminal$attributes STRUCTURE(
                    num$words           WORD,
                    num$used            WORD,
                    connection$flags    WORD,
                    terminal$flags      WORD,
                    in$baud$rate        WORD,
                    out$baud$rate       WORD,
                    scroll$lines        WORD,
                    x$y$size            WORD,
                    x$y$offset          WORD,
                    special$modes       WORD,
                    high$water$mark     WORD,
                    low$water$mark      WORD,
                    fc$on$char          WORD,
                    fc$off$char         WORD,
                    link$parameter      WORD,
                    spc$hi$water$mark   WORD,
                    special$char(4)     BYTE);
```

where

num$words — The number of words, not including num$words and num$used, that are reserved for the remainder of the terminal$attributes data structure. To access all of the information, set this field to at least 16. Intel reserves the right to expand the length of this structure in later releases.

num$used — The number of fields, following the num$used field, that are actually being used for getting or setting terminal characteristics.

In getting and setting terminal information, the amount of data returned or sent is governed by the num$used field. For example, if function is 4 and num$used is 2, then an S$SPECIAL call returns data in the connection$flags and terminal$flags fields, but not in the remainder of the fields.

However, when setting terminal attributes, specifying a zero value for any of the next five fields (connection$flags through scroll$lines) causes the I/O System to skip over the zeroed field, leaving it at its previous setting. For example, if num$used is 2, while connection$flags is 0 and terminal$flags is not 0, then S$SPECIAL uses the contents of the terminal$flags field to set terminal attributes, but it ignores the contents of connection$flags field. In this way, you can set some parameters without affecting others.

For the functions represented by the remaining fields in this structure, invoking S$SPECIAL is not the only way to set the functions. You can also set them with OSC sequences. The description of each field mentions, in parentheses, the OSC characters you can use. (OSC sequences are described in the *Extended iRMX II Device Drivers User's Guide*.) You can also use the OSC Query sequence when debugging, to ensure that your tasks invoked S$SPECIAL correctly.

connection$flags

This word applies only to this connection to the terminal. (All other parameters apply to the terminal itself and therefore to all connections to the terminal.) If you attempt to set this field to zero, the I/O System ignores your entry and leaves the field set to its previous value.

The flags in this word are encoded as follows. (Bit 0 is the low-order bit.)

Bits	Value and Meaning
0-1	Line editing control (corresponds to OSC characters C:T). Line editing refers to how the TSC (Terminal Support Code) handles control characters such as those that delete characters entered at a terminal, scroll terminal output, and others. Refer to the *Extended iRMX II Device Drivers User's Guide* for more information.

NOTE

Line editing is supported on input only (that is, the stream of data entered at, but not sent to, a terminal).

Bits	Value and Meaning
	0 = Invalid Entry.
	1 = No line editing (transparent mode). Input is transmitted to the requesting task exactly as entered at the terminal*. Before being transmitted, data accumulates in a buffer until the requested number of characters has been entered.
	2 = Line editing (normal mode). Edited data accumulates in a buffer until a line terminator is entered.
	3 = No line editing (flush mode). Input is transmitted to the requesting task exactly as entered at the terminal*. Before being transmitted, data accumulates in a buffer until an input request is received. At that time, the contents of the buffer (or the number of characters requested, if the buffer contains more than that number) is transmitted to the requesting task. If any characters remain in the buffer, they are saved for the next input request.
2	Echo control (corresponds to OSC characters C:E).
	0 = Echo. Characters entered at the terminal are "echoed" to the terminal's display screen.
	1 = Do not echo.
	* Except (1) signal characters (e.g., the Human Interface CONTROL-C) set by specifying "set signal" in the spec$func parameter of A$SPECIAL or S$SPECIAL, and (2) any enabled output control characters or OSC sequences.

3	Input parity control (corresponds to OSC characters C:R). Characters entered into the terminal have their parity bits (bit 7) set to 0 or not set by the Terminal Support Code, according to the value of the input parity control bit. 0 = Set parity bit to 0. 1 = Do not alter parity bit.
4	Output parity control (corresponds to OSC characters C:W). Characters being output to the terminal have their parity bits (bit 7) set to 0 or not set by the Terminal Support Code, according to the value of the output parity control bit. 0 = Set parity bit to 0. 1 = Do not alter parity bit.
5	Output control character control (corresponds to OSC characters C:O). This bit specifies whether output control characters are effective when entered at the terminal. The value of this bit applies only to output through this connection. Control characters are described in the *Extended iRMX II Device Drivers User's Guide*. Note that the output control characters are supported only on input from a terminal, not as output to a terminal. 0 = Accept output control characters in the input stream. 1 = Ignore output control characters in the input stream.
6-7	OSC control sequence enable/disable (corresponds to OSC characters C:C). These bits specify whether OSC control sequences should be acted upon when they appear in the input stream and, separately, when they appear in the output stream. These bits apply only to input or output through this connection. OSC control sequences are described in *Extended iRMX II Device Drivers User's Guide*.

		0 = Act upon OSC sequences that appear in either the input or output stream. 1 = Act upon OSC sequences in the input stream only. 2 = Act upon OSC sequences in the output stream only. 3 = Do not act upon any OSC sequences.
	8-15	Reserved bits. For future compatibility, set to 0.

terminal$flags — This word applies to the terminal and therefore to all connections to the terminal. If you attempt to set this field to zero, the Basic I/O System ignores your entry and leaves the field set to its previous value. The flags in this word are encoded as follows. (Bit 0 is the low-order bit.)

Bits	Value and Meaning
0	Reserved bit. Set to 1.
1	Line protocol indicator (corresponds to OSC characters T:L). Full-duplex terminals support simultaneous and independent input and output. Half-duplex terminals support independent input and output, but not simultaneously. 0 = Full duplex. 1 = Half duplex.
2	Output medium (corresponds to OSC characters T:H). 0 = Video display terminal (VDT). 1 = Printed (Hard copy).
3	Modem indicator (corresponds to OSC characters T:M). 0 = Not used with a modem. 1 = Used with a modem.

4-5

Input parity control bits (corresponding to OSC characters T:R) determines how the terminal driver handles input parity. The parity bit (bit 7) of each input byte can be used in a variety of ways. A byte has even parity if the sum of its bits is an even number. Otherwise, the byte has odd parity.

0 = Terminal driver always sets parity bit to 0.

1 = Terminal driver never alters the parity bit.

2 = Even parity is expected on input. The terminal driver uses the parity bit to indicate the presence (1) or absence (0) of an error on input. That is, the driver sets the parity bit to 0 unless the received byte has odd parity or there is some other error, such as (a) the received stop bit has a value of 0 (framing error) or (b) the previous character received has not yet been fully processed (overrun error).

3 = Odd parity is expected on input. The terminal driver uses the parity bit to indicate the presence (1) or absence (0) of an error on input. That is, the driver sets the parity bit to 0 unless the received byte has even parity or there is some other error, such as (a) the received stop bit has a value of 0 (framing error) or (b) the previous character received has not yet been fully processed (overrun error).

6-8

Output parity control bits (corresponding to OSC characters T:W). Determines how the terminal driver handles output parity. The parity bit (bit 7) of each output byte can be used in a variety of ways. A byte has even parity if the sum of its bits is an even number. Otherwise, the byte has odd parity.

0 = Terminal driver always sets parity bit to 0.

1 = Terminal driver always sets parity bit to 1.

2 = Terminal driver sets parity bit to give the byte even parity.

3 = Terminal driver sets parity bit to give the byte odd parity.

4 = Terminal driver does not alter the parity bit.

5-7 Invalid values.

9 Translation control (corresponds to OSC characters T:T). Translation refers to the ability to define certain control characters so that whenever these characters are entered at or written to a terminal, certain actions, usually cursor movements, take place automatically. Translation is described in the *Extended iRMX II Device Drivers User's Guide*.

0 = Do not enable translation.

1 = Enable translation.

10 Terminal axes sequence control (corresponds to OSC characters T:F). This specifies the order in which Cartesian-like coordinates of elements on a terminal's screen are to be listed or entered.

0 = List or enter the horizontal coordinate first.

1 = List or enter the vertical coordinate first.

11 Horizontal axis orientation control (corresponds to OSC characters T:F). This specifies whether the coordinates on the terminal's horizontal axis increase or decrease as you move from left to right across the screen.

0 = Coordinates increase from left to right.

1 = Coordinates decrease from left to right.

12 Vertical axis orientation control (corresponds to OSC characters T:F). This specifies whether the coordinates on the terminal's vertical axis increase or decrease as you move from top to bottom across the screen.

0 = Coordinates increase from top to bottom.

1 = Coordinates decrease from top to bottom.

13-15	Reserved bits. For future compatibility, set to 0.

NOTE

If bits 4-5 contain 2 or 3, and bits 6-8 also contain 2 or 3, then they must both contain the same value. That is, they must both reflect the same parity convention (even or odd).

in$baud$rate	The input baud rate indicator (corresponds to OSC characters T:I). If you attempt to set this field to zero, the Basic I/O System ignores your entry and leaves the field set to its previous value. The word is encoded as follows: 0 = Leave field set to the previous value. 1 = Use the input baud rate for output. Other = Actual output baud rate, such as 9600.
out$baud$rate	The output baud rate indicator (corresponds to OSC characters T:O). If you attempt to set this field to zero, the Basic I/O System ignores your entry and leaves the field set to its previous value. The word is encoded as follows: 0 = Leave field set to the previous value. 1 = Use the input baud rate for output. Other = Actual output baud rate, such as 9600. Most applications require the input and output baud rates to be equal. In such cases, use in$baud$rate to set the baud rate and specify a one for out$baud$rate.
scroll$lines	An operator at a terminal can enter a control character (default is CONTROL-W) when he/she is ready for data to appear on the terminal's display screen. The scroll$lines value (corresponding to OSC characters T:S) specifies the maximum number of lines that are to be sent to the terminal each time the operator enters the control character. If you attempt to set this field to zero, the Basic I/O System ignores your entry and leaves the field set to its previous value.
x$y$size	The low-order byte of this word specifies the number of character positions on each line of the terminal's screen (and corresponds to OSC characters T:X). The high-order byte specifies the number of lines on the terminal's screen (and corresponds to OSC characters T:Y).

x$y$offset

The low-order byte of this word specifies the value that starts the numbering sequence of both the X and Y axes (and corresponds to OSC characters T:U). The high-order byte specifies the value to which the numbering of the axes must "fall back" after reaching 127 (and corresponds to OSC characters T:V).

special$modes

This and the following fields apply only to buffered devices (such as the iSBC 544A and the iSBC 188/48 boards). These devices maintain their own input and output buffers separately from the ones managed by the Basic I/O System's Terminal Support Code. If you aren't sure whether you can set these fields, invoke S$SPECIAL with function code 4 to get the terminal attributes. If bit 15 of the special$modes field is set, your board is a buffered device and you can set the bits in special$modes and the following fields. (If your board is not a buffered device, setting any of the following fields will cause the Terminal Support Code to return an E$PARAM Condition Code.)

Bits	Value and Meaning
0	Flow control mode specifies whether the communications board sends flow control characters (selected by the fconchar and fcoffchar fields, but usually XON and XOFF) to turn input on and off (corresponds to the OSC characters T:G). The low-order bit (bit 0) controls this option, as follows: 0 = Disable flow control. 1 = Enable flow control. When flow control is enabled, the communication board can control the amount of data sent to it to prevent buffer overflow. This is especially important when communicating with another computer.
1	With the Special Character Mode (corresponds to OSC characters T:D) you can define up to four special characters. These special characters are different from the signal characters provided by the Terminal Support Code, though they may be signal characters. If your driver supports special characters, it processes these characters differently when the Special Character Mode is on.

	0 = Disable Special Character Mode.
	1 = Enable Special Character Mode.
2-14	Reserved bits. Set to 0.
15	Buffered Device Control. This bit is set by the Terminal Support Code to show if a device is buffered. If invoking the S$SPECIAL system call to get terminal attributes shows that this bit is set, then the special$modes bits and the data fields following are valid. If the Buffered Device Control bit is not set and you attempt to alter these data fields, an E$PARAM error is returned
	0 = Not a buffered device.
	1 = Buffered device.

The remaining fields in the structure apply only to buffered devices.

high$water$mark	When the communication board's buffer fills to contain the number of bytes represented by this field, the board's firmware sends the flow control "off" character to stop input. (This field corresponds to the OSC characters T:J.)
	The high-water mark of the iSBC 544A board is not configurable; therefore, setting this field has no effect on that board.
low$water$mark	When the number of bytes in the communication board's input buffer drops to the number represented by this field, the board's firmware sends the flow control "on" character to start input. (This field corresponds to the OSC characters T:K.)
	The low-water mark of the iSBC 544A board is not configurable; therefore, setting this field has no effect on that board.
fconchar	An ASCII character that the communication board sends to the connecting device when the number of bytes in its input buffer drops to the low-water mark. Normally this character tells the connecting device to resume sending data. (This field corresponds to the OSC characters T:P.)
	The fconchar for the iSBC 544A board is set to the XON (CONTROL-Q) character and is not configurable; therefore, setting this field has no effect on that board.

fcoffchar

An ASCII character that the communication board sends to the connecting device when the number of characters in its input buffer rises to the high-water mark. Normally this character tells the connecting device to stop sending data. (This field corresponds to the OSC characters T:Q.)

The fcoffchar for the iSBC 544A board is set to the XOFF (CONTROL-S) character and is not configurable; therefore, setting this field has no effect on that board.

link$parameter

This word specifies the characteristics of the physical link between the terminal and a device. Not all device drivers support link$parameter. (This field corresponds to the OSC characters T:N, and is supported by the Terminal Communications Controller driver.)

Bits	Value and Meaning
0-1	Parity 0 = No parity 1 = Invalid Value 2 = Even parity 3 = Odd parity
2-3	Character length 0 = 6 bits/character. 1 = 7 bits/character. 2 = 8 bits/character. 3 = Invalid Value
4-5	Number of stop bits. 0 = 1 stop bit. 1 = 1 1/2 stop bits. 2 = 2 stop bits.
6-14	Reserved
15	Check if this word is to be used 0 = not used 1 = used

If parity is enabled, an additional bit position beyond those specified in the Character Length control is added to the transmitted data and expected in received data. The received parity bit is transferred to the CPU as part of the data unless 8 bits/character is selected. If a parity error is detected on input, the character is discarded.

In the 6 and 7 bits/character modes unused bit positions in transmit data are ignored. Unused bits in receive data are set to 1. If a framing error is detected on input, the character is returned as an 8-bit null (00H).

Bit 15 is checked to see if this word is to be used. If set to 1, the driver passes the low-order byte to the controller, which sets the parity, character length, and stop bits. If set to 0, this word is skipped and the terminal$flags field is used.

spchiwater$mark — This word specifies the high-water mark used by the special character mode (bit 1 of special$modes) and is ignored if the special character mode is off. If your device driver supports the special character mode, the driver processes special characters differently when the number of characters in the input buffer reaches the high-water mark. You can define up to four special signal characters (corresponds to the OSC characters T:A)

special$char(4) — This array holds the characters you define as special characters (and corresponds to the OSC characters T:Z). If you define less than four special characters, then you must fill the remaining slots in the array with duplicates of the last character you define.

Designating Characters for Signaling from a Terminal Keyboard (Function Code 6)

You can use the S$SPECIAL system call to associate a keyboard character with a semaphore, so that whenever the character is entered into the terminal, the Basic I/O System automatically sends a unit to the semaphore. Up to 12 character-semaphore pairs can be so associated simultaneously; each character being associated with a different semaphore, if desired. Character-semaphore pairs are called Signal Characters.

To set up a signal character, call S$SPECIAL with a device connection, with spec$func equal to 6, and with data$ptr pointing to a structure of the following form:

```
DECLARE signal$pair STRUCTURE(
                        semaphore       TOKEN,
                        character       BYTE);
```

where

semaphore — A TOKEN for the semaphore that is to be associated with the character.

character — If the character value is in the range 0 to 1FH, or is 7FH, the terminal support code sends a unit to the associated semaphore when it receives the ASCII equivalent of this value.

If you add 20H to the character values in the 0 to 1FH range (making this range 20H to 3FH), or if the value is 40H, then the type ahead buffer (and the input buffer if this is a buffered device) is cleared and a unit is sent to the associated semaphore.

To delete a signal character, call S$SPECIAL with the semaphore field set to 0 and character set to the signal character to be deleted.

Tape Drive Functions (Function Codes 7, 8, 9, and 10)

The S$SPECIAL system call performs four different functions that apply to tape drives only. These functions include rewinding a tape, searching for file marks, writing file marks, and retentioning a tape.
To rewind a tape, call S$SPECIAL with the following information:

connection — A TOKEN for a connection to a physical file.

function — Seven.

data$ptr — Set to NIL.

This function terminates tape read and write operations and rewinds a tape to its load point. If the tape drive is performing a write operation when you invoke this call, the tape drive writes a file mark before it rewinds the tape.

To search for a file mark, call S$SPECIAL with the following information:

connection — A TOKEN for a connection to a physical file.

function — Eight.

data$ptr — A POINTER to a structure of the following form:

```
DECLARE read$file$mark      STRUCTURE(
                  search              BYTE);
```

where

search — A value indicating the direction of the search, as follows:

00 Search forward

0FFH Search backward (for start/stop drives only)

This function terminates tape read operations and moves the tape to the next file mark. Any outstanding requests are completed before this call takes effect.

To write a file mark, call S$SPECIAL with the following information:

connection	A TOKEN for a connection to a physical file.
function	Nine.
data$ptr	Set to NIL.

This function terminates tape write operations and writes a file mark at the current position on the tape.

To retention a tape, call S$SPECIAL with the following information:

connection	A TOKEN for a connection to a physical file.
function	Ten.
data$ptr	Set to NIL.

This function fast-forwards the tape to the end and then rewinds it to the load point.

Condition Codes

E$OK	0000H	No exceptional conditions.
E$CONN$NOT$OPEN	0034H	At least one of the following is true: • The connection is not open. • The connection was opened by A$OPEN rather than S$OPEN.
E$EXIST	0006H	The connection parameter is not a token for an existing object.
E$FLUSHING	002CH	The specified device is being detached.
E$IDDR	002AH	The requested function is not supported by the device containing the specified file.
E$IFDR	002FH	The Extended I/O System does not support the requested function for the file driver associated with the connection.
EIOHARD	0052H	A hard I/O error occurred. A retry is probably useless.

EIOMODE	0056H	One of the following is true: • A tape drive attempted to perform a read operation before the previous write operation completed. • A tape drive attempted to perform a write operation before the previous read operation completed.
EIONO$DATA	0055H	The tape drive attempted to read the next record, but it found no data.
EIOOPRINT	0053H	The device was off-line. Operator intervention is required.
EIOSOFT	0051H	A soft I/O error occurred. The I/O System tried to perform the operation a number of times and failed (the number of retries is a configuration parameter). Another retry might still be successful.
EIOUNCLASS	0050H	An unknown type of I/O error occurred.
EIOWRPROT	0054H	The volume is write-protected.
E$LIMIT	0004H	At least one of the following is true: • Either the calling task's job or the job's default user object is already involved in 255 (decimal) I/O operations. • The calling task's job is not an I/O job.
E$MEM	0002H	The memory available to the calling task's job is not sufficient to complete the call.
ENOTCONFIGURED	0008H	This call is not part of the present configuration.
ENOTCONNECTION	8042H	The connection parameter is a token for an object that is not a file connection.
E$PARAM	8004H	The function code is not a legitimate value.
E$SPACE	0029H	At least one of the following is true: • The call attempted to format a track that is beyond the end of the volume. • When formatting a RAM-disk or a tape, the call attempted to format a track other than track zero.

E$STREAM$SPECIAL	003CH	At least one of the following is true: • The calling task is attempting to satisfy a stream file request, but there is no request queued at the stream file. • The calling task attempted to satisfy a stream file request, but the only queued request is a query. • The calling task is querying a stream file, but the only request queued at the file is another query. The Extended I/O System removes both queries from the queue and returns this exception code.
E$SUPPORT	0023H	The specified connection was created by a task outside of the calling task's job.

The S$TRUNCATE$FILE system call removes information from the end of a named data file. This system call can be used only with named files.

```
CALL RQ$S$TRUNCATE$FILE(connection, except$ptr);
```

INPUT PARAMETER

connection	A TOKEN for a connection to the named data file that is to be truncated. The file pointer for this connection tells the Extended I/O System where to truncate the file. The byte indicated by the pointer is the first byte to be dropped from the file.

OUTPUT PARAMETER

except$ptr	A POINTER to a WORD where the Extended I/O System returns a condition code.

Description

This system call applies to named data files only. When called, it truncates a file. "Truncate" means to get rid of the data in the file from the current location of the file pointer to the end of the file.

Unless the file pointer is already where you want it, your task should use the S$SEEK system call to position the pointer before using the S$TRUNCATE$FILE system call.

Truncation will occur immediately, regardless of the status of other connections to the same file.

If the pointer is at or beyond the end-of-file, no truncation occurs.

Access Requirements

Three access requirements pertain to this system call. First the connection must be open for writing only or for both reading and writing. If this is not the case, your task can use the S$OPEN system call to open the connection.

Second, the connection must have update access to the file. Recall that the Extended I/O System computes a connection's access when the connection is created.

Third, the connection must have been created by a task within the calling task's job. If this is not the case, use the existing connection as a prefix, and have the calling task invoke the S$ATTACH$FILE system call.

Condition Codes

E$OK	0000H	No exceptional conditions.
E$CONN$NOT$OPEN	0034H	At least one of the following is true: • The connection is open in the wrong mode. It must be open for writing or for both reading and writing. • The connection is not open. • The connection was opened by an A$OPEN rather than an S$OPEN.
E$FACCESS	0026H	The connection does not have update access to the file.
E$EXIST	0006H	The connection parameter is not a token for an existing object.
E$IFDR	002FH	Your task is attempting to truncate a stream or physical file. The S$TRUNCATE$FILE system call can be used only on named files.
EIOHARD	0052H	A hard I/O error occurred. A retry is probably useless.
EIOOPRINT	0053H	The device was off-line. Operator intervention is required.
EIOSOFT	0051H	A soft I/O error occurred. The I/O System tried to perform the operation a number of times and failed (the number of retries is a configuration parameter). Another retry might still be successful.
EIOUNCLASS	0050H	An unknown type of I/O error occurred.
EIOWRPROT	0054H	The volume is write-protected.
E$LIMIT	0004H	At least one of the following is true: • The calling task's job is not an I/O job. • Either the calling task's job, or the job's default user object, is already involved in 255 (decimal) I/O operations.
E$MEM	0002H	The memory available to the calling task's job is not sufficient to complete the call.

ENOTCONFIGURED	0008H	This system call is not part of the present configuration.
ENOTCONNECTION	8042H	The connection parameter is a token for an object that is not a file connection.
E$SPACE	0029H	The truncation required writing the contents of a buffer to the file, but the volume was full.
E$SUPPORT	0023H	The connection was created by a task outside the calling task's job.

The S$UNCATALOG$CONNECTION deletes a logical name from the object directory of a job.

```
CALL RQ$S$UNCATALOG$CONNECTION(job, log$name$ptr, except$ptr);
```

Input Parameters

job	A TOKEN for a job. The Extended I/O System deletes the logical name from this job's object directory. Setting the job parameter to SELECTOR$OF(NIL) specifies the calling task's job.
log$name$ptr	A POINTER to a STRING (of 1 to 12 characters) containing the logical name to uncatalog. The name can be delimited with colons (:). The operating system removes the colons so that a logical name with colons is the same as one without (e.g., :F0: is effectively the same as F0). Colons do not count in the length of the name.

Output Parameter

except$ptr	A POINTER to a WORD where the Extended I/O System returns the condition code.

Description

Your tasks should invoke this system call to delete logical names that were added to the object directory by the S$CATALOG$CONNECTION system call.

Condition Codes

E$OK	0000H	No exceptional conditions.
E$EXIST	0006H	The job parameter is not a token for an existing object.
E$LIMIT	0004H	The calling task's job is not an I/O job.
ELOGNAME$-NEXIST	0045H	The call could not find the logical name in the job's object directory.

ELOGNAME$-SYNTAX	0040H	The syntax of the specified logical name is incorrect because at least one of the following conditions is true: • The STRING pointed to by the log$name$ptr parameter is of zero length or has a length greater than 12 (not including colons (:)). • The logical name contains invalid characters.
E$MEM	0002H	The memory available to the calling task's job is not sufficient to complete the call.
ENOTCONFIGURED	0008H	This system call is not part of the present configuration.
E$TYPE	8002H	The job parameter is a token for an object that is not a job.

The S$WRITE$MOVE system call writes a collection of bytes from a buffer to a file.

```
bytes$written = RQ$S$WRITE$MOVE(connection, buf$ptr, count,
                                except$ptr);
```

Input Parameters

connection	A TOKEN for the connection to the file in which the information is to be written.
buf$ptr	A POINTER to a contiguous collection of bytes that are to be written to the specified file.
count	A WORD containing the number of bytes to be written from the buffer to the file.

Output Parameters

bytes$written	A WORD containing the number of bytes that were actually written to the file. This number will always be equal to or less than the number specified in the count parameter.
except$ptr	A POINTER to a WORD where the Extended I/O System returns a condition code.

Description

This system call causes the Extended I/O System to write the specified number of bytes from the buffer to the file.

Access Control

To write information into a file, the connection parameter must satisfy the following two requirements:

- The connection must have been created by a task within the calling task's job. If this is not the case, the Extended I/O System returns an E$SUPPORT exception code.
- The connection must be open for writing or for both reading and writing.

If the file is a named data file, the access rights associated with the connection must permit the kind of writing being performed. That is, if you are writing over data in the file, the connection must have update access or you will get an exception code; if you are writing data beyond the end-of-file, the connection must have append access or you will receive an exception code.

The connection can have access rights for updating, appending, or both. For information regarding the process of assigning access to a connection, see the descriptions for the S$ATTACH$FILE and S$CREATE$FILE system calls.

Number of Bytes Actually Written

Occasionally, the Extended I/O System writes fewer bytes than requested by the calling task (upon return from the call, bytes$written is less than count). This happens under two circumstances:

- When the Extended I/O System encounters an I/O error. Your task will be informed of this circumstance because the Extended I/O System returns an exception code.
- When the volume to which your task is writing becomes full. The Extended I/O System informs your task of this condition by returning an E$SPACE exception code.

Where the Bytes are Written

The Extended I/O System writes the first byte starting at the byte pointed to by the file pointer. As the Extended I/O System writes the bytes, it also updates the pointer. After the writing operation is complete, the file pointer points to the byte immediately following the last byte written.

Use the S$SEEK system call to position the file pointer if you are performing random-access operations.

If your task is using a connection that has append access, the task can start a writing operation beyond (rather than at) the EOF. The Extended I/O System extends the file and performs the writing operation. If the file is extended, the extended section of the file contains unknown, random information (you can write data into this area later). For example, if the EOF is at location 200 and your task positions the file pointer at 250 and begins writing, locations 200 through 249 contain undetermined information.

Effects of Priority

The priority of the task invoking this system call can greatly affect the performance of the application system. For better performance, the priority of the invoking task should be equal to or lower than (numerically greater than) 130. If the priority of the calling task is greater than 130, the operating system cannot overlap the write operation with computation or with other I/O operations. (To find out how to set priorities for application tasks, refer to the *Extended iRMX II Nucleus User's Guide*.)

Condition Codes

E$OK	0000H	No exceptional conditions.
EBADBUFF	8023H	One of the following is true: • The specified source memory buffer is not writeable. • The specified source memory buffer crosses segment boundaries.
E$CONN$NOT$OPEN	0034H	At least one of the following is true: • The connection is not open for writing. • The connection is not open. • The connection was opened with A$OPEN rather than with S$OPEN.
E$EXIST	0006H	The connection parameter is not a token for an existing object.
E$FACCESS	0026H	The call tried to write beyond the end-of-file, but the connection specified does not have append access to the file.
E$FLUSHING	002CH	The specified device is being detached.
E$FRAGMENTATION	0030H	The file is too fragmented to be extended.
EIOHARD	0052H	A hard I/O error occurred. Another retry is probably useless.
EIOMODE	0056H	One of the following is true: • A tape drive attempted to perform a read operation before the previous write operation completed. • A tape drive attempted to perform a write operation before the previous read operation completed.
EIOOPRINT	0053H	The device was off-line. Operator intervention is required.
EIOSOFT	0051H	A soft I/O error occurred. The I/O System tried to perform the operation a number of times and failed (the number of retries is a configuration parameter). Another retry might still be successful.
EIOUNCLASS	0050H	An unknown type of I/O error occurred.
EIOWRPROT	0054H	The volume is write-protected.

E$LIMIT	0004H	At least one of the following is true: • The calling task's job is not an I/O job. • The calling task's job, or the job's default user object, is already involved in 255 (decimal) I/O operations.
E$MEM	0002H	The memory available to the calling task's job is not sufficient to complete the call.
ENOTCONFIGURED	0008H	This system call is not part of the present configuration.
ENOTCONNECTION	8042H	The connection parameter is a token for an object that is not a file connection.
E$PARAM	8004H	The calling task is attempting to write beyond the end of a physical file.
E$SPACE	0029H	The volume is full.
E$SUPPORT	0023H	The connection parameter refers to a connection that was created by a task outside of the calling task's job.

The VERIFY$USER system call validates a user's name and password.

```
CALL RQ$VERIFY$USER(user$t, name$ptr, password$ptr, except$ptr);
```

Input Parameters

user$t	A TOKEN for the user object to be verified.
name$ptr	A POINTER to a STRING containing the user name. This name would typically be entered from the console during dynamic logon. Only the first eight characters are used; any additional characters are ignored.
password$ptr	A POINTER to a STRING containing the unencrypted user password. This password would typically be entered from the console at the same time as the name$ptr parameter. Only the first eight characters are used.

Output Parameter

except$ptr	A POINTER to a WORD where the Extended I/O System returns the condition code.

Description

The VERIFY$USER system call validates a non-resident user's name and password. Validation means determining if the name and password supplied as parameters identify a predefined user of an extended EXTENDED iRMX II system. This system call searches the file :CONFIG:UDF (User Definition File) for a matching user name and password. (See the *Guide To The Extended iRMX II Interactive Configuration Utility* for information on the :CONFIG:UDF file.) The name must have the exact same form as it appears in the UDF for a match to occur. The password parameter is encrypted and then compared to the encrypted version in the UDF. The ID defined in the UDF is also compared with the ID contained in the user object.

If a matching name, password, and ID are found, the user object is modified to indicate the user has been verified. If iRMX-NET is configured into your system and the VERIFY$USER call succeeds, then you also gain access to remote files. (See the *iRMX Networking Software User's Guide* for more information on iRMX-NET.)

If the name is not found or if the password, once encrypted, does not match the encrypted password associated with the name in :CONFIG:UDF, or if the IDs are not the same, an error is returned and the user object is not modified.

The Human Interface can use the VERIFY$USER system call to check a dynamic logon process.

NOTE

The remote file driver will reject all user tokens created by the CREATE$USER system call unless the VERIFY$USER system call is used to verify the user tokens created.

Condition Codes

E$OK	0000H	No exceptional conditions.
EBADCALL	8005H	A task wrote over the interface library or over the EIOS job.
E$CONTEXT	0005H	The user TOKEN has already been verified.
E$DEVFD	0022H	The device cannot be used with the file driver as specified in the preceding logical attach operation.
E$DEVICE$-DETACHING	0039H	An I/O operation could not be performed on the device because it was being detached.
E$EXIST	0006H	The user TOKEN parameter is not valid.
E$FACCESS	0026H	The user does not have the proper access rights for the requested operation.
E$FLUSHING	002CH	The device is being detached.
E$FNEXIST	0021H	One of the following is true: • The file or a file in its path does not exist. • The specified physical device was not found.
E$FTYPE	0027H	A path component is not a directory file.
E$ILLVOL	002DH	The file driver given in the volume label conflicts with the file driver specified in the preceding logical attach operation.
E$INVALID$FNODE	003DH	The fnode associated with a file is either marked not allocated, or the fnode number is out of range. This file should be deleted.

EIOHARD	0052H	A hard error occurred; the BIOS cannot retry the request.
EIOMEM	0042H	The BIOS job did not have enough memory to perform the requested function.
EIOOPRINT	0053H	The device is off-line; operator intervention is required.
EIOSOFT	0051H	A soft error occurred and the BIOS has retried the operation and failed; a retry is not possible.
EIOUNCLASS	0050H	An unclassified I/O error occurred.
EIOWR$PROT	0054H	The volume is write protected.
E$LIMIT	0004H	The caller's job is not an I/O job.
ELOGNAME$-NEXIST	0045H	The logical name was not found in the caller's object directory, the global job object directory, or the root job object directory.
ELOGNAME$-SYNTAX	0040H	One of the following was true: • A leading colon in the pathname STRING indicated the start of a logical name, but a terminate colon was not found. • The logical name STRING has a length of 0 or more than 12 characters. • The logical name STRING contains invalid characters.
E$MEDIA	0044H	The device associated with the system call is off-line.
E$MEM	0002H	The caller's job does not have enough memory to perform the requested operation.
E$NAME$NEXIST	0049H	The name specified in this call is not defined.
E$NOPREFIX	8022H	The caller's job does not have a default prefix, or it is invalid.
ENOTCONFIGURED	0008H	This call is not part of the present configuration.
ENOTLOG$NAME	8040H	The token referred to by the logical name supplied does not refer to a valid device or file connection.
E$NOUSER	8021H	The caller's job does not have a default user or it is invalid.
E$PARAM	8004H	The name or the password contain invalid characters or the name length is equal to zero.

E$PASSWORD$-MISMATCH	004BH	The password is incorrect.
E$SHARE	0028H	The file cannot be shared using the requested access.
E$TYPE	8002H	The user$t parameter is not a TOKEN for a user object.
EUDFFORMAT	0048H	The UDF is not in the correct format.
EUIDNEXIST	004AH	The user ID present in the user token does not match that specified in the UDF.

Intel®

INDEX

A

B

B

D

E

F

G

H

L

M

R

S

T

V

W

intel

EXTENDED iRMX® II APPLICATION LOADER SYSTEM CALLS REFERENCE MANUAL

Intel Corporation
3065 Bowers Avenue
Santa Clara, California 95051

PREFACE

This manual documents the system calls of the Application Loader, a subsystem of the extended iRMX II Operating System. The information provided in this manual is intended as a reference to the system calls and provides detailed descriptions of each call.

READER LEVEL

This manual is intended for programmers who are familiar with the concepts and terminology introduced in the *Extended iRMX II Nucleus User's Guide* and with the PL/M-286 programming language.

CONVENTIONS

System call names appear as headings on the outside upper corner of each page. The first appearance of each system call name is printed in ink; subsequent appearances are in black.

Throughout this manual, system calls are shown using a generic shorthand (such as A$LOAD instead of RQ$A$LOAD). This convention is used to allow easier alphabetic arrangement of the calls. The actual PL/M-286 external-procedure names must be used in all calling sequences. The only exceptions to this convention are calls that exist only in extended iRMX II. These calls begin with RQE$ and appear in their complete form.

You can also invoke the system calls from assembly language programs, but you must adhere to the PL/M-286 calling sequences when doing so. For more information on these calling sequences refer to the *Extended iRMX II Programming Techniques Reference Manual*.

Intel®

CONTENTS

Intel®

iRMX® II APPLICATION LOADER SYSTEM CALLS

1.1. INTRODUCTION

This manual describes the PL/M-286 calling sequences for the system calls of the Application Loader.

Throughout this manual, PL/M-286 data types, such as BYTE, WORD, and SELECTOR are used. In addition, the extended iRMX II data type TOKEN is used. Data types always appear in capital letters. If your compiler supports the SELECTOR data type, a TOKEN can be declared literally as SELECTOR. Because TOKEN is not a PL/M-286 data type, you must declare it to be literally a SELECTOR every place you use it. Definitions of both PL/M-286 and extended iRMX II data types are given in the *Extended iRMX II Application Loader User's Guide*, Appendix A. The word "token" in lowercase refers to a value that the iRMX II Operating System returns to a TOKEN (the data type) when it creates an object.

1.2 RESPONSE MAILBOX PARAMETER

Three system calls described in this manual are asynchronous. These are the A$LOAD, A$LOADIOJOB, and RQEALOADIOJOB system calls. Your task must specify a mailbox whenever it invokes an asynchronous system call. The purpose of this mailbox is to receive a Loader Result Segment.

In general, the Loader Result Segment indicates the result of the loading operation. The format of a Loader Result Segment depends on which system call was invoked, so details about Loader Result Segments are included in descriptions of the A$LOAD, A$LOADIOJOB, and RQEALOADIOJOB system calls.

Avoid using the same response mailbox for more than one concurrent invocation of asynchronous system calls. This is necessary because it is possible for the Application Loader to return Loader Result Segments in an order different than the order of invocation. On the other hand, it is safe to use the same mailbox for multiple invocations of asynchronous system calls if only one task invokes the calls and the task always obtains the result of one call via RQ$RECEIVE$MESSAGE before making the next call.

1.3 CONDITION CODES

The Application Loader returns a condition code whenever a system call is invoked. If the call executes without error, the Application Loader returns the code E$OK. If an error occurs, the Application Loader returns a condition code.

This manual includes, for each of the Application Loader's system calls, descriptions of the condition codes that the system call can return. The system call manuals for the other layers of the extended iRMX II Operating System do the same thing for those layers. You can use the condition code information to write code to handle exceptional conditions that arise when system calls fail to perform as expected. See the *Extended iRMX II Nucleus User's Guide* for a discussion of condition codes and how to write code to handle them.

1.3.1 Condition Codes For Synchronous System Calls

For system calls that are synchronous (S$LOAD$IO$JOB, RQE$S$LOAD$IO$JOB, and S$OVERLAY), the Application Loader returns a single condition code each time the call is invoked. If your system has an exception handler, it will receive this code when an exceptional condition occurs, depending on how the exception$mode parameter is set. For more information see the *Extended iRMX II Nucleus User's Guide* and the *Extended iRMX II Interactive Configuration Utility Reference Manual.*

1.3.2 Condition Codes For Asynchronous System Calls

For system calls that are asynchronous (A$LOAD, A$LOADIOJOB, RQEALOADIOJOB), the Application Loader returns two condition codes each time the call is invoked. One code is returned after the sequential part of the system call is executed, and the other is returned after the concurrent part of the call is executed. Your task must process these two condition codes separately.

The *Extended iRMX II Application Loader User's Guide* describes the sequential and concurrent portions of asynchronous system calls.

1.3.2.1 Sequential Condition Codes

The Application Loader returns the sequential condition code in the word pointed to by the except$ptr parameter. If your system has an exception handler, it will receive this code when an exceptional condition occurs, depending upon how the exception$mode parameter is set.

1.3.2.2 Concurrent Condition Codes

The Application Loader returns the concurrent condition code in the Loader Result Segment it sends to the response mailbox. If the code is E$OK, the asynchronous loading operation ran successfully. If the code is other than E$OK, a problem occurred during the asynchronous loading operation, and your task must decide what to do about the problem. Regardless of the exception mode setting for the application, the exception handler is not invoked by concurrent condition codes, so your program must handle it.

1.4 SYSTEM CALL DICTIONARY

The following list is a summary of the extended iRMX II Application Loader system calls, together with a brief description of each call and the page where the description of the call begins.

Name	Description	Type	Page
A$LOAD	Loads object code or data into memory.	Asynchronous	4
A$LOAD$IO$JOB	Creates an I/O job, loads the job's code, and causes the job's task to run.	Asynchronous	10
RQEALOADIOJOB	Creates an I/O job with a memory pool of up to 16M bytes, and loads the code as the initial task.	Asynchronous	18
S$LOAD$IO$JOB	Creates an I/O job, loads the job's code, and causes the job's task to run.	Synchronous	26
RQESLOADIOJOB	Creates an I/O job with a memory pool of up to 16M bytes and loads the code as the initial task.	Synchronous	31
S$OVERLAY	Loads an overlay into memory.	Synchronous	36

The A$LOAD system call loads an object file from secondary storage into memory.

```
CALL RQ$A$LOAD(connection, response$mbox, except$ptr);
```

Input Parameters

connection — A TOKEN for a connection to the file that is to be loaded. The connection must satisfy all of the following requirements:

- It must have been created in the calling task's job.
- It must be a connection to a named file.
- The calling user must have had READ access to the file.
- It must be closed.

If all of these connection requirements are not met, the Application Loader returns an exception code.

response$mbox — A TOKEN for the mailbox to which the Application Loader sends the Loader Result Segment after the concurrent part of the system call finishes running. The format of the Loader Result Segment is given in the following DESCRIPTION section.

Output Parameter

except$ptr — A POINTER to a WORD where the Application Loader will place the condition code generated by the sequential part of the system call.

Description

A$LOAD allows your task to load object code files from secondary storage into memory. The object code to be loaded must be of the Single Task Loadable (STL) type with LODFIX records.

Unlike the A$LOAD$IO$JOB and S$LOADIOJOB system calls, A$LOAD cannot automatically cause the code to be executed as a task. The caller must explicitly cause the code to be executed.

Asynchronous Behavior

The A$LOAD system call is asynchronous. It allows the calling task to continue running while the loading operation is in progress. When the loading operation is finished, the Application Loader sends a Loader Result Segment to the mailbox designated by the response$mbox parameter. Refer to Appendix C of the *Extended iRMX II Application Loader User's Guide* for an explanation of how asynchronous system calls work.

File Sharing

The Application Loader does not expect exclusive access to the file. However, other tasks sharing the file are affected by the following:

- The other tasks should not attempt to share the connection passed to the Application Loader, but instead should obtain their own connections to the file.
- The Application Loader specifies "share with readers only" when opening the connection, so, during the loading operation, other tasks can access the file only for reading.

The A$LOAD Loader Result Segment

The Application Loader uses memory from the pool of the calling task's job to create the Loader Result Segment for this system call. The calling task should delete the segment after it is no longer needed. Creating multiple segments without deleting them can result in an E$MEM or E$SLOT exception code.

The Loader Result Segment has the following form:

```
STRUCTURE   (except$code            WORD,
             reserved$word$1        WORD,
             reserved$byte          BYTE,
             reserved$word$2        WORD,
             code$seg$offset        WORD,
             code$seg$base          TOKEN,
             stack$offset           WORD,
             stack$seg$base         TOKEN,
             stack$size             WORD,
             data$seg$base          TOKEN
             num$more$slots         BYTE
             more$slots(*)          TOKEN);
```

where:

except$code — A WORD containing the condition code for the concurrent part of the system call. If the code is other than E$OK, some problem occurred during the loading operation.

reserved$word$1 — Reserved for use by Intel.

reserved$byte	Reserved for use by Intel.
reserved$word$2	Reserved for use by Intel.
codesegoffset	A WORD containing the initial value for the loaded program's instruction pointer (IP register) taken from the Task State Segment (TSS) of the object file.
codesegbase	A TOKEN containing the initial value of the code segment selector.
stack$offset	A WORD containing the initial value of the stack pointer taken from the Task State Segment (TSS) of the object file
stacksegbase	A TOKEN containing the initial value of the stack segment selector.
stack$size	A WORD specifying the number of bytes required for the loaded program's stack. The Application Loader sets this value to 0 whenever stack$offset is 0 and stack$seg$base is SELECTOR$OF(NIL).
datasegbase	A TOKEN containing the initial value of the data segment selector taken from the Task State Segment (TSS) of the object file. The Application Loader sets this value to Selector$of(NIL) if the target file contains no initial data segment selector.
num$more$slots	A BYTE having a value between 0 and 255 that indicates how many GDT or LDT slots were allocated, this number includes the initial code, data, and stack segments. If the actual value was greater than 255, the value returned is set to 255.
more$slots(*)	A TOKEN ARRAY that lists the SELECTORS of all the segments that were allocated for the loaded program. The length of this array is contained in num$more$slots, up to the maximum of 255.

Using The Loader Result Segment

The contents of the result segment enable you to start the loaded code by creating a task or job. When doing this, you must specify the initial address, stack pointer, stack size, and data segment. These are all available in the Loader Result Segment.

Once the loaded program has stopped running, you may want to delete all the segments allocated for this program. Deleting these segments frees the memory for use by other tasks or jobs. To find tokens for the segments to delete, check the TOKEN ARRAY in more$slots(*).

Condition Codes

The A$LOAD system call can return condition codes at two different times. Codes returned to the calling task immediately after invocation of the system call are sequential condition codes. Codes returned after the concurrent part of the system call has finished running are concurrent condition codes. The following list is divided into two parts, one for sequential codes and one for concurrent codes.

Sequential Condition Codes

The Application Loader can return any of the following condition codes to the WORD pointed to by the except$ptr parameter of this system call.

E$OK	0000H	No exceptional conditions.
EBADHEADER	0062H	The object file contains an invalid header record.
E$CONN$NOT$OPEN	0034H	The Application Loader opened the connection but some other task closed the connection before the loading operation was begun.
E$CONN$OPEN	0035H	The calling task specified a connection that was already open.
E$EOF	0065H	The Application Loader encountered an unexpected End-Of-File while reading a record.
E$EXIST	0006H	At least one of the following is true: • The connection parameter is not a token for an existing object. • The msg$mbox parameter did not refer to an existing object.
E$FACCESS	0026H	The specified connection did not have "read" access to the file.
E$FLUSHING	002CH	The device containing the target file is being detached.
EIOHARD	0052H	A hard I/O error occurred. This means that another try is probably useless.
EIOOPRINT	0053H	The device containing the target file was off-line. Operator intervention is required.

EIOSOFT	0051H	A soft I/O error occurred. This means that the I/O System tried to perform the operation and failed, but another try might still be successful.
EIOUNCLASS	0050H	An unknown type of I/O error occurred.
E$LIMIT	0004H	At least one of the following is true: • The calling task's job has already reached its object limit. • Either the calling task's job, or the job's default user object, is already involved in 255 (decimal) I/O operations.
E$LOADER$SUPPORT	006FH	To load the target file requires capabilities not configured into the Application Loader.
E$MEM	0002H	The memory available to the calling task's job or the Basic I/O System is not sufficient to complete the call.
ENOTFILE$CONN	0032H	The calling task specified a connection to a device rather than to a named file.
E$SHARE	0028H	The calling task tried to open a connection to a file already being used by some other task, and the file's sharing attribute is not compatible with the open request.
E$SUPPORT	0023H	The specified connection was not created by the calling task's job.
E$TYPE	8002H	The connection parameter is a token for an object that is not a connection.

Concurrent Condition Codes

After the Application Loader attempts the loading operation, it returns a condition code in the except$code field of the Loader Result Segment. The Application Loader can return the following condition codes in this manner.

E$OK	0000H	No exceptional conditions.
E$EOF	0065H	The call encountered an unexpected End-Of-File.
E$EXIST	0006H	At least one of the following is true: • The mailbox specified in the response$mbox parameter was deleted before the loading operation was completed. • The device containing the file to be loaded was detached before the loading operation was completed.
E$FLUSHING	002CH	The device containing the target file is being detached.
EIOHARD	0052H	A hard I/O error occurred. This means that another try is probably useless.
EIOOPRINT	0053H	The device containing the target file was off-line. Operator intervention is required.
EIOSOFT	0051H	A soft I/O error occurred. This means that the I/O System tried to perform the operation and failed, but another try might still be successful.
EIOUNCLASS	0050H	An unknown type of I/O error occurred.
E$LIMIT	0004H	The calling task's job has already reached its object limit.
ENOLOADER$MEM	0067H	The memory pool of the newly created I/O job does not currently have a block of memory large enough to allow the Application Loader to run.
E$PARAM	8004H	The target file has a stack smaller than 16 bytes.

The A$LOAD$IO$JOB system call asynchronously loads an object file from secondary storage to main memory and creates an I/O job for it.

```
job = RQ$A$LOAD$IO$JOB(connection, pool$min, pool$max,
                       except$handler, job$flags, task$priority,
                       task$flags, msg$mbox, except$ptr);
```

Input Parameters

connection

A TOKEN for a connection to the file that the Application Loader will load. The connection must be a connection to a named file. Also, the connection must be closed, the user object specified when the connection was created must have had READ access, and the connection must have been created in the calling task's job.

The Application Loader opens the connection for sharing with readers only, so, during the loading operation, other tasks may access the file only for reading.

pool$min

A WORD containing a value the Application Loader uses to compute the pool$min size for the new I/O job that will be created for the loaded program.

pool$max

A WORD containing a value the Application Loader uses to compute the pool size for the new I/O job.

except$handler

A POINTER to a structure of the following form:

```
STRUCTURE(
        EXCEPTION$HANDLER$PTR   POINTER,
        EXCEPTION$MODE  BYTE);
```

This parameter is used as the input to RQE$CREATE$IO$JOB, when it is called to create a new job for the loaded code. If the exception handler pointer field is NIL, the new job will have the same exception handler as its parent. For more details, see the description of this parameter in RQE$CREATE$IO$JOB in the EXTENDED iRMX II EXTENDED I/O SYSTEM CALLS manual.

job$flags

A WORD specifying whether the Nucleus is to check the validity of objects used as parameters in system calls. Setting bit 1 (where bit 0 is the low-order bit) to 0 specifies that the Nucleus is to check the validity of objects. All bits other than bit 1 must be set to 0.

task$priority — A BYTE which,

- if equal to 0, indicates that the new job's initial task is to have a priority equal to the maximum priority of the initial job of the Extended I/O System.
- if not equal to 0, contains the priority of the initial task of the new job. If this priority is higher (numerically lower) than the maximum priority of the initial job of the Extended I/O System, an E$PARAM error occurs.

task$flags — A WORD indicating whether the initial task uses floating-point instructions, and whether to start the task immediately.

Set bit 0 (the low-order bit) to 1 if the task uses floating-point instructions; otherwise set it to 0.

Bit 1 indicates whether the initial task in the job should run immediately, or whether it should be suspended until a STARTIOJOB system call is issued to start it. Set it to 0 if the task is to be made ready immediately; set it to 1 if the task is to be suspended.

Set bits 2 through 15 to 0.

msg$mbox — A TOKEN for a mailbox that receives the Loader Result Segment after the loading operation is completed.

- You must always specify a valid mailbox TOKEN for this parameter.

The second purpose of this parameter is to receive an exit message from the newly created I/O job. The description of the CREATEIOJOB system call in the *Extended iRMX II Extended I/O System Calls* manual shows the format of an exit message.

The format of the Loader Result Segment is provided later in this description.

Output Parameters

job — A TOKEN, returned by the Application Loader, for the newly created I/O job. This token is valid only if the Application Loader returns an E$OK condition code to the WORD pointed to by the except$ptr parameter.

except$ptr — A POINTER to a WORD where the Application Loader is to place the condition code generated by the sequential part of the system call.

Description

This system call operates in two phases. The first phase occurs during the sequential part of this system call. (Refer to the *Application Loader User's Guide* for a discussion of the sequential and concurrent parts of an asynchronous system call.) During this first phase, the Application Loader does the following:

- Checks the validity of the header record of the target file, and calculates the required memory pool that will be given to the new job.
- Creates an I/O job. This I/O job is a child of the calling task's job. The initial task of this job is a loader task that will asynchronously load the object file.
- Returns a condition code reflecting the success or failure of the first phase. The Application Loader places this condition code in the WORD pointed to by the except$ptr parameter. If the condition code is not E$OK, the job token returned is not valid and the asynchronous part of the call did not execute.

The second phase occurs during the concurrent part of the system call. This part runs as the initial task in the new job and does the following:

- Loads the file designated by the connection parameter.
- Creates the task that will execute the loaded code. If there are no errors while the file is being loaded and if bit 1 of the task$flags parameter is 0, the concurrent part makes the task in the new job ready to run. If bit one of task$flags is one, the task will be suspended until an RQ$START$IO$JOB is issued for this task.
- Sends a Loader Result Segment to the mailbox specified by the msg$mbox parameter. One element in this segment is a condition code indicating the success or failure of the second phase.
- If the object file does not contain overlays, the loader task will delete itself at this point. If it does contain overlays, the loader task will be suspended, until a request to load an overlay is issued.

Restriction

This system call should be invoked only by tasks running within I/O jobs. Failure to heed this restriction causes a sequential exception condition.

Format Of The Loader Result Segment

The Loader Result Segment has the form described below. This structure is deliberately compatible with the structure of the message returned when an I/O job exits. (See the *Extended iRMX II Extended I/O System User's Guide* for a description of exit messages.)

```
STRUCTURE   (termination$code       WORD,
             except$code            WORD,
             job$token              TOKEN,
             return$data$len        BYTE,
             reserved$word$1        WORD,
             reserved$byte          BYTE,
             reserved$word$2        WORD,
             mem$requested          WORD,
             mem$received           WORD);
```

where:

termination$code — A WORD indicating the success or failure of the loading operation.

- A value of 100H indicates that the loading operation succeeded.
- A value of 2 indicates that the loading operation failed. In this case, your system should delete the newly created I/O job; the Application Loader doesn't do so.

except$code — A WORD containing the concurrent condition code. Codes and interpretations follow this description.

job$token — A TOKEN for the newly created I/O job.

return$data$len — A BYTE that indicates the length of the remainder of the data structure minus 13 bytes.

reserved$word$1 — Reserved for use by Intel.

reserved$byte — Reserved for use by Intel.

reserved$word$2 — Reserved for use by Intel.

mem$requested — A WORD indicating the number of 16-byte paragraphs the target file requested for the new job, including the memory needed for all segments and that needed for the job's memory pool.

mem$received — A WORD indicating the number of 16-byte paragraphs actually allocated to the new job.

Condition Codes

This system call can return condition codes at two different times. Codes returned to the calling task immediately after the invocation of the system call are considered sequential condition codes. Codes returned after the concurrent part of the system call has finished running are considered concurrent condition codes. The following list is divided into two parts -- one for sequential codes and one for concurrent codes.

Sequential Condition Codes

The Application Loader returns one of the following condition codes to the WORD pointed to by the except$ptr parameter:

E$OK	0000H	No exceptional conditions.
EBADHEADER	0062H	The object file contains an invalid header record.
E$CONN$NOT$OPEN	0034H	The Application Loader opened the connection, but some other task closed the connection before the loading operation was begun.
E$CONN$OPEN	0035H	The specified connection was already open.
E$CONTEXT	0005H	The calling task's job is not an I/O job.
E$EOF	0065H	The Application Loader encountered an unexpected End-Of-File while reading a record.
E$EXIST	0006H	At least one of the following is true: • The connection parameter is not a token for an existing object. • The calling task's job has no global job. Refer to the *Extended iRMX II Extended I/O System User's Guide* for a definition of global job. • The msg$mbox parameter does not refer to an existing object.
E$FACCESS	0026H	The specified connection does not have "read" access to the file.
E$FLUSHING	002CH	The device containing the target file is being detached.
EIOHARD	0052H	A hard I/O error occurred. This means that another try is probably useless.
EIOOPRINT	0053H	The device containing the target file is off-line. Operator intervention is required.

EIOSOFT	0051H	A soft I/O error occurred. This means that the I/O System tried to perform the operation and failed, but another try might still be successful.
EIOUNCLASS	0050H	An unknown type of I/O error occurred.
EIOWRPROT	0054H	The volume is write-protected.
EJOBPARAM	8060H	The pool$max parameter is both non-zero and smaller than the pool$min parameter.
EJOBSIZE	006DH	The pool$max parameter is non-0 and too small for the target file.
E$LOADER$SUPPORT	006FH	The target file requires capabilities not configured into the Application Loader.
E$MEM	0002H	The memory available to the calling task's job or the Basic I/O System is not sufficient to complete the call.
E$SLOT	000CH	The Global Descriptor Table (GDT) has no available slots.
ENOLOADER$MEM	0067H	The memory pool of the newly created I/O job does not currently have a block of memory large enough to allow the Application Loader to run.
ENOTCONFIGURED	0008H	This system call is not part of the present configuration.
ENOTFILE$CONN	0032H	The specified connection is to a device rather than to a named file.
E$PARAM	8004H	The value of the except$mode field within the except$handler structure lies outside the range 0 through 3.
E$SHARE	0028H	The calling task tried to open a connection to a file already being used by some other task, and the file's sharing attribute is not compatible with the open request.
E$SUPPORT	0023H	The specified connection was not created in this job.

E$TIME	0001H	The calling task's job is not an I/O job.
E$TYPE	8002H	The connection parameter is a token for an object that is not a connection.

Concurrent Condition Codes

After the Application Loader attempts the loading operation, it returns a condition code in the except$code field of the Loader Result Segment. The Application Loader can return the following condition codes:

E$OK	0000H	No exceptional conditions.
E$EOF	0065H	The call encountered an unexpected End-Of-File.
E$EXIST	0006H	At least one of the following is true: • The mailbox specified in the msg$mbox parameter was deleted before the loading operation was completed. • The device containing the target file was detached before the loading operation was completed.
E$FACCESS	0026H	The default user of the newly created I/O job does not have "read" access to the target file.
E$FLUSHING	002CH	The device containing the target file is being detached.
EIOHARD	0052H	A hard I/O error occurred. This means that another try is probably useless.
EIOOPRINT	0053H	The device containing the target file is off-line. Operator intervention is required.
EIOSOFT	0051H	A soft I/O error occurred. This means that the I/O System tried to perform the operation and failed, but another try might still be successful.
EIOUNCLASS	0050H	An unknown type of I/O error occurred.

E$LIMIT	0004H	At least one of the following is true: • The task$priority parameter is higher (numerically lower) than the newly created I/O job's maximum priority. This maximum priority is specified during the configuration of the Extended I/O System (if the job is a descendant of the Extended I/O System) or during configuration of the Human Interface (if the job is a descendant of the Human Interface). • Either the newly created I/O job, or its default user, is already involved in 255 (decimal) I/O operations. • The calling task's object directory is full. • The root object directory is full.
ENOLOADER$MEM	0067H	There is not sufficient memory available to the newly created I/O job or the Basic I/O System to allow the Application Loader to run.
E$NOSTART	006CH	The target file does not specify the entry point for the program being loaded.
E$PARAM	8004H	The target file has a stack smaller than 16 bytes.

The RQEALOADIOJOB system call asynchronously loads an object file from secondary storage to main memory and creates an I/O job for it. The difference between this call and A$LOAD$IO$JOB is the maximum memory pool for RQEALOADIOJOB is 16M bytes.

RQEALOADIOJOB creates a new job using RQE$CREATE$IO$JOB and loads the specified object file. The loaded file's code becomes the initial task of the new job. The calling task continues to run during the loading operation. If the task$flags parameter specifies delayed activation, a STARTIOJOB call must be issued to start the new task. If the task$flags parameter specifies immediate activation, the task becomes ready at the end of the loading operation.

```
job = RQE$A$LOAD$IO$JOB(connection, pool$min, pool$max,
        except$handler, job$flags, task$priority,
        task$flags, msg$mbox, except$ptr);
```

Input Parameters

connection	A TOKEN for a connection to the file that the Application Loader will load. The connection must be a connection to a named file. Also, the connection must be closed, the user object specified when the connection was created must have had READ access, and the connection must have been created in the calling task's job. The Application Loader opens the connection for sharing with readers only, so, during the loading operation, other tasks may access the file only for reading.
pool$min	A DWORD containing a value the Application Loader uses to compute the pool size for the new I/O job.
pool$max	A DWORD containing a value the Application Loader uses to compute the pool size for the new I/O job.

except$handler

A POINTER to a structure of the following form:

```
STRUCTURE(
        EXCEPTION$HANDLER$PTR   POINTER,
        EXCEPTION$MODE  BYTE);
```

If exception$handler$ptr is not NIL, then it is a POINTER to the first instruction of the new job's own exception handler. If exception$handler$ptr is NIL, the new job's exception handler is the system default exception handler. In both cases, the exception handler for the new task becomes the default exception handler for the job.

Set exception$mode to specify when control is to pass to the new task's exception handler. Encode the mode as follows:

Value	When Control Passes To Exception Handler
0	Never
1	On programmer errors only
2	On environmental conditions only
3	On all exceptional conditions

For more information regarding exception handlers and the exception mode, refer to the *Extended iRMX II Nucleus User's Guide*.

job$flags

A WORD specifying whether the Nucleus is to check the validity of objects used as parameters in system calls. Setting bit 1 (where bit 0 is the low-order bit) to 0 specifies that the Nucleus is to check the validity of objects. All bits other than bit 1 must be set to 0.

task$priority

A BYTE which,

- if equal to 0, indicates that the new job's initial task is to have a priority equal to the maximum priority of the initial job of the Extended I/O System.
- if not equal to 0, contains the priority of the initial task of the new job. If this priority is higher (numerically lower) than the maximum priority of the initial job of the Extended I/O System, an E$LIMIT error occurs.

task$flags

A WORD indicating whether the initial task uses floating-point instructions, and whether to start the task immediately.

Set bit 0 (the low-order bit) to 1 if the task uses floating-point instructions; otherwise set it to 0.

Bit 1 indicates whether the initial task in the job should run immediately, or whether it should be suspended until a STARTIOJOB system call is issued to start it. Set it to 0 if the task is to be made ready immediately; set it to 1 if the task is to be suspended.

Set bits 2 through 15 to 0.

msg$mbox — A TOKEN for a mailbox that receives the Loader Result Segment after the loading operation is completed. This parameter is similar to the corresponding parameter in the CREATEIOJOB system call in the Extended I/O System, with these exceptions:

- You must always specify a valid mailbox TOKEN for this parameter.
- SELECTOR$OF(NIL) may not be used as a value for the TOKEN.
- Each call to A$LOAD$IO$JOB requires a unique mailbox.

The second purpose of this parameter is to receive an exit message from the newly created I/O job. The description of the CREATEIOJOB system call in the *Extended iRMX II Extended I/O System Calls* manual shows the format of an exit message.

The format of the Loader Result Segment is provided later in this description.

Output Parameters

job — A TOKEN, returned by the Application Loader, for the newly created I/O job. This token is valid only if the Application Loader returns an E$OK condition code to the WORD pointed to by the except$ptr parameter.

except$ptr — A POINTER to a WORD where the Application Loader is to place the condition code generated by the sequential part of the system call.

Description

This system call operates in two phases. The first phase occurs during the sequential part of this system call. (Refer to the *Application Loader User's Guide* for a discussion of the sequential and concurrent parts of an asynchronous system call.) During this first phase, the Application Loader does the following:

- Checks the validity of the header record of the target file.

- Creates an I/O job. This I/O job is a child of the calling task's job. (Refer to the *Extended iRMX II Extended I/O System User's Guide* for a definition of I/O jobs.)
- Returns a condition code reflecting the success or failure of the first phase. The Application Loader places this condition code in the WORD pointed to by the except$ptr parameter.

The second phase occurs during the concurrent part of the system call. This part runs as the initial task in the new job and does the following:

- Loads the file designated by the connection parameter.
- Creates the task that will execute the loaded code. If there are no errors while the file is being loaded and if bit 1 of the task$flags parameter is 0, the concurrent part makes the task in the new job ready to run.
- Sends a Loader Result Segment to the mailbox specified by the msg$mbox parameter. One element in this segment is a condition code indicating the success or failure of the second phase.
- Deletes itself.

Restriction

This system call should be invoked only by tasks running within I/O jobs. Failure to heed this restriction causes a sequential exception condition.

Format Of The Loader Result Segment

The Loader Result Segment has the form described in this section. This structure is compatible with the structure of the message returned when an I/O job exits. (See the *Extended iRMX II Extended I/O System User'S Guide* for a description of exit messages.)

```
STRUCTURE   (termination$code   WORD,
             except$code        WORD,
             job$token          TOKEN,
             return$data$len    BYTE,
             reserved$word$1    WORD,
             reserved$byte      BYTE,
             reserved$word$2    WORD,
             mem$requested      WORD,
             mem$received       WORD);
```

where:

termination$code — A WORD indicating the success or failure of the loading operation.

- A value of 100H indicates that the loading operation succeeded.

- A value of 2 indicates that the loading operation failed. In this case, your system should delete the newly created I/O job; the Application Loader doesn't do so.

except$code	A WORD containing the concurrent condition code. Codes and interpretations follow this description.
job$token	A TOKEN for the newly created I/O job.
return$data$len	A BYTE that indicates the length of the remainder of the data structure minus 13 bytes.
reserved$word$1	Reserved for use by Intel.
reserved$byte	Reserved for use by Intel.
reserved$word$2	Reserved for use by Intel.
mem$requested	A WORD indicating the number of 16-byte paragraphs the target file requested for the new job, including the memory needed for all segments and that needed for the job's memory pool. If more than 1M byte was requested, this field will contain 0FFFFFH.
mem$received	A WORD indicating the number of 16-byte paragraphs actually allocated to the new job. If more than 1M byte was allocated, this field will contain 0FFFFFH.

Condition Codes

This system call can return condition codes at two different times. Codes returned to the calling task immediately after the invocation of the system call are considered sequential condition codes. Codes returned after the concurrent part of the system call has finished running are considered concurrent condition codes. The following list is divided into two parts -- one for sequential codes and one for concurrent codes.

Sequential Condition Codes

The Application Loader returns one of the following condition codes to the WORD pointed to by the except$ptr parameter:

E$OK	0000H	No exceptional conditions.
E$CONN$NOT$OPEN	0034H	The Application Loader opened the connection, but some other task closed the connection before the loading operation was begun.
E$CONN$OPEN	0035H	The specified connection was already open.
E$CONTEXT	0005H	The calling task's job is not an I/O job.

E$EXIST	0006H	At least one of the following is true: • The connection parameter is not a token for an existing object. • The calling task's job has no global job. Refer to the *Extended iRMX II Extended I/O System User's Guide* for a definition of global job. • The msg$mbox parameter does not refer to an existing object.
E$FACCESS	0026H	The specified connection does not have "read" access to the file.
E$FLUSHING	002CH	The device containing the target file is being detached.
EIOHARD	0052H	A hard I/O error occurred. This means that another try is probably useless.
EIOOPRINT	0053H	The device containing the target file is off-line. Operator intervention is required.
EIOSOFT	0051H	A soft I/O error occurred. This means that the I/O System tried to perform the operation and failed, but another try might still be successful.
EIOUNCLASS	0050H	An unknown type of I/O error occurred.
EJOBPARAM	8060H	The pool$max parameter is both non-zero and smaller than the pool$min parameter.
EJOBSIZE	006DH	The pool$max parameter is non-0 and too small for the target file.
E$LOADER$SUPPORT	006FH	The target file requires capabilities not configured into the Application Loader.
E$MEM	0002H	The memory available to the calling task's job or the Basic I/O System is not sufficient to complete the call.
ENOLOADER$MEM	0067H	The memory pool of the newly created I/O job does not currently have a block of memory large enough to allow the Application Loader to run.

ENOTCONFIGURED	0008H	This system call is not part of the present configuration.
ENOTFILE$CONN	0032H	The specified connection is to a device rather than to a named file.
E$PARAM	8004H	The value of the except$mode field within the except$handler structure lies outside the range 0 through 3.
E$SHARE	0028H	The calling task tried to open a connection to a file already being used by some other task, and the file's sharing attribute is not compatible with the open request.
E$SUPPORT	0023H	The specified connection was not created in this job.
E$TIME	0001H	The calling task's job is not an I/O job.
E$TYPE	8002H	The connection parameter is a token for an object that is not a connection.
E$SLOT	000CH	The Global Descriptor Table (GDT) has no available slots.

Concurrent Condition Codes

After the Application Loader attempts the loading operation, it returns a condition code in the except$code field of the Loader Result Segment. The Application Loader can return the following condition codes:

E$OK	0000H	No exceptional conditions.
E$EOF	0065H	The call encountered an unexpected End-Of-File.
E$EXIST	0006H	At least one of the following is true: • The mailbox specified in the msg$mbox parameter was deleted before the loading operation was completed. • The device containing the target file was detached before the loading operation was completed.

E$FACCESS	0026H	The default user of the newly created I/O job does not have "read" access to the target file.
E$FLUSHING	002CH	The device containing the target file is being detached.
EIOHARD	0052H	A hard I/O error occurred. This means that another try is probably useless.
EIOOPRINT	0053H	The device containing the target file is off-line. Operator intervention is required.
EIOSOFT	0051H	A soft I/O error occurred. This means that the I/O System tried to perform the operation and failed, but another try might still be successful.
EIOUNCLASS	0050H	An unknown type of I/O error occurred.
E$LIMIT	0004H	At least one of the following is true: • The task$priority parameter is higher (numerically lower) than the newly created I/O job's maximum priority. This maximum priority is specified during the configuration of the Extended I/O System (if the job is a descendant of the Extended I/O System) or during configuration of the Human Interface (if the job is a descendant of the Human Interface). • Either the newly created I/O job, or its default user, is already involved in 255 (decimal) I/O operations. • The calling task's object directory is full. • The root object directory is full.
ENOLOADER$MEM	0067H	There is not sufficient memory available to the newly created I/O job or the Basic I/O System to allow the Application Loader to run.
E$NOSTART	006CH	The target file does not specify the entry point for the program being loaded.
E$PARAM	8004H	The target file has a stack smaller than 16 bytes.

The S$LOAD$IO$JOB system call synchronously loads an object file from secondary storage to memory and creates an I/O job for it.

RQSLOADIOJOB creates a new job using RQE$CREATE$IO$JOB and loads the specified object file. The loaded file's code becomes the initial task of the new job. The calling task is suspended during the loading operation. If the task$flags parameter specifies delayed activation, a STARTIOJOB call must be issued to start the new task. If the task$flags parameter specifies immediate activation, the task becomes ready at the end of the loading operation.

```
job = RQ$S$LOAD$IO$JOB(path$ptr, pool$min, pool$max,
                       except$handler, job$flags, task$priority,
                       task$flags, msg$mbox, except$ptr);
```

Input Parameters

path$ptr — A POINTER to a STRING containing a path name for the named file with the object code to be loaded. The path name must conform to the Extended I/O System path name syntax for named files. If you are not familiar with extended iRMX II path name syntax, refer to the *Extended iRMX II Extended I/O System User's Guide*.

pool$min — A WORD containing a value that the Application Loader uses to compute the pool size for the new I/O job. See the DESCRIPTION section for details.

pool$max — A WORD containing a value that the Application Loader uses to compute the pool size for the new I/O job. See the DESCRIPTION section for details.

except$handler — A POINTER to a structure of the following form:

```
STRUCTURE(
    EXCEPTION$HANDLER$PTR   POINTER,
    EXCEPTION$MODE  BYTE);
```

If exception$handler$ptr is not NIL, then it is a POINTER to the first instruction of the new job's own exception handler. If exception$handler$ptr is NIL, the new job's exception handler is the system default exception handler. In both cases, the exception handler for the new task becomes the default exception handler for the job.

Set the exception$mode to tell the Application Loader when to pass control to the new task's exception handler. Encode the mode as follows:

Value	When Control Passes To Exception Handler
0	Control never passes to handler
1	On programmer errors only
2	On environmental conditions only
3	On all exceptional conditions

For more information regarding exception handlers and the exception mode, refer to the *Extended iRMX II Nucleus User's Guide*.

job$flags

A WORD specifying whether the Nucleus is to check the validity of objects used as parameters in system calls. Setting bit 1 (where bit 0 is the low-order bit) to 0 specifies that the Nucleus is to check the validity of objects. All bits other than bit 1 must be set to 0.

task$priority

A BYTE which,

- if equal to 0, indicates that the new job's initial task is to have a priority equal to the maximum priority of the initial job of the Extended I/O System.
- if not equal to 0, contains the priority of the initial task of the new job. If this priority is higher (numerically lower) than the maximum priority of the initial job of the Extended I/O System, an E$LIMIT error occurs.

task$flags

A WORD indicating whether the initial task uses floating-point instructions, and whether to start the task immediately.

Set bit 0 (the low-order bit) to 1 if the task uses floating-point instructions; otherwise set it to 0.

Bit 1 indicates whether the initial task in the job should run immediately, or whether it should be suspended until a STARTIOJOB system call is issued to start it. Set bit 1 to 0 if the task is to be made ready immediately; set it to 1 if the task is to be suspended.

Set bits 2 through 15 to 0.

msg$mbox

A TOKEN for a mailbox that receives an exit message from the newly created I/O job. This parameter is similar to the CREATEIOJOB system call documented in the *Extended iRMX II Extended I/O System Calls* manual, with these exceptions:

- You must always specify a valid mailbox TOKEN for this parameter.
- SELECTOR$OF(NIL) may not be used as a value for the TOKEN.
- Each call to S$LOAD$IO$JOB requires a unique mailbox.

Output Parameters

job	A TOKEN, returned by the Application Loader, for the newly created I/O job. This token is valid only if the Application Loader returns an E$OK condition code to the WORD specified by the except$ptr parameter.
except$ptr	A POINTER to a WORD where the Application Loader is to place a condition code.

Description

This system call performs the same function as A$LOAD$IO$JOB. The only difference between the calls is that S$LOAD$IO$JOB is synchronous. That is, the calling task resumes running only after the call has completed its attempt to create an I/O job and a user task in that job.

The Application Loader does not necessarily have exclusive access to the file being loaded. During the loading operation, however, if other tasks are also using the file, they may access the file only for reading.

NOTE

This system call should be invoked only by tasks running within I/O jobs. Failure to heed this restriction causes the Application Loader to return an E$CONTEXT exception code.

Condition Codes

The Application Loader returns one of the following condition codes to the WORD specified by the except$ptr parameter of this system call:

E$OK	0000H	No exceptional conditions.
EBADHEADER	0062H	The object file contains an invalid header record.
E$CONTEXT	0005H	The calling task's job is not an I/O job.

E$EOF	0065H	The call encountered an unexpected End-Of-File.
E$EXIST	0006H	At least one of the following is true: • The msg$mbox parameter is not a token for an existing object. • The calling task's job has no global job. (Refer to the *Extended iRMX II Extended I/O System User's Guide* for a definition of global job.) • The device containing the target file was detached.
E$FACCESS	0026H	The default user object for the new I/O job does not have "read" access to the specified file.
E$FNEXIST	0021H	The specified target file, or some file in the specified path, does not exist or is marked for deletion.
E$FLUSHING	002CH	The device containing the target file is being detached.
E$INVALID$FNODE	003DH	The fnode for the specified file is invalid, so the file must be deleted.
EIOHARD	0052H	A hard I/O error occurred. This means that another try is probably useless.
EIOJOB	0047H	The calling task's job is not an I/O job.
EIOOPRINT	0053H	The device containing the target file is off-line. Operator intervention is required.
EIOSOFT	0051H	A soft I/O error occurred. This means that the I/O System tried to perform the operation and failed, but another try might still be successful.
EIOUNCLASS	0050H	An unknown type of I/O error occurred.
EJOBPARAM	8060H	The pool$max parameter is nonzero and smaller than the pool$min parameter.
EJOBSIZE	006DH	The pool$max parameter is nonzero and too small for the target file.

E$LIMIT	0004H	At least one of the following is true: • The task$priority parameter is higher (numerically lower) than the newly created I/O job's maximum priority. This maximum priority is specified during the configuration of the Extended I/O System (if the job is a descendant of the Extended I/O System) or of the Human Interface (if the job is a descendant of the Human Interface). • Either the newly created I/O job or its default user object is already involved in 255 (decimal) I/O operations.
E$LOADER$SUPPORT	006FH	The target file requires capabilities not configured into the Application Loader.
E$MEM	0002H	The memory available to the calling task's job is not sufficient to complete the call.
ENOLOADER$MEM	0067H	The memory pool of the newly created I/O job does not currently have a block of memory large enough to allow the Application Loader to run.
ENOSTART	006CH	The target file does not specify the entry point for the program being loaded.
ENOTCONFIGURED	0008H	This system call is not part of the present configuration.
E$PARAM	8004H	At least one of the following is true: • The value of the except$mode field within the except$handler structure lies outside the range 0 through 3. • The target file requested a stack smaller than 16 bytes.
E$PATHNAME$SYNTAX	003EH	The specified pathname contains one or more invalid characters.
E$TIME	0001H	The calling task's job is not an I/O job.
E$TYPE	8002H	The connection parameter is a token for an object that is not a connection.

The RQESLOADIOJOB system call creates an I/O job containing the Application Loader task, which loads the code for the user task from secondary storage. The RQESLOADIOJOB allows you to specify memory pools of up to 16M bytes using the DWORD parameters, pool$min and pool$max.

```
job = RQ$RQE$S$LOAD$IO$JOB(path$ptr, pool$min, pool$max,
                    except$handler, job$flags, task$priority,
                    task$flags, msg$mbox, except$ptr);
```

Input Parameters

path$ptr — A POINTER to a STRING containing a path name for the named file with the object code to be loaded. The path name must conform to the Extended I/O System path name syntax for named files. If you are not familiar with extended iRMX II path name syntax, refer to the *Extended iRMX II Extended I/O System User's Guide*.

pool$min — A DWORD containing a value that the Application loader uses to compute the pool size for the new I/O job.

pool$max — A DWORD containing a value that the Application Loader uses to compute the pool size for the new I/O job.

except$handler — A POINTER to a structure of the following form:

```
STRUCTURE(
    EXCEPTION$HANDLER$PTR    POINTER,
    EXCEPTION$MODE           BYTE);
```

If exception$handler$ptr is not NIL, then it is a POINTER to the first instruction of the new job's own exception handler. If exception$handler$ptr is NIL, the new job's exception handler is the system default exception handler. In both cases, the exception handler for the new task becomes the default exception handler for the job.

Set exception$mode to tell the Application Loader when to pass control to the new task's exception handler. Encode the mode as follows:

Value	When Control Passes To Exception Handler
0	Never
1	On programmer errors only
2	On environmental conditions only
3	On all exceptional conditions

For more information regarding exception handlers and the exception mode, refer to the *Extended iRMX II Nucleus User's Guide*.

job$flags

A WORD specifying whether the Nucleus is to check the validity of objects used as parameters in system calls. Setting bit 1 (where bit 0 is the low-order bit) to 0 specifies that the Nucleus is to check the validity of objects. All bits other than bit 1 must be set to 0.

task$priority

A BYTE which,

- if equal to 0, indicates that the new job's initial task is to have a priority equal to the maximum priority of the initial job of the Extended I/O System.
- if not equal to 0, contains the priority of the initial task of the new job. If this priority is higher (numerically lower) than the maximum priority of the initial job of the Extended I/O System, an E$PARAM error occurs.

task$flags

A WORD indicating whether the initial task uses floating-point instructions, and whether to start the task immediately.

Set bit 0 (the low-order bit) to 1 if the task uses floating-point instructions; otherwise set it to 0.

Bit 1 indicates whether the initial task in the job should run immediately, or whether it should be suspended until a STARTIOJOB system call is issued to start it. Set bit 1 to 0 if the task is to be made ready immediately; set it to 1 if the task is to be suspended.

Set bits 2 through 15 to 0.

msg$mbox

A TOKEN for a mailbox that receives an exit message from the newly created I/O job. This parameter is similar to the CREATEIOJOB system call documented in the *Extended iRMX II Extended I/O System Calls* manual, with these exceptions:

- You must always specify a valid mailbox TOKEN for this parameter.
- SELECTOR$OF(NIL) may not be used as a value for the TOKEN.

- Each call to RQESLOADIOJOB requires a unique mailbox.

Output Parameters

job	A TOKEN, returned by the Application Loader, for the newly created I/O job. This token is valid only if the Application Loader returns an E$OK condition code to the WORD specified by the except$ptr parameter.
except$ptr	A POINTER to a WORD where the Application Loader is to place a condition code.

Description

This system call performs the same function as A$LOAD$IO$JOB. The only difference between the calls is that RQESLOADIOJOB is synchronous. That is, the calling task resumes running only after the call has completed its attempt to create an I/O job and a user task in that job.

The Application Loader does not necessarily have exclusive access to the file being loaded. During the loading operation, however, if other tasks are also using the file, they may access the file only for reading.

NOTE

This system call should be invoked only by tasks running within I/O jobs. Failure to heed this restriction causes the Application Loader to return an E$CONTEXT exception code.

Condition Codes

The Application Loader returns one of the following condition codes to the WORD specified by the except$ptr parameter of this system call:

E$OK	0000H	No exceptional conditions.
E$CONTEXT	0005H	The calling task's job is not an I/O job.
E$EOF	0065H	The call encountered an unexpected End-Of-File.
E$EXIST	0006H	At least one of the following is true: • The msg$mbox parameter is not a token for an existing object.

		• The calling task's job has no global job. (Refer to the *Extended iRMX II Extended I/O System User's Guide* for a definition of global job.) • The device containing the target file was detached.
E$FACCESS	0026H	The default user object for the new I/O job does not have "read" access to the specified file.
E$FNEXIST	0021H	The specified target file, or some file in the specified path, does not exist or is marked for deletion.
E$FLUSHING	002CH	The device containing the target file is being detached.
E$INVALID$FNODE	003DH	The fnode for the specified file is invalid, so the file must be deleted.
EIOHARD	0052H	A hard I/O error occurred. This means that another try is probably useless.
EIOJOB	0047H	The calling task's job is not an I/O job.
EIOOPRINT	0053H	The device containing the target file is off-line. Operator intervention is required.
EIOSOFT	0051H	A soft I/O error occurred. This means that the I/O System tried to perform the operation and failed, but another try might still be successful.
EIOUNCLASS	0050H	An unknown type of I/O error occurred.
EJOBPARAM	8060H	The pool$max parameter is nonzero and smaller than the pool$min parameter.
EJOBSIZE	006DH	The pool$max parameter is nonzero and too small for the target file.
E$LIMIT	0004H	At least one of the following is true:

		• The task$priority parameter is higher (numerically lower) than the newly created I/O job's maximum priority. This maximum priority is specified during the configuration of the Extended I/O System (if the job is a descendant of the Extended I/O System) or of the Human Interface (if the job is a descendant of the Human Interface). • Either the newly created I/O job or its default user object is already involved in 255 (decimal) I/O operations.
E$LOADER$SUPPORT	006FH	The target file requires capabilities not configured into the Application Loader.
E$MEM	0002H	The memory available to the calling task's job is not sufficient to complete the call.
ENOLOADER$MEM	0067H	The memory pool of the newly created I/O job does not currently have a block of memory large enough to allow the Application Loader to run.
E$SLOT	000CH	The Global Descriptor Table (GDT) has no available slots.
ENOSTART	006CH	The target file does not specify the entry point for the program being loaded.
ENOTCONFIGURED	0008H	This system call is not part of the present configuration.
E$PARAM	8004H	At least one of the following is true: • The value of the except$mode field within the except$handler structure lies outside the range 0 through 3. • The target file requested a stack smaller than 16 bytes.
E$PATHNAME$SYNTAX	003EH	The specified pathname contains one or more invalid characters.
E$TIME	0001H	The calling task's job is not an I/O job.
E$TYPE	8002H	The connection parameter is a token for an object that is not a connection.

In programs with overlays, the root module of the program calls S$OVERLAY to load overlay modules. The root module must be loaded using one of the system calls that create an I/O job.

```
CALL RQ$S$OVERLAY(name$ptr, except$ptr);
```

Input Parameter

name$ptr	A POINTER to a STRING containing the name of an overlay. The overlay name should have only uppercase letters, both in this string and when you specify the name in the overlay definition file. For information about OVL286, refer to the *iAPX 286 Utilities User's Guide For iRMX II Systems*.

Output Parameter

except$ptr	A POINTER to a WORD in which the Application Loader will place a condition code.

Description

Root modules issue this system call when they want to load an overlay module. The *Extended iRMX II Application Loader User's Guide* describes overlays.

Synchronous Behavior

This system call is synchronous. The calling task resumes running only after the system call has completed its attempt to load the overlay.

File Sharing

The Application Loader does not expect exclusive access to the file containing the overlay module. However, while the overlay is being loaded, if other tasks are also using the file, they can access the file only for reading.

Condition Codes

The Application Loader returns one of the following condition codes to the calling task:

E$OK	0000H	No exceptional conditions.
E$EOF	0065H	The call encountered an unexpected End-Of-File.
E$EXIST	0006H	The specified device does not exist.
E$FLUSHING	002CH	The device containing the target file is being detached.
EIOHARD	0052H	A hard I/O error occurred. This means that another try is probably useless.
EIOOPRINT	0053H	The device containing the target overlay is off-line. Operator intervention is required.
EIOSOFT	0051H	A soft I/O error occurred. This means that the I/O System tried to perform the operation and failed, but another try might still be successful.
EIOUNCLASS	0050H	An unknown type of I/O error occurred.
E$LIMIT	0004H	Either the calling task's job, or its default user object, is already involved in 255 (decimal) I/O operations.
E$NOMEM	0068H	The memory pool of the new I/O job does not have a block of memory large enough to allow the Application Loader to load the overlay module.
ENOTCONFIGURED	0008H	This system call is not part of the present configuration.
E$OVERLAY	006EH	The overlay name indicated by the name$ptr parameter does not match any overlay module name in your overlay definition file.

Intel®

INDEX

A

C

F

R

S

U

intel

EXTENDED iRMX® II HUMAN INTERFACE SYSTEM CALLS REFERENCE MANUAL

Intel Corporation
3065 Bowers Avenue
Santa Clara, California 95051

PREFACE

This manual documents the system calls of the Human Interface, a subsystem of the iRMX® II Operating System. The information provided in this manual is intended as a reference to the system calls and provides detailed descriptions of each call.

READER LEVEL

This manual is intended for programmers who are familiar with the concepts and terminology introduced in the *Extended iRMX II Nucleus User's Guide* and with the PL/M-286 programming language.

CONVENTIONS

System call names appear as headings on the outside upper corner of each page. The first appearance of each system call name is printed in ink; subsequent appearances are in black.

Throughout this manual, system calls are shown using a generic shorthand (such as CGETCHAR instead of RQCGET$CHAR). This convention is used to allow easier alphabetic arrangement of the calls. The actual PL/M-286 external-procedure names must be used in
all calling sequences.

You can also invoke the system calls from assembly language, but you must obey the PL/M-286 calling sequences when doing so. For more information on these calling sequences refer to the *Extended iRMX II Programming Techniques Reference Manual.*

Intel®

CONTENTS

Intel®

iRMX® II HUMAN INTERFACE SYSTEM CALLS

1.1 INTRODUCTION

The Human Interface system calls described in this manual are presented in alphabetical sequence and are not organized by function. However, the calls are grouped according to function in the System Call Dictionary. For each call, the following information is provided:

- Brief functional description
- Calling sequence format
- Input parameter definitions, if applicable
- Output parameter definitions, if applicable
- Considerations and consequences of call usage
- Potential exception codes and their possible causes

This manual refers to PL/M-286 data types such as BYTE, WORD, and SELECTOR, and extended iRMX II data types such as STRING. These words, when used as data types, are always capitalized; their definitions are found in Appendix A of the *Extended iRMX II Human Interface User's Guide*. This manual also refers to an extended iRMX II data type called TOKEN. You declare a TOKEN to be literally a SELECTOR. The word "token" in lowercase refers to a value that the extended iRMX II Operating System assigns to an object. The operating system returns this value to a TOKEN (the data type) when it creates the object.

If you are a new user of the Human Interface calls, you should review the parsing considerations in the *Extended iRMX II Human Interface User's Guide* before writing your source code. You should also review the format of the released Human Interface commands. They are described in the *Operator's Guide To The Extended iRMX II Human Interface*.

This manual assumes that you are familiar with terms and concepts of the extended iRMX II Operating System. If you are not, you should read *Introduction To The Extended iRMX II Operating System* and the chapters in the *Extended iRMX II Nucleus User's Guide* that refer to the terms "memory pool" and "catalog."

1.2 System Call Dictionary

System Call	Synopsis	Page
	I/O Processing Calls	
CGETINPUT$CONNECTION	Return an EIOS connection for the specified input file.	16
CGETOUTPUT$CONNECTION	Return an EIOS connection for the specified output file.	27
	Command Parsing Calls	
CSBACKUP$CHAR	Move the parsing buffer pointer back one bye.	4
CSGET$CHAR	Get a character from the command line	12
CGETINPUT$PATHNAME	Parse the command line and return an input pathname.	21
CGETPARAMETER	Parse the command line for the next parameter and return it as a keyword name and a value.	36
CGETOUTPUT$PATHNAME	Parse the command line and return an output pathname.	33
CSSET$PARSE$BUFFER	Parse a buffer other than the current command line.	56
CSGET$COMMAND$NAME	Return the command name by which the current command was invoked	14
	Message Processing Calls	
CSFORMAT$EXCEPTION	Create a default message for an exception code and place it in a user buffer.	9
CSSENDCORESPONSE	Send a message to the command output (CO) and read a response from the command input (CI).	48
CSSENDEORESPONSE	Send a message to the operator's terminal and return a response from that terminal.	51

C$BACKUP$CHAR, a command parsing call, moves the parsing buffer pointer back one byte.

```
CALL C$BACKUP$CHAR(except$ptr);
```

Output Parameter

except$ptr	A POINTER to a WORD in which the Human Interface returns a condition code.

Description

When an operator invokes a command, the command's parameters are placed in a parsing buffer. The C$BACKUP$CHAR system call allows you to move the parsing buffer's pointer back one character for each occurrence of the call.

Exception Codes

E$OK	0000H	No exceptional conditions were encountered.
E$LIMIT	0004H	The parsing buffer's pointer is at the start of the command.
E$CONTEXT	0005H	The job that issued the call is not an I/O job.

C$CREATE$COMMAND$CONNECTION.I.C$CREATE$COMMAND$CONNECTION;, a command processing call, creates an extended iRMX II object called a command connection that is required in order to invoke commands programmatically.

```
command$conn = RQ$C$CREATE$COMMAND$CONNECTION(default$ci,
                                    default$co, flags, except$ptr);
```

Input Parameters

default$ci	A TOKEN for a connection that is used as the :CI: (console input) for any commands you invoke using this command connection.
default$co	A TOKEN for a connection that is used as the :CO: (console output) for any commands you invoke using this command connection.
flags	A WORD used to indicate that the Human Interface should return an E$ERROR$OUTPUT exception code if the system call C$SEND$EO$RESPONSE is used by any task. If the user wants the exception code, then the parameter is set to one (1); otherwise, the parameter must equal zero (0).

Output Parameters

command$conn	A TOKEN which receives a token for the new command connection.
except$ptr	A POINTER to a WORD in which the Human Interface returns a condition code.

Description

You can use this call when you want to invoke a command programmatically instead of interactively. It provides a place to store command lines until the command invocation is complete.

The call creates an extended iRMX II object called a command connection and returns a token for that command connection. The C$SEND$COMMAND system call can use this token to send command lines to the command connection, where they are stored until the command invocation is complete. The command connection also defines default :CI: and :CO: connections that are used by any commands invoked via this command connection.

C$CREATE$COMMAND$CONNECTION

Although a job can contain multiple command connections, the tasks in a job cannot create command connections simultaneously. Attempts to do this result in an E$CONTEXT exception code. Therefore, it is advisable for one task to create the command connections for all tasks in the job.

A possible application where the parameter "flags" might be set to one is when you want to write a custom CLI to perform batch jobs in the background. When any of the background batch jobs attempt to communicate with the terminal through C$SEND$EO$RESPONSE, the Human Interface issues an exception code. In this way, the Human Interface keeps all the jobs in the background. Note that the Human Interface CLI does not provide resident background or batch processing capability.

Exception Codes

E$OK	0000H	No exceptional conditions were encountered.
E$ALREADY$ATTACHED	0038H	While creating a STREAM file, the Extended I/O System was unable to attach the :STREAM: device because another task had already invoked a Basic I/O system call to attach the :STREAM: device.
E$CONTEXT	0005H	At least one of the following is true: • Two command connections were being created simultaneously by two tasks in the same job. • The calling task's job was not created by the Human Interface.(Refer to the *Extended iRMX II Extended I/O System User's Guide* for information.)
EDEVDETACHING	0039H	The :STREAM: device, the default$ci device, or the default$co device was in the process of being detached.

E$DEVFD	0022H	The Extended I/O System attempted the physical attachment of the :STREAM: device. This device had formerly been only logically attached. In the process, the Extended I/O System found that the device and the device driver specified in the logical attachment are incompatible. The operating system would not have returned this exception code if the :STREAM: device had been properly configured.
E$EXIST	0006H	The default$ci or default$co parameter is not a token for an existing object.
E$FNEXIST	0021H	The :STREAM: file does not exist or is marked for deletion.
E$IFDR	002FH	The Extended I/O System attempted to obtain information about the default$ci or default$co connection. However, the request for information resulted in an invalid file driver request.
E$INVALID$FNODE	003DH	The fnode associated with the specified file (:CI: or :CO:) is invalid. Delete the file.
EIOMEM	0042H	The Basic I/O System job does not currently have a block of memory large enough to allow the Human Interface to create a stream file.
E$LIMIT	0004H	At least one of the following is true: • The object directory of the calling task's job has already reached the maximum object directory size. • The calling task's job has exceeded its object limit. • The calling task's job (or that job's default user object) is already involved in 255 (decimal) I/O operations. • The calling task's job was not created by the Human Interface. (Refer to the *Extended iRMX II Extended I/O System User's Guide* for information.)

ELOGNAME$NEXIST	0045H	The call was unable to find the logical name :STREAM: in the object directories of the local job, the global job, or the root job.
E$MEM	0002H	The memory available to the calling task's job is not sufficient to complete the call.
E$NOPREFIX	8022H	The calling task's job does not have a valid default prefix.
ENOTCONNECTION	8042H	The default$ci or default$co parameter is a token for an object that is not a connection to a file.
ENOTLOG$NAME	8040H	The logical name :STREAM: refers to an object that is not a file or device connection.
E$NOUSER	8021H	The calling task's job does not have a valid default user object.
E$PARAM	8004H	The system call forced the Extended I/O System to attempt the physical attachment of the :STREAM: device, which had formerly been only logically attached. In the process, the Extended I/O System found that the stream file driver is not properly configured into your system, so the physical attachment is not possible.
E$SUPPORT	0023H	The default$ci or default$co device connection was not created by this job.

C$DELETE$COMMAND$CONNECTION, a command processing call, deletes a command connection object and frees the memory used by the command connection's data structures.

```
CALL RQ$C$DELETE$COMMAND$CONNECTION(command$conn, except$ptr);
```

Input Parameter

command$conn	A TOKEN for a valid command connection.

Output Parameter

except$ptr	A POINTER to a WORD in which the Human Interface returns a condition code.

Description

This call deletes a command connection object previously defined in a C$CREATE$COMMAND$CONNECTION call and releases the memory used by the command connection's data structures.

Exception Codes

E$OK	0000H	No exceptional conditions were encountered.
E$EXIST	0006H	The command$conn parameter is not a token for an existing object.
E$TYPE	8002H	The command$conn parameter is a token for an object that is not a command connection object.

C$FORMAT$EXCEPTION, a message processing call, creates a default message for a given exception code and writes that message into a user-provided string.

```
CALL RQ$C$FORMAT$EXCEPTION(buff$p, buff$max, exception$code,
                           reserved$byte, except$ptr);
```

Input Parameters

buff$max	A WORD that specifies the maximum number of bytes that may be contained in the string pointed to by buff$p.
exception$code	A WORD containing the exception code value for which a message is to be created.
reserved$byte	A BYTE reserved for future use. Its value must be one (1).

Output Parameters

buff$p	A POINTER to a STRING into which the Human Interface concatenates the formatted exception message.
except$ptr	A POINTER to a WORD in which the Human Interface returns a condition code.

Description

C$FORMAT$EXCEPTION causes the Human Interface to create a message for the exception code. The message consists of the exception code value and exception code mnemonic in the following format:

value : mnemonic

where the mnemonics are provided by the Human Interface from an internal table and are listed in the *Operator's Guide To The Extended iRMX II Human Interface.*

The call concatenates the message to the end of the string pointed to by the buff$p pointer and updates the count byte to reflect the addition. If a string is not already present in the buffer, the first byte of the buffer must be a zero. The message added by C$FORMAT$EXCEPTION will not be longer than 30 characters (not including the length byte).

Exception Codes

E$OK	0000H	No exceptional conditions were encountered.
E$PARAM	8004H	An undefined exception code value was specified.
E$STRING	8084H	The message to be returned exceeds the length limit of 255 characters.
E$STRING$BUFFER	0081H	The buffer pointed to by the buff$p parameter is not large enough to contain the exception message.

CGETCHAR, a command parsing call, gets a character from the parsing buffer.

```
char = RQ$C$GET$CHAR(except$ptr);
```

Output Parameters

char	A BYTE in which the Human Interface places the next character of the parsing buffer. A null (00H) character is returned when the parsing buffer's pointer is at the end of the parsing buffer.
except$ptr	A POINTER to a WORD in which the Human Interface returns a condition code.

Description

When an operator invokes a command, the command's parameters are placed in a parsing buffer. The CGETCHAR system call gets a single character from that buffer and moves the parsing buffer pointer to the next character. Consecutive calls to CGETCHAR return consecutive characters from the parsing buffer.

Exception Codes

E$OK	0000H	No exceptional conditions were encountered.
E$CONTEXT	0005H	The calling task's job was not created by the Human Interface. Refer to the *Extended iRMX II Extended I/O System User's Guide* for information.
E$LIMIT	0004H	At least one of the following situations occurred: • The object directory of the calling task's job has already reached the maximum object directory size. • The calling task's job has exceeded its object limit. • The calling task's job was not created by the Human Interface. Refer to the *Extended iRMX II Extended I/O System User's Guide* for information.

E$MEM	0002H	The memory available to the calling task's job is not sufficient to complete the call.

CGETCOMMAND$NAME, a command parsing call, obtains the pathname of the command that the operator used when invoking the command.

```
CALL RQ$C$GET$COMMAND$NAME (path$name$p, name$max, except$ptr);
```

Input Parameter

name$max	A WORD that specifies the maximum length in bytes of the string pointed to by the path$name$p parameter.

Output Parameters

path$name$p	A POINTER to a buffer that receives a STRING containing the name of the command.
except$ptr	A POINTER to a WORD in which the Human Interface returns a condition code.

Description

If a command needs to know the name under which it was invoked, the CGETCOMMAND$NAME returns this information. This information is available to each command and is stored in a buffer that is separate from the parsing buffer. Therefore, calling CGETCOMMAND$NAME does not obtain information from the parsing buffer, nor does it move the parsing pointer.

If the operator invokes the command without specifying a logical name, the Human Interface automatically searches a number of directories for the command. In such cases, the value returned by CGETCOMMAND$NAME also includes the directory name (such as :SYSTEM:, :PROG:, or :$:) as a prefix to the command name.

Exception Codes

E$OK	0000H	No exceptional conditions were encountered.
E$LIMIT	0004H	The calling task's job was not created by the Human Interface.
E$PATHNAME$SYNTAX	003EH	The specified pathname contains invalid characters.

E$STRING$BUFFER	0081H	The buffer pointed to by the path$name$p parameter is not large enough to contain the command name.
E$TIME	0001H	The calling task's job was not created by the Human Interface.

CGETINPUT$CONNECTION, an I/O processing call, returns an Extended I/O System connection to the specified input file.

```
connection = RQ$C$GET$INPUT$CONNECTION(path$name$p, except$ptr);
```

Input Parameter

path$name$p	A POINTER to a buffer that receives a STRING. (The path of the specified input file.)

Output Parameters

connection	A TOKEN in which the operating system returns the token for the connection to the specified pathname.
except$ptr	A POINTER to a WORD in which the Human Interface returns a condition code.

Description

CGETINPUT$CONNECTION obtains a connection to the specified file. This connection is open for reading and has the following attributes:

- Read only
- Accessible to all users
- Has two 1024-byte buffers (This is the default size.)

CGETINPUT$CONNECTION causes an error message to be displayed at the operator's terminal (:CO:) whenever the operating system encounters an exceptional condition. The exceptional condition that triggers the error message can either be one of those listed for CGETINPUT$CONNECTION or it can be one of those associated with the Extended I/O System calls S$ATTACH$FILE and S$OPEN. The following messages can occur:

- <pathname>, file does not exist

  The input file does not exist.

- <pathname>, invalid file type

  The input file was a data file and a directory was required, or vice versa.

- <pathname>, invalid logical name

The input pathname contains a logical name that is longer than 12 characters, that contains unmatched colons, invalid characters, or zero characters.

- <pathname>, logical name does not exist

  The input pathname contains a logical name that does not exist.

- <pathname>, READ access required

  The user does not have read access to the input file.

- <pathname>, <exception value>:<exception mnemonic>

  An exceptional condition occurred when CGETINPUT$CONNECTION attempted to obtain the input connection. The <exception value> and <exception mnemonic> portions of the message indicate the exception code encountered. Refer to "Exception Codes" in this call description and to the descriptions of S$ATTACH$FILE and S$OPEN in the *Extended iRMX II Extended I/O System Calls Reference Manual.*

Exception Codes

E$OK	0000H	No exceptional conditions were encountered.
E$ALREADY$ATTACHED	0038H	The device containing the file specified in the path$name$p parameter is already attached.
E$CONTEXT	0005H	At least one of the following is true: • The calling task's job was not created by the Human Interface. (Refer to the *Extended iRMX II Extended I/O System User's Guide* for information.) • The calling task's job was not created by the Human Interface.
EDEVDETACHING	0039H	The device specified in the path$name$p parameter is in the process of being detached.
E$DEVFD	0022H	The call attempted the physical attachment of a device that had formerly been only logically attached. In the process, the call found that the device and the device driver specified in the logical attachment were incompatible.
E$EXIST	0006H	The specified device does not exist.
E$FACCESS	0026H	The specified connection does not have read access to the file.

E$FNEXIST	0021H	At least one of the following is true: • The target file does not exist or is marked for deletion. • While attaching the file pointed to by the path$name$p parameter, the call attempted the physical attachment of the device as a named device. It could not complete this process because the device specified when the logical attachment was made was not defined during configuration.
E$FTYPE	0027H	The path pointed to by the path$name$p parameter contained a file name that should have been the name of a directory, but is not. (Except for the last path component, each file in a pathname must be a named directory.)
E$ILLVOL	002DH	The call attempted the physical attachment of the specified device as a named device. This device had formerly been only logically attached. The call found that the volume did not contain named files. This prevented the call from completing physical attachment because the named file driver was requested during logical attachment.
E$INVALID$FNODE	003DH	The fnode for the specified file is invalid, so the file must be deleted.
EIOHARD	0052H	While attempting to access the file specified in the path$name$p parameter, the call detected a hard I/O error. Another call is useless.
EIOMEM	0042H	While attempting to create a connection, the call needed memory from the Basic I/O subsystem's memory pool. However, the Basic I/O System job does not currently have a block of memory large enough to allow this call to run to completion.
EIONOT$READY	0053H	While attempting to access the file specified in the path$name$p parameter, the call found that the device was off-line. Operator intervention is required.

EIOSOFT	0051H	While attempting to access the file specified in the path$name$p parameter, the call detected a soft I/O error. It tried the operation again but was unsuccessful. Another try might be successful.
EIOUNCLASS	0050H	An unknown type of I/O error occurred while this call tried to access the file given in the path$name$p parameter.
E$LIMIT	0004H	At least one of the following is true: • The calling task's job or the job's default user object is already involved in 255 (decimal) I/O operations. • The calling task's job was not created by the Human Interface. • The object limit of the calling job has been reached.
ELOGNAME$-NEXIST	0045H	The pathname for the specified device contains an explicit logical name. The call was unable to find this name in the object directories of the local job, the global job, or the root job.
ELOGNAME$SYNTAX	0040H	The pathname pointed to by the path$name$p parameter contains a logical name. This logical name contains an unmatched colon, is longer than 12 characters, has zero (0) characters, or contains invalid characters.
E$MEDIA	0044H	The specified device was off-line.
E$MEM	0002H	The memory available to the calling task's job is not sufficient to complete the call.
E$NOPREFIX	8022H	The calling task's job does not have a valid default prefix.
ENOTLOG$NAME	8040H	The logical name specified by the path$name$p parameter does not refer to a file or device connection.
E$NOUSER	8021H	The calling task's job does not have a valid default user.

E$PARAM	8004H	At least one of the following is true: • The system call forced the Extended I/O System to attempt the physical attachment of the device referenced by the path$name$p parameter. This device had formerly been only logically attached. In the process, the Extended I/O System found that the logical attachment referred to a file driver (named, physical, or stream) that is not configured into your system, so the physical attachment is not possible. • The connection to the specified file cannot be opened for reading.
E$PATHNAME$SYNTAX	003EH	The specified pathname contains invalid characters.
E$SHARE	0028H	The file sharing attribute currently does not allow new connections to the file to be opened for reading.
E$STREAM$SPECIAL	003CH	The call attempted to attach a stream file and in so doing issued an invalid stream file request.

CGETINPUT$PATHNAME, a command parsing call, gets a pathname from the list of input pathnames in the parsing buffer.

```
CALL RQ$C$GET$INPUT$PATHNAME(path$name$p, path$name$max,
                             except$ptr);
```

Input Parameter

path$name$max — A WORD that specifies the maximum length in bytes of the string pointed to by the path$name$p parameter. The maximum length that you can specify is 256 bytes (255 characters for the pathname and one byte for the count).

Output Parameters

path$name$p — A POINTER to a STRING which receives the next pathname in the pathname list. A zero-length string indicates that there are no more pathnames.

except$ptr — A POINTER to a WORD in which the Human Interface returns a condition code.

Description

The first call to CGETINPUT$PATHNAME retrieves the entire input pathname list and moves the parsing pointer to the next parameter. CGETINPUT$PATHNAME stores the list in an internal buffer and returns the first pathname in the string pointed to by the path$name$p parameter. Succeeding calls to CGETINPUT$PATHNAME return additional pathnames from the input pathname list but do not move the parsing pointer. CGETINPUT$PATHNAME denotes the end of the pathname list by returning a zero-length string.

CGETINPUT$PATHNAME accepts wild-card characters in the last component of a pathname. It treats a pathname that contains a wild-card as a list of pathnames. To obtain each pathname, it searches in the parent directory of the component containing the wild-card, comparing the "wild-carded" name with the names of all files in the directory. It returns the next pathname that matches.

The pathname returned by CGETINPUT$PATHNAME can be used for any purpose. However, it is most often used in a call to CGETINPUT$CONNECTION, to obtain a connection.

Exception Codes

E$OK	0000H	No exceptional conditions were encountered.
E$ALREADY$ATTACHED	0038H	The device containing the file pointed to by the path$name$p parameter is already attached.
E$CONTEXT	0005H	At least one of the following is true: • The calling task's job was not created by the Human Interface. (Refer to the *Extended iRMX II Extended I/O System User's Guide* for more information.) • The task called CGETOUTPUT$PATHNAME before calling CGETINPUT$PATHNAME.
EDEVDETACHING	0039H	The device pointed to by the path$name$p parameter is in the process of being detached.
E$DEVFD	0022H	The Extended I/O System attempted the physical attachment of a device that had formerly been only logically attached. In the process, the Extended I/O System found that the device and the device driver specified in the logical attachment were incompatible.
E$EXIST	0006H	At least one of the following is true: • The connection to the parent directory of the file pointed to by the path$name$p parameter is not a token for the existing job. • The calling task's job was not created by the Human Interface.
E$FACCESS	0026H	The connection used to open the directory does not have read access to the directory.
E$FLUSHING	002CH	The device containing the directory was in the process of being detached.
E$FNEXIST	0021H	At least one of the following is true: • The target file does not exist or is marked for deletion.

		• While attaching the parent directory of the file pointed to by the path$name$p parameter, the I/O System attempted the physical attachment of the device as a named device. It could not complete this process because the device specified when the logical attachment was made was not defined during configuration.
E$FTYPE	0027H	The path pointed to by the path$name$p parameter contained a file name that should have been the name of a directory, but is not. (Except for the last file, each file in a pathname must be a named directory.)
E$IFDR	002FH	The specified file is a stream or physical file.
E$ILLVOL	002DH	The call attempted the physical attachment of the specified device as a named device. This device had formerly been only logically attached. The call found that the volume did not contain named files. This prevented the call from completing physical attachment because the named file driver was requested during logical attachment.
E$INVALID$FNODE	003DH	The fnode for the specified file is invalid, so the file must be deleted.
EIOHARD	0052H	While attempting to access the parent directory of the file pointed to by the path$name$p parameter, the call detected a hard I/O error. This means that another call is probably useless.
EIOMEM	0042H	While attempting to create a connection, this call needed memory from the Basic I/O System's memory pool. However, the Basic I/O System job does not currently have a block of memory large enough to allow this call to run to completion.

EIONOT$READY	0053H	While attempting to access the parent directory of the file pointed to by the path$name$p parameter, this call detected that the device was off-line. Operator intervention is required. C$FORMAT$EXCEPTION returns the value EIONOT$READY for this code.
EIOSOFT	0051H	While attempting to access the parent directory of the file pointed to by the path$name$p parameter, this call detected a soft I/O error. It tried the operation again, but was unsuccessful. Another try might be successful.
EIOUNCLASS	0050H	An unknown type of I/O error occurred while this call tried to access the parent directory of the file pointed to by the path$name$p parameter.
E$LIMIT	0004H	At least one of the following is true: • The calling task's job has already reached its object limit. • The calling task's job or the job's default user object is already involved in 255 (decimal) I/O operations. • The calling task's job was not created by the Human Interface.
E$LIST	0085H	The last value of the input pathname list is missing. For example, "ABLE,BAKER," has no value following the second comma.
ELOGNAME$NEXIST	0045H	The pathname for the specified device contains an explicit logical name. The call was unable to find this name in the object directory of the local job, the global job, or the root job.
ELOGNAME$SYNTAX	0040H	The pathname pointed to by the path$name$p parameter contains a logical name that has an unmatched colon, is longer than 12 characters, has zero (0) characters, or contains invalid characters.
E$MEDIA	0044H	The specified device was off-line.

E$MEM	0002H	The memory available to the calling task's job is not sufficient to complete the call.
E$NOPREFIX	8022H	The calling task's job does not have a valid default prefix.
ENOTLOG$NAME	8040H	The logical name specified by the path$name$p parameter does not refer to a file or device connection.
E$NOUSER	8021H	The calling task's job does not have a valid default user object.
E$PARAM	8004H	At least one of the following is true: • The Extended I/O System attempted the physical attachment of the device pointed to by the path$name$p parameter. This device had formerly been only logically attached. In the process, the Extended I/O System found that the logical attachment referred to a file driver (named, physical, or stream) that is not configured into your system, so the physical attachment is not possible. • The connection to the parent directory cannot be opened for reading.
E$PARSE$TABLES	8080H	The call detected an error in an internal table used by the Human Interface.
E$PATHNAME$SYNTAX	003EH	The specified pathname contains invalid characters.
E$SHARE	0028H	The connection to the parent directory cannot be opened for reading.
E$STREAM$SPECIAL	003CH	The Extended I/O System issued an invalid stream file request when an attempt to attach a stream file failed.
E$STRING	8084H	The pathname to be returned exceeds the length limit of 255 characters.
E$STRING$BUFFER	0081H	The buffer pointed to by the path$name$p parameter was not large enough for the pathname to be returned.

E$SUPPORT	0023H	This call attempted to read the parent directory of the pathname pointed to by the path$name$p parameter. However, the file driver corresponding to that directory does not support this operation.
E$WILDCARD	0086H	The pathname to be returned contains an invalid wild-card specification.

CGETOUTPUT$CONNECTION, an I/O processing call, parses the command line and returns an Extended I/O System connection referring to the requested output file.

```
connection = RQ$C$GET$OUTPUT$CONNECTION(path$name$p, preposition,
                                        except$ptr);
```

Input Parameters

path$name$p — A POINTER to a STRING containing the pathname of the file to be accessed.

preposition — A BYTE that defines which preposition to use to create the output file. Use one of the following values to specify the preposition mode:

Value	Meaning
0	Use same preposition as was returned by the last CGETOUTPUT$PATHNAME call
1	TO
2	OVER
3	AFTER
4-255	Undefined, results in an error

Output Parameters

connection — A TOKEN in which the Human Interface returns a token for the connection to the output file.

except$ptr — A POINTER to a WORD in which the Human Interface returns a condition code.

Description

CGETOUTPUT$CONNECTION obtains a connection to the specified file.

This connection is open for writing and has the following attributes:

- Write only
- Accessible to all
- Has two 1024-byte buffers

If the call to CGETOUTPUT$CONNECTION specifies the TO preposition and the output file already exists, CGETOUTPUT$CONNECTION issues the following message to the terminal (:CO:):

```
<pathname>, already exists, OVERWRITE?
```

If the operator enters Y, y, R, or r, CGETOUTPUT$CONNECTION returns a connection to the existing file, allowing the command to write over the file. Any other response causes CGETOUTPUT$CONNECTION to return an E$FACCESS exception code.

CGETOUTPUT$CONNECTION causes an error message to be displayed at the operator's terminal (:CO:) whenever an exceptional condition occurs. The exceptional condition that causes the error message can be one of those listed below or one associated with an Extended I/O System call. The following messages can occur:

- <pathname>, DELETE access required

  The user does not have delete access to an existing file.

- <pathname>, directory ADD entry access required

  The user does not have add entry access to the parent directory.

- <pathname>, file does not exist

  The output file does not exist.

- <pathname>, invalid file type

  The output file was a data file and a directory was required, or vice versa.

- <pathname>, invalid logical name

  The output pathname contains a logical name longer than 12 characters, contains unmatched colons, contains invalid characters, or zero characters.

- <pathname>, logical name does not exist

  The output pathname contains a logical name that does not exist.

- <pathname>, <exception value>:<exception mnemonic>

  An exceptional condition occurred when CGETOUTPUT$CONNECTION attempted to obtain the output connection. The <exception value> and <exception mnemonic> portions of the message indicate the exception code encountered. Refer to "Exception Codes" in this call description and to the *Extended iRMX II Extended I/O System User's Guide*.

Exception Codes

E$OK	0000H	No exceptional conditions were encountered.

E$ALREADY$ATTACHED	0038H	The Extended I/O System was unable to attach the device containing the file because the Basic I/O System has already attached the device.
E$CONTEXT	0005H	The calling task's job was not created by the Human Interface.
EDEVDETACHING	0039H	The device referred to by the path$name$p parameter was in the process of being detached.
E$DEVFD	0022H	The call attempted the physical attachment of a device that had formerly been only logically attached. In the process, the call found that the device and the device driver specified in the logical attachment were incompatible.
E$EXIST	0006H	The connection parameter for the device containing that file is not a token for an existing object.
E$FACCESS	0026H	At least one of the following is true: • The default user for the calling task's job did not have update access to an existing file and/or add-entry access to the parent directory. • The TO or OVER preposition was specified and the default user for the calling task's job did not have the ability to truncate the file.
E$FNEXIST	0021H	At least one of the following is true: • The target file does not exist or is marked for deletion. • While attaching the file pointed to by the path$name$p parameter, the Extended I/O System attempted the physical attachment of the device as a named device. It could not complete this process because the device specified when the logical attachment was made was not defined during configuration.

E$FTYPE	0027H	The path pointed to by the path$name$p parameter contained a file name that should have been the name of a directory, but is not. (Except for the last component, each file in a pathname must be a named directory.)
E$IFDR	002FH	The call requested information about the specified file, but the request was an invalid file driver request.
E$ILLVOL	002DH	The call attempted the physical attachment of the specified device as a named device. This device had formerly been only logically attached. The call found that the volume did not contain named files. This prevented the call from completing physical attachment because the named file driver was requested during logical attachment.
E$INVALID$FNODE	003DH	The fnode for the specified file is invalid, so the file must be deleted.
EIOHARD	0052H	While attempting to access the file specified in the path$name$p parameter, the call detected a hard I/O error. A retry is probably useless.
EIOMEM	0042H	While attempting to create a connection, this call needed memory from the Basic I/O System's memory pool. However, the Basic I/O System job does not currently have a block of memory large enough to allow this call to run to completion.
EIONOT$READY	0053H	While attempting to access the file specified in the path$name$p parameter, the call detected that the device was off-line. Operator intervention is required. C$FORMAT$EXCEPTION returns the value EIONOT$READY for this code.
EIOSOFT	0051H	While attempting to access the file specified in the path$name$p parameter, the call detected a soft I/O error. It tried the operation again but was unsuccessful. Another try might be successful.

EIOUNCLASS	0050H	An unknown type of I/O error occurred while this call tried to access the file given in the path$name$p parameter.
EIOWRPROT	0054H	While attempting to obtain an input connection to the file specified in the path$name$p parameter, this call found that the volume containing the file is write-protected.
E$LIMIT	0004H	At least one of the following is true: • The calling task's job or the job's default user object is already involved in 255 (decimal) I/O operations. • The calling task's job was not created by the Human Interface. • The calling task's job has reached its object limit. (Refer to the *Extended iRMX II Extended I/O System User's Guide* for more information about I/O jobs.)
ELOGNAME$NEXIST	0045H	The specified pathname contains an explicit logical name. The call was unable to find this name in the object directory of the local job, the global job, or the root job.
ELOGNAME$SYNTAX	0040H	The pathname pointed to by the path$name$p parameter contains a logical name. However, the logical name contains unmatched colons, is longer than 12 characters, contains invalid characters, or contains zero characters.
E$MEDIA	0044H	The specified device was off-line.
E$MEM	0002H	The memory available to the calling task's job is not sufficient to complete the call.
E$NOPREFIX	8022H	The calling task's job does not have a valid default prefix.
ENOTLOG$NAME	8040H	The logical name specified by the path$name$p parameter does not refer to a file or device connection.
E$NOUSER	8021H	The calling task's job does not have a valid default user object.

E$PARAM	8004H	The system call forced the Extended I/O System to attempt the physical attachment of the device referenced by the path$name$p parameter. The device had formerly been only logically attached. In the process, the Extended I/O System found that the logical attachment referred to a file driver (named, physical, or stream) that is not configured into your system, so the physical attachment is not possible.
E$PATHNAME$SYNTAX	003EH	The specified pathname contains invalid characters.
E$PREPOSITION	0087H	One of the following is true: • The command line contained an invalid preposition value (a value greater than 3). • The command line contained a zero as the preposition value. This indicated that the same preposition was to be used as in the last call to CGETOUTPUT$CONNECTION. However, this is the first call to CGETOUTPUT$CONNECTION.
E$SHARE	0028H	The new connection cannot be opened for writing.
E$SPACE	0029H	One of the following is true: • The volume is full. • The volume already contains the maximum number of files.
E$STREAM$SPECIAL	003CH	The Extended I/O System issued an invalid stream file request when an attempt to attach a stream file failed.

CGETOUTPUT$PATHNAME, a command parsing call, gets a pathname from the list of output pathnames in the parsing buffer.

```
preposition = RQ$C$GET$OUTPUT$PATHNAME(path$name$p, path$name$max,
                                       default$output$p, except$ptr);
```

Input Parameters

path$name$max — A WORD that specifies the maximum length in bytes of the string pointed to by the path$name$p parameter. The maximum length that you can specify is 256 bytes (255 characters for the pathname and one byte for the count).

default$output$p — A POINTER to a STRING containing the command's default standard output. If the first invocation of this system call does not encounter a TO/OVER/AFTER preposition, the text of this parameter will be used as though it had appeared in the command line. The text must specify TO, OVER, or AFTER for the output mode. Examples: TO :CO: or TO :LP:.

Output Parameters

preposition — A BYTE describing the preposition type that CGETOUTPUT$PATHNAME encountered. You can pass this value to CGETOUTPUT$CONNECTION when obtaining an output connection to the file. The value will be one of the following:

Value	Meaning
1	TO
2	OVER
3	AFTER

path$name$p — A POINTER to a buffer that receives a STRING. (The next pathname in the pathname list.)

except$ptr — A POINTER to a WORD in which the Human Interface returns a condition code.

Description

You should not call CGETOUTPUT$PATHNAME before first calling CGETINPUT$PATHNAME.

CGETOUTPUT$PATHNAME

The first call to CGETOUTPUT$PATHNAME retrieves the preposition (TO/OVER/AFTER) and the entire output pathname list; it then moves the parsing pointer to the next parameter. If the parsing buffer does not contain a preposition and pathname list, CGETOUTPUT$PATHNAME uses the default pointed to by the default$output$p parameter (and does not move the parsing pointer). After retrieving the pathname list, CGETOUTPUT$PATHNAME stores it in an internal buffer, returns the first pathname in the string pointed to by the path$name$p parameter, and returns the preposition in the preposition parameter. Succeeding calls to CGETOUTPUT$PATHNAME return additional pathnames from the output pathname list (as well as the preposition), but they do not move the parsing pointer. CGETOUTPUT$PATHNAME denotes the end of the pathname list by returning a zero-length string in the STRING pointed to by path$name$p.

CGETOUTPUT$PATHNAME accepts characters with a wild-card as the last component of a pathname. It generates each output pathname based on this pathname and wild-card, the corresponding pathname and wild-card that was input to CGETINPUT$PATHNAME, and the most recent input pathname returned by CGETINPUT$PATHNAME.

The pathname returned by CGETOUTPUT$PATHNAME can be used for any purpose. However, it is most often used in a call to CGETOUTPUT$CONNECTION to obtain a connection to the file. In such a case, CGETOUTPUT$CONNECTION processes the TO/OVER/AFTER preposition. If the pathname is used as input to a system call other than CGETOUTPUT$CONNECTION, the interpretation of the TO/OVER/AFTER preposition is the user's responsibility.

Exception Codes

E$OK	0000H	No exceptional conditions were encountered.
E$CONTEXT	0005H	The calling task's job was not created by the Human Interface.
E$DEFAULT$SO	8083H	The default output string pointed to by default$output$p contained an invalid preposition or pathname.

E$LIMIT	0004H	At least one of the following is true: • The calling task's job has already reached its object limit. • The calling task's job was not created by the Human Interface. • The calling task's job or the job's default user object is already involved in 255 (decimal) I/O operations.
E$MEM	0002H	The memory available to the calling task's job is not sufficient to complete the call.
E$PATHNAME$SYNTAX	003EH	The specified pathname contains invalid characters.
E$STRING	8084H	The pathname to be returned exceeds the length limit of 255 characters.
E$STRING$BUFFER	0081H	The buffer pointed to by the path$name$p parameter was not large enough for the pathname to be returned.
E$UNMATCHED$LISTS	008BH	The numbers of files in the input and output lists are not the same.
E$WILDCARD	0086H	The output pathname contains an invalid wild-card specification.

GET$PARAMETER, a command parsing call, gets a parameter from the parsing buffer.

```
more = RQ$C$GET$PARAMETER(name$p, name$max, value$p, value$max,
                          index$p, predict$list$p, except$ptr);
```

Input Parameters

name$max	A WORD that specifies the maximum length in bytes of the string pointed to by the name$p parameter. The maximum length is 256 bytes (255 characters for the name and one byte for the count).
value$max	A WORD that specifies the maximum length in bytes of the string pointed to by the value$p parameter. The maximum length is 65535 decimal bytes.
predict$list$p	A POINTER to a STRING$TABLE, as described in Appendix C of the *Extended iRMX II Human Interface User's Guide*, that specifies the values that this system call accepts as prepositions. The predict$list$p POINTER should be NIL if you do not intend to retrieve parameters that use prepositions.

Output Parameters

more	A BYTE value that indicates whether or not the current call to CGETPARAMETER returned a parameter. A value of 00H indicates that there are no more parameters (and that no parameter was returned); a value of 0FFH indicates that a parameter was returned.
name$p	A POINTER to a buffer that receives the keyword portion of the parameter. If this parameter does not contain a keyword portion, the Human Interface returns a null (zero-length) string.
value$p	A POINTER to a buffer used to store a STRING$TABLE, as described in Appendix C of the *Extended iRMX II Human Interface User's Guide*, that receives the value portion of the parameter. If the value portion contains a list of values separated by commas, the Human Interface returns the values to the string table one value per string.

index$p	A POINTER to a BYTE that receives the index to the list of prepositions pointed to by predict$list$p. This index identifies the name$p keyword as a preposition and identifies it out of the list of possible prepositions. If the predict$list$p list is empty, or if the keyword name is not contained in the predict$list$p list, the system call returns a value of zero for the index. That is, the index will be non-zero only if a keyword exists and it is one of the prepositions in the predict$list$p list.
except$ptr	A POINTER to a WORD in which the Human Interface returns a condition code.

Description

CGETPARAMETER retrieves one parameter from the parsing buffer and moves the parsing pointer to the next parameter. The parameter can be one of the following:

- keyword/value-list parameter using parentheses
- keyword/value-list parameter using an equal sign
- keyword/value-list parameter with the keyword as a preposition
- value-list without a keyword

A description of the types, format, and syntax of acceptable parameters is provided in the *Extended iRMX II Human Interface User's Guide*.

CGETPARAMETER places the keyword portion of the parameter in the string pointed to by name$p; it places the keyword list in the string table pointed to by value$p.

Without input from you, CGETPARAMETER cannot determine whether groups of characters separated by spaces are separate parameters or a single parameter that uses a preposition. CGETPARAMETER uses the list of prepositions that you supply in the string table pointed to by predict$list$p to determine the prepositions that can appear. When CGETPARAMETER retrieves a parameter, it obtains, from the parsing buffer, the next group of characters that are separated by spaces. These characters are checked against those in the predict$list$p list. If the characters match one of the values in the list, CGETPARAMETER realizes that the characters represent a preposition and not an entire parameter; it then obtains the next group of characters separated by spaces as the value portion of the parameter.

Exception Codes

E$OK	0000H	No exceptional conditions were encountered.

E$CONTEXT	0005H	The calling task's job was not an I/O job. (Refer to the *Extended iRMX II Extended I/O System User's Guide* for information about I/O jobs.)
E$CONTINUED	0083H	The call found a continuation character in the parse buffer. Command lines should not contain continuation characters.
E$LIMIT	0004H	At least one of the following is true: • The calling task's job has already reached its object limit. • The calling task's job was not an I/O job. (Refer to the *Extended iRMX II Extended I/O System User's Guide* for information about I/O jobs.)
E$LIST	0085H	At least one of the following is true: • The parameter contains an unmatched parenthesis. • A value in the value list is missing or an improper value was entered. Examples of both these conditions are

Value	Comments
A,B,	No value following second comma.
A,B=C,D	The equal sign can not be used unless it is between quotes: 'B=C' is valid.
A,B(C,E),F	The parentheses can not be used in a value unless it is between quotes or set off by commas. A,B,(C,E),F is valid.

E$LITERAL	0080H	The call found a literal (quoted string) in the parsing buffer with no closing quote. This condition should not occur in the command line buffer.
E$MEM	0002H	The memory available to the calling task's job is not sufficient to complete the call.

E$PARAM	8004H	The predict$list$p parameter pointed to a string table, but the index$p parameter was set to zero (0).
E$PARSE$TABLES	8080H	The call found an error in an internal table used by the Human Interface.
E$SEPARATOR	0082H	The call found an invalid command separator in the parsing buffer. This condition should not occur in the command line buffer. The following is a list of invalid command separators: >>, <<, ||, |, [, and].
E$STRING	8084H	The string to be returned as the parameter name or one of the parameter values exceeds the length limit of 255 characters.
E$STRING$BUFFER	0081H	The string to be returned as the parameter name or one of the parameter values exceeds the buffer size provided in the call.

C$SEND$COMMAND, a command processing call, sends command lines to a command connection created by C$CREATE$COMMAND$CONNECTION and, when the command is complete, invokes the command.

```
CALL RQ$C$SEND$COMMAND(command$conn, line$p, command$except$ptr,
                       except$ptr);
```

Input Parameters

command$conn	A TOKEN for the command connection that receives the command line.
line$p	A POINTER to a buffer used to store a STRING containing a command line to execute.

Output Parameters

command$except$ptr	A POINTER to a WORD in which the Human Interface returns a condition code indicating the status of the invoked command. This parameter is undefined if an exceptional condition code is returned in the WORD pointed to by except$ptr.
except$ptr	A POINTER to a WORD in which the Human Interface returns a condition code indicating the status of the C$SEND$COMMAND system call.

Description

You can use this system call when you want to invoke a command programmatically instead of interactively. It stores a command line in the command connection created by the C$CREATE$COMMAND$CONNECTION call, concatenates the command line with any others already stored there, and (if the command invocation is complete) invokes the command. The command can be any standard Human Interface command (as described in the *Operator's Guide To The Extended iRMX II Human Interface*) or a command that you create.

As described in greater detail in the *Extended iRMX II Universal Development Interface User's Guide*, a command invocation can contain several continuation marks. The continuation mark (&) indicates that the command line is continued on the next line. If the command line sent by C$SEND$COMMAND is continued on another line (that is, contains a continuation mark), the Human Interface returns an E$CONTINUED exception code and does not invoke the command. You can then call C$SEND$COMMAND any number of times to send the continuation lines.

C$SEND$COMMAND concatenates the original command line and all continuation lines into a single command line in the command connection. It removes all continuation marks and comments from this command line.

When the command invocation is complete (that is, the line sent by C$SEND$COMMAND does not contain a continuation mark), the Human Interface parses the command pathname from the command line. If no exception conditions halt the process at this point, the Human Interface requests the Application Loader to load and execute the command.

An Application Loader call creates an I/O job for the command, and validates the header, group definition and segment definition records of the command's object file. Refer to the *iAPX 286 Utilities User's Guide For Extended iRMX II Systems* for explanations of segments, groups and object file formats.

C$SEND$COMMAND returns two condition codes: one for the C$SEND$COMMAND call and one for the invoked command. The word pointed to by the except$ptr parameter returns the C$SEND$COMMAND conditions, as described under the "Exception Codes" heading in this command description. The WORD pointed to by the command$except$ptr returns the invoked command's condition codes; the values returned depend on the command invoked. The E$CONTROL$C exception code can be returned at either place.

NOTE

When a C$SEND$COMMAND call is made, the Human Interface sets the CONTROL-C semaphore to the default Human Interface CONTROL-C handler. If you previously set the CONTROL-C handler, it must be set again after making this call. For more information see the *Extended iRMX II Human Interface User's Guide*.

Exception Codes

E$OK	0000H	No exceptional conditions were encountered.

E$ALREADY$ATTACHED	0038H	The Extended I/O System was unable to attach the device containing the object file because the Basic I/O System has already attached the device.
EBADGROUP	0061H	The object file represented by the command's pathname contained an invalid group definition record.
EBADHEADER	0062H	The object file represented by the command's pathname does not begin with a header record for a loadable object module.
EBADSEGDEF	0063H	The object file represented by the command's pathname contains an invalid segment definition record.
E$CHECKSUM	0064H	At least one record of the object file represented by the command's pathname contains a checksum error. This situation could occur if the CHECKSUM amount calculated during the read operation did not match the CHECKSUM field of the record being read.
E$CONTEXT	0005H	The calling task's job was not created by the Human Interface.
E$CONTINUED	0083H	The operating system detected a continuation character while scanning the command line pointed to by the line$p parameter. This condition should occur if the command line is to continue on the next line.
EDEVDETACHING	0039H	The device containing the object file was in the process of being detached.
E$DEVFD	0022H	The Extended I/O System attempted the physical attachment of a device that had formerly been only logically attached. In the process, the Extended I/O System found that the device and the device driver specified in the logical attachment were incompatible.
E$EOF	0065H	The Application Loader encountered an unexpected end of file on the object file represented by the command's pathname.

E$EXIST	0006H	At least one of the following is true: • The call detached the device containing the object file before completing the loading operation. • The command$conn parameter is not a TOKEN for a command connection.
E$FACCESS	0026H	The default user for the calling task's job does not have read access to the object file.
E$FLUSHING	002CH	The device containing the object file was being detached.
E$FNEXIST	0021H	At least one of the following is true: • The file in the command's pathname is either marked for deletion or does not exist. • While attaching the file specified in the line$p parameter, the Extended I/O System attempted the physical attachment of the device as a named device. It could not complete this process because the device specified when the logical attachment was made was not defined during configuration.
E$FTYPE	0027H	The path pointed to by the path$name$p parameter contained a component name that should have been the name of a directory, but is not. (Except for the last file, each file in a pathname must be a named directory.)
E$ILLVOL	002DH	The call attempted the physical attachment of the specified device as a named device. This device had formerly been only logically attached. The call found that the volume did not contain named files. This prevented the call from completing physical attachment because the named file driver was requested during logical attachment.
E$INVALID$FNODE	003DH	The fnode for the specified file is invalid, so the file must be deleted.
EIOHARD	0052H	While attempting to access the object file, this call detected a hard I/O error.

EIOMEM	0042H	The Basic I/O System does not currently have enough memory to allow the Human Interface to create the connection necessary to allow this call to run to completion.
EIONOT$READY	0053H	While attempting to access the object file, this call found that the device was off-line. Operator intervention is required. C$FORMAT$EXCEPTION returns the value EIONOT$READY when given this code.
EIOSOFT	0051H	While attempting to access the object file, this call detected a soft I/O error. It tried again, but was not successful. Another try might be successful.
EIOUNCLASS	0050H	An unknown type of I/O error occurred while this call tried to access the object file.
E$LIMIT	0004H	At least one of the following is true: • The calling task's job has already reached its object limit. • The calling task's job , or the job's default user object, is already involved in 255 (decimal) I/O operations. • The new I/O job, or its default user, is already involved in 255 (decimal) I/O operations. • The calling task's job was not created by the Human Interface. (See to the *Extended iRMX II Extended I/O System User's Guide* for information.)
E$LITERAL	0080H	The call found a literal (quoted string) with no closing quote while scanning the contents of the command line pointed to by the line$p parameter.
ELOGNAME$NEXIST	0045H	The command's pathname contains an explicit logical name, but the call was unable to find this name in the object directory of the local job, the global job, or the root job.

ELOGNAME$SYNTAX	0040H	The pathname pointed to by the path$name$p parameter contains a logical name. However, the logical name contains an unmatched colon, is longer than 12 characters, has zero (0) characters, or contains invalid characters.
E$MEDIA	0044H	The device containing the object file was off-line.
E$MEM	0002H	The memory available to the calling task's job, the new I/O job, or the Basic I/O System job is not sufficient to complete the call.
ENOLOADER$MEM	0067H	At least one of the following is true: • The memory pool of the newly created I/O job does not currently have a block of memory large enough to allow the Application Loader to run. • The memory pool of the Basic I/O System's job does not currently have a block of memory large enough to allow the Application Loader to run.
E$NOPREFIX	8022H	The calling task's job does not have a valid default prefix.
ENOSTART	006CH	The object file represented by the command pathname does not specify the entry point for the program being loaded.
ENOTCONNECTION	8042H	The default$ci or default$co parameter is a token for an object that is not a file connection.
ENOTLOG$NAME	8040H	The command pathname contains a logical name. The logical name of an object that is neither a device connection nor a file connection.
E$NOUSER	8021H	The calling task's job does not have a valid default user.

E$PARAM	8004H	The Extended I/O System attempted the physical attachment of a device containing the object file. This device had formerly been only logically attached. While attempting this, the Extended I/O System found that the logical attachment referred to a file driver (named, physical, or stream) that is not configured into your system. Hence, the physical attachment is not possible.
E$PARSE$TABLES	8080H	The call found an error in an internal table.
E$PATHNAME$SYNTAX	003EH	The command's pathname contains invalid characters.
ERECFORMAT	0069H	At least one record in the object file contains a record format error.
ERECLENGTH	006AH	The object file contains a record that is longer than the Loader's maximum record length. The Application Loader's maximum record length is a parameter specified during the configuration of the Loader. (Refer to the *Extended iRMX II Interactive Configuration Utility Reference Manual* for details.)
ERECTYPE	006BH	At least one of the following is true: • At least one record in the file being loaded is of a type that the Application Loader cannot process. • The Application Loader has encountered records in a sequence that it cannot process.
ESEGBOUNDS	0070H	The Application Loader created multiple segments in which to load information. One of the data records in the object file specified a load address outside of the created segments.
E$SEPARATOR	0082H	The call found an invalid separator while scanning the command line. The following is a list of the invalid command separators: > <, < >, | |, |, [, and].
E$STRING	8084H	The size of the command's pathname exceeds the length limit of 255 (decimal) characters.

E$STRING$BUFFER	0081H	The size of the command's pathname exceeds the size of the command name buffer specified during the configuration of the Human Interface.
E$TIME	0001H	The calling task's job was not created by the Human Interface.
E$TYPE	8002H	The command$conn parameter is a token for an object that is not a command connection.

C$SEND$CO$RESPONSE, a message processing call, sends a message to :CO: and reads a response from :CI:.

```
CALL RQ$C$SEND$CO$RESPONSE(response$p, response$max, message$p,
                           except$ptr);
```

Input Parameters

response$max	A WORD whose value specifies the maximum length in bytes of the string pointed to by the response$p parameter. The value in response$max must equal the length of the string plus one (stringlength + 1). If response$max is zero or one, no response from :CI: will be requested; control returns to the calling task immediately.
message$p	A POINTER to a STRING containing the message to be sent to :CO:. If NIL, no message is sent.

Output Parameters

response$p	A POINTER to a buffer that receives the operator's response from :CI:.
except$ptr	A POINTER to a WORD in which the Human Interface returns a condition code.

Description

When used with all its features, C$SEND$CO$RESPONSE sends the string pointed to by message$p to :CO: and waits for a response from :CI:. It places this response in the string pointed to by response$p. However, if message$p is NIL, C$SEND$CO$RESPONSE omits sending the message to :CO:; if either response$max or response$p is NIL, it does not wait for a response from :CI:. Therefore, the operations performed by C$SEND$CO$RESPONSE depend on the values of the message$p and response$max parameters, as follows:

message$p	response$max	Action
NIL	zero	Perform no I/O
NIL	non-zero	Send no message, wait for input
NOT NIL	non-zero	Send message, wait for input
NOT NIL	zero	Send message, don't wait

If C$SEND$CO$RESPONSE requests a response from :CI:, output from other tasks can be displayed at :CO: while the system waits for a response from :CI:.

The difference between the C$SEND$CO$RESPONSE and C$SENDEORESPONSE calls is that C$SEND$EO$RESPONSE always sends messages to and receives messages from the operator's terminal; input and output cannot be redirected to another device. In contrast, C$SEND$CO$RESPONSE sends messages to :CO: and receives messages from :CI:; therefore, programs such as SUBMIT can redirect this input and output.

Exception Codes

E$OK	0000H	No exceptional conditions were encountered.
E$CONTEXT	0005H	The calling task's job was not created by the Human Interface.
E$CONN$OPEN	0035H	At least one of the following is true: • The connection to :CI: was not open for reading or the connection to :CO: was not open for writing. • The connection to :CI: or :CO: was not open. • The connection to :CI: or :CO: was opened with A$OPEN rather than S$OPEN.
E$EXIST	0006H	The token value for :CI: or :CO: is not a token for an existing object.
E$FLUSHING	002CH	The device containing the :CI: and :CO: files was being detached.
EIOHARD	0052H	While attempting to access the :CI: or :CO: file, the operating system detected a hard I/O error.
EIONOT$READY	0053H	While attempting to access the :CI: or :CO: file, this call found that the device was off-line. Operator intervention is required. C$FORMAT$EXCEPTION returns the value EIONOT$READY for this code.
EIOSOFT	0051H	While attempting to access the :CI: or :CO: file, this call detected a soft I/O error. It tried again, but was unsuccessful. Another try might be successful.

EIOUNCLASS	0050H	An unknown type of I/O error occurred while this call tried to access the :CI: or :CO: file.
EIOWRPROT	0054H	While attempting to obtain a connection to the :CO: file, this call found that the volume containing the file is write-protected.
E$LIMIT	0004H	At least one of the following is true: • The calling task's job has already reached its object limit. • The calling task's job, or the job's default user object, is already involved in 255 (decimal) I/O operations. • The calling task's job was not created by the Human Interface.
E$MEM	0002H	The memory available to the calling task's job is not sufficient to complete the call.
ENOTCONNECTION	8042H	The call obtained a token for an object that should have been a connection to :CI: or :CO:, but was not a file connection.
E$PARAM	8004H	The call attempted to write beyond the end of a physical file.
E$SPACE	0029H	One of the following is true: • The output volume is full. The call attempted to write beyond the end of a physical file.
E$STREAM$SPECIAL	003CH	When attempting to read or write to :CI: or :CO:, the Extended I/O System issued an invalid stream file request.
E$SUPPORT	0023H	The connection to :CI: or :CO: was not created by this job.
E$TIME	0001H	The calling task's job was not created by the Human Interface.

C$SEND$EO$RESPONSE, a message processing call, sends a message to and reads a response from the operator's terminal.

```
CALL RQ$C$SEND$EO$RESPONSE(response$p, response$max, message$p,
                           except$ptr);
```

Input Parameters

response$max — A WORD that specifies the maximum length in bytes of the string pointed to by the response$p parameter. The value in response$max must equal the length of the string plus one (stringlength + 1). If response$max is zero or one, no response from the operator's terminal will be requested; control returns to the calling task immediately.

message$p — A POINTER to a buffer containing the message to be sent to the operator's terminal. If NIL, no message is sent.

Output Parameters

response$p — A POINTER to a STRING that receives the operator's response from the terminal.

except$ptr — A POINTER to a WORD in which the Human Interface returns a condition code.

Description

When used with all its features, C$SEND$EO$RESPONSE sends the string pointed to by message$p to the operator's terminal and waits for a response from the operator. It places this response in the string pointed to by response$p. However, if message$p is NIL, C$SEND$EO$RESPONSE omits sending the message to the operator; if either response$max is zero or response$p is NIL, it does not wait for a response. Therefore, the operations performed by C$SEND$EO$RESPONSE depend on the values of the message$p and response$max parameters, as follows:

message$p	response$max	Action
NIL	zero	Perform no I/O
NIL	non-zero	Send no message, wait for input
NOT NIL	non-zero	Send message, wait for input
NOT NIL	zero	Send message, don't wait

C$SEND$EO$RESPONSE

If C$SEND$EO$RESPONSE requests a response from the terminal, no other output can be displayed at the terminal until C$SEND$EO$RESPONSE receives a line terminator from the operator. However, the operator can choose to ignore the displayed message by entering a line terminator only.

The main distinction between the C$SEND$CO$RESPONSE and C$SENDEORESPONSE calls is that C$SEND$EO$RESPONSE always sends messages to and receives messages from the operator's terminal; input and output cannot be redirected to another device. In contrast, C$SEND$CO$RESPONSE sends messages to :CO: and receives messages from :CI:; therefore, programs such as SUBMIT can redirect this input and output.

Exception Codes

E$OK	0000H	No exceptional conditions were encountered.
E$CONN$OPEN	0035H	At least one of the following is true: • Either, the connection to the operator's terminal was not open for reading or it was not open for writing. • The connection to the operator's terminal was not open. • The connection to the operator's terminal was opened with A$OPEN rather than S$OPEN.
E$CONTEXT	0005H	The calling task's job was not created by the Human Interface.
E$ERROR$OUTPUT	8085H	The call to SENDEORESPONSE was attempted through an invalid method.
E$EXIST	0006H	The token values for the operator's terminal are not for existing objects.
E$FLUSHING	002CH	The operator's terminal was being detached.
EIONOT$READY	0053H	While attempting to access the terminal, this call found that the device was off-line. Operator intervention is required. C$FORMAT$EXCEPTION returns the value EIONOT$READY when given this code.

E$LIMIT	0004H	At least one of the following is true: • The calling task's job has already reached its object limit. • The calling task's job or the job's default user object is already involved in 255 (decimal) I/O operations. • The calling task's job was not created by the Human Interface.
E$MEM	0002H	The memory pool of the calling task's job does not currently have a block of memory large enough to allow this system call to run to completion.
ENOTCONNECTION	8042H	The call obtained a token for an object that should have been a connection to the operator's terminal, but was not a file connection.
E$PARAM	8004H	The call attempted to write beyond the end of a physical file.
E$STREAM$SPECIAL	003CH	When attempting to read or write to the operator's terminal, the Extended I/O System issued an invalid stream file request.
E$SUPPORT	0023H	The connection to the terminal was not created by this job.
E$TIME	0001H	The calling task's job was not created by the Human Interface.

CSETCONTROL$C, a program control call, changes a calling task's CONTROL-C exchange to the semaphore specified by the first parameter in the CSETCONTROL$C call.

```
CALL RQ$C$SET$CONTROL$C(control$c$semaphore, except$ptr);
```

Input Parameter

control$c$semaphore	A TOKEN for a user-created semaphore that will receive units when a CONTROL-C is typed on the console keyboard.

NOTE

When a C$SEND$COMMAND call is made, the Human Interface sets the CONTROL-C semaphore to the default Human Interface CONTROL-C handler. If you previously set the CONTROL-C handler, it must be set again after making this call. For more information see the *Extended iRMX II Human Interface User's Guide*.

Output Parameter

except$ptr	A POINTER to a WORD in which the Human Interface returns a condition code.

Description

This call lets you change the default response to a CONTROL-C entry to a response that meets the needs of your task. (The Human Interface's default CONTROL-C action is to delete the acting job--for example, any Human Interface command.)

One unit will be sent to the semaphore each time a CONTROL-C is typed. Any units sent to the semaphore that exceed the maximum number specified during system configuration will be ignored.

A job running in background mode cannot set CONTROL-C.

Exception Codes

E$OK	0000H	No exceptional conditions were encountered.

E$CONTEXT	0005H	The calling task's job was not an I/O job. (Refer to the *Extended iRMX II Extended I/O System User's Guide* for information about I/O jobs.)
E$LIMIT	0004H	At least one of the following is true: • The calling task's job has already reached its limit. • The calling task's job was not created by the Human Interface. • The calling task's job or the job's default user object is already involved in 255 (decimal) I/O operations.
E$TYPE	8002H	The TOKEN given in the parameter control$c$semaphore is not a TOKEN for a semaphore.

CSETPARSE$BUFFER, a command parsing call, permits parsing the contents of a buffer other than the command line buffer whenever the parsing system calls are used.

```
offset = RQ$C$SET$PARSE$BUFFER(buff$p, buff$max, except$ptr);
```

Input Parameters

buff$p	A POINTER to a buffer containing a STRING containing the text to be parsed. If the buff$p is NIL, the buffer used for parsing reverts to the command line buffer and the buff$max parameter is ignored.
buff$max	A WORD that specifies the length in bytes of the STRING pointed to by the buff$p parameter.

Output Parameters

offset	A WORD in which the Human Interface places the byte offset from the start of the parsing buffer of the last byte parsed in the previous parsing buffer.
except$ptr	A POINTER to a WORD in which the Human Interface returns a condition code.

Description

CSETPARSE$BUFFER allows you to parse buffers other than the command line. You can change buffers at will; you can also revert to the command line parsing buffer by calling CSETPARSE$BUFFER with buff$p = NIL. However, only one parsing buffer per job can be active at any given time.

When called, CSETPARSE$BUFFER sets the parsing pointer to the beginning of the specified buffer. However, it also returns a value (in the offset parameter) that identifies the last byte parsed in the previous parsing buffer. This gives you the ability, when switching back to the previous buffer, of positioning the parsing pointer to its previous position with successive calls to CGETCHAR.

Note that CSETPARSE$BUFFER does not affect the buffer from which CGETINPUT$PATHNAME and C$GET$OUTPUT$PATHNAME retrieve pathnames. These system calls always obtain their pathnames from the command line.

Exception Codes

E$OK	0000H	No exceptional conditions were encountered.
E$CONTEXT	0005H	The calling task's job was not created by the Human Interface. (Refer to the *Extended iRMX II Extended I/O System User's Guide* for information.)
E$LIMIT	0004H	At least one of the following is true: • The calling task's job has already reached its object limit. • The calling task's job was not created by the Human Interface.
E$MEM	0002H	The memory available to the calling task's job is not sufficient to complete the call.

INDEX

C

E

F

S

V

intel

EXTENDED iRMX® II UDI SYSTEM CALLS REFERENCE MANUAL

Intel Corporation
3065 Bowers Avenue
Santa Clara, California 95051

PREFACE

This manual documents the system calls of the Universal Development Interface, a subsystem of the extended iRMX II Operating System. The information provided in this manual is intended as a reference to the system calls and provides detailed descriptions of each call.

READER LEVEL

This manual is intended for programmers who are familiar with the concepts and terminology introduced in the *Extended iRMX II Nucleus User's Guide* and with the PL/M-286 programming language.

CONVENTIONS

System call names appear as headings on the outside upper corner of each page. The first appearance of each system call name is printed in ink; subsequent appearances are in black.

Throughout this manual, system calls are shown using a generic shorthand (such as ALLOCATE instead of DQ$ALLOCATE). This convention is used to allow easier alphabetic arrangement of the calls. The actual PL/M-286 external-procedure names must be used in all calling sequences.

You can also invoke the system calls from assembly language, but you must obey the PL/M-286 calling sequences when doing so. For more information on these calling sequences refer to the *Extended iRMX II Programming Techniques Reference Manual.*

Intel®

CONTENTS

UDI SYSTEM CALLS

1.1 INTRODUCTION

This manual describes the requirements and behavior of UDI system calls in the Extended iRMX II Operating System environment.

Table 1. Standard UDI Condition Codes and Their Meanings

Hex Value	Mnemonic	UDI Calls	Meaning
0000H	E$OK	All but DQ$EXIT	No exceptional conditions.
0002H	E$MEM	DQ$ALLOCATE DQ$ATTACH DQ$CREATE DQ$OPEN DQ$RESERVE$IO$- MEMORY DQ$MALLOCATE	Insufficient memory for the requested operation.
0020H	E$FEXIST	DQ$RENAME	The specified file exists.
0021H	E$FNEXIST	DQ$ATTACH DQ$DELETE DQ$RENAME DQ$CHANGE$ACCESS	The specified file does not exist.

(continued)

Table 1. Standard UDI Condition Codes and Their Meanings (continued)

Hex Value	Mnemonic	UDI Calls	Meaning
0023H	E$SUPPORT	DQ$ATTACH DQ$CHANGE$ACCESS DQ$CREATE DQ$DECODE$TIME DQ$FILE$INFO DQGETCONNECTION$- STATUS DQ$OPEN DQ$OVERLAY DQ$READ DQ$RENAME DQ$RESERVEIOMEMORY DQ$SEEK DQ$SPECIAL DQ$TRUNCATE DQ$WRITE	An unsupported operation was attempted.
0026H	E$FACCESS	DQ$CHANGE$ACCESS DQ$DELETE DQ$OPEN	Access to the specified file is denied.
0028H	E$SHARE	DQ$OPEN	The specified file may not be shared.
0029H	E$SPACE	DQ$CREATE DQ$WRITE	The operation attempted to add a directory entry to a full directory.
0081H	E$STRING- $BUFFER	DQGETARGUMENT DQ$CHANGE$EXTENSION	The string is over 45 characters long or the argument is over 80 characters long.

1.2 DESCRIPTIONS OF SYSTEM CALLS

This section describes the individual UDI calls in detail. Immediately preceding the detailed descriptions, the UDI Call Dictionary (Table 2) arranges the calls in functional groups, and lists the page numbers of the more detailed descriptions.

Every system call description contains the following information in this order:

- The name of the system call.
- A brief summary of the function of the call.
- The form of the call as it is invoked from a PL/M-286 program, with symbolic names for each parameter.
- Definition of input and output parameters.
- A complete explanation of the system call, including any information you will need to use it.
- Condition codes--a list of the error codes that can be incurred.

1.3 UDI SYSTEM CALLS DICTIONARY

Table 2. UDI System Calls Dictionary

UDI Call	Function Performed	Page
	PROGRAM CONTROL CALLS	
DQ$EXIT	Exits from the current application job.	22
DQ$OVERLAY	Causes the specified overlay to be loaded.	45
DQ$TRAP$CC	Captures control when CONTROL-C is typed.	61
	FILE-HANDLING CALLS	
DQ$ATTACH	Creates a connection to a specified file.	8
DQ$CHANGE$-ACCESS	Changes access rights associated with a file or directory.	10
DQ$CHANGE$-EXTENSION	Changes the extension of a file name in memory.	13
DQ$CLOSE	Closes the specified file connection.	15
DQ$CREATE	Creates a file for use by the application.	16
DQ$DELETE	Deletes a file.	20
DQ$DETACH	Closes a file and deletes its connection.	21
DQ$FILE$INFO	Returns data about a file connection	24
DQGETCON-NECTION$STATUS	Returns status of a file connection.	32
DQ$OPEN	Opens a file for a particular type of access.	42
DQ$READ	Reads the next sequence of bytes from a file.	47

Table 2. UDI System Calls Dictionary (continued)

UDI Call	Function Performed	Page
	FILE-HANDLING CALLS	
DQ$RENAME	Renames the specified file.	49
DQ$SEEK	Moves the current position pointer of a file.	53
DQ$SPECIAL	Se.s terminal line-edit/transparent mode.	55
DQ$TRUNCATE	Truncates a file to the specified length.	63
DQ$WRITE	Writes a sequence of bytes to a file.	64
	MEMORY MANAGEMENT CALLS	
DQ$ALLOCATE	Requests a memory segment of a specified size.	7
DQ$FREE	Returns a memory segment to the system.	28
DQGETMSIZE	Returns the size of the specified memory block.	35
DQGETSIZE	Returns the size of the specified segment.	36
DQ$MALLOCATE	Requests a logically contiguous memory segment of a specified size.	39
DQ$MFREE	Returns memory allocated by DQ$MALLOCATE to the Free Space Pool.	41
DQ$RESERVE$-IO$MEMORY	Requests memory to be set aside for overhead to be incurred by I/O operations.	51

Table 2. UDI System Calls Dictionary (continued)

UDI Call	Function Performed	Page
	EXCEPTION-HANDLING CALLS	
DQ$DECODE$-EXCEPTION	Converts an exception numeric code into its equivalent mnemonic.	17
DQGETEXCEPT-ION$HANDLER	Returns a POINTER to the address of the program currently being used to process errors.	34
DQ$TRAP$-EXCEPTION	Identifies a custom exception processing program for a particular type of error.	59
	UTILITY AND COMMAND PARSING	
DQ$DECODE$-TIME	Returns system time and date in both binary and ASCII-character format	18
DQGETARGUMENT	Returns an argument from a STRING.	29
DQGET-SYSTEM$ID	Returns the identity of the environment for the UDI.	37
DQGETTIME	Obsolete: included for compatibility.	38
DQ$SWITCH$-BUFFER	Selects a new buffer from which to process commands.	58

DQ$ALLOCATE requests a memory segment from the free memory pool.

```
seg$t = DQ$ALLOCATE (size, except$ptr);
```

Input Parameter

size — A WORD which,

- if not zero, contains the size, in bytes, of the requested segment.text deleted
- if zero, indicates that the size of the request is 65536 (64K) bytes.

Output Parameters

seg$t — A TOKEN, into which the operating system places the base address of the memory segment. If the request fails because the memory requested is not available, this value will be undefined and the system will return an E$MEM exception code.

except$ptr — A POINTER to a WORD where the system places the condition code.

Description

The DQ$ALLOCATE system call is used to request additional memory from the free space pool of the program. Tasks may use the additional memory for any desired purpose.

Condition Codes

E$OK	0000H	No exceptional conditions.
E$MEM	0002H	Insufficient memory to create a segment of the desired size.

In addition to the condition codes listed above, DQ$ALLOCATE can return the condition codes associated with the Nucleus system calls RQGETPOOL$ATTRIBUTES and RQ$CREATE$SEGMENT. See the *Extended iRMX II Nucleus System Calls Reference Manual* for details.

The DQ$ATTACH system call creates a connection to an existing file.

```
connection$t = DQ$ATTACH (path$ptr, except$ptr);
```

Input Parameter

path$ptr	A POINTER to a STRING containing the pathname of the file to be attached.

Output Parameters

connection$t	A TOKEN for the connection to the file.
except$ptr	A POINTER to a WORD where the system places the condition code.

Description

This system call allows a program to obtain a connection to any existing file. When the DQ$ATTACH call returns a connection, all existing connections to the file remain valid.

Your program can use the DQ$RESERVE$IO$MEMORY call to reserve memory that the UDI can use for its internal data structures when the program calls DQ$ATTACH and for buffers when the program calls DQ$OPEN. The advantage of reserving memory is that the memory is guaranteed to be available when needed. If memory is not reserved, a call to DQ$ATTACH might not be successful because of a memory shortage. See the description of DQ$RESERVE$IO$MEMORY later in this chapter for more information about reserving memory.

Condition Codes

E$OK	0000H	No exceptional conditions.
E$FNEXIST	0021H	The specified file does not exist.
E$MEM	0002H	Insufficient memory for the requested operation.
E$SUPPORT	0023H	An unsupported operation was attempted.

In addition to the condition codes listed above, DQ$ATTACH can return the exception codes associated with the Extended I/O System call RQSATTACH$FILE. See the *Extended iRMX II Extended I/O System Calls Reference Manual* for details.

The DQ$CHANGE$ACCESS enables you to change the access rights of the owner of a file (or directory), or the access rights of the WORLD user.

```
CALL DQ$CHANGE$ACCESS (path$ptr, user, access, except$ptr);
```

INPUT PARAMETERS

path$ptr — A POINTER to a STRING containing a pathname of the file.

user — A BYTE specifying the user whose access is to be changed.

Value	User
0	Owner of the file
1	WORLD (all users on the system)
2	GROUP (ignored by iRMX II Operating System
3-255	Reserved

If you specify a value of 3-255, an E$SUPPORT exception will be returned.

access — A BYTE specifying the type of access to be granted the user. This WORD is to be encoded as follows. (Bit 0 is the low-order bit.)

Bit	Meaning
0	User can delete the file or directory
1	Read (the file) or List (the directory)
2	Append (to the file) or Add entry (to the directory)
3	Update (read and write to the file) or Change Access (to the directory)
4	User can execute the file. Set to the value of bit one for compatibility with other operating systems.
5-7	Reserved. If you specify bits 5-7, an E$SUPPORT exception will be returned.

Output Parameter

except$ptr — A POINTER to a WORD where the system places the condition code.

Description

In the general extended iRMX II environment, every program is associated with a user object, usually referred to as the default user for the program. The default user consists of one or more user IDs. Each file has an associated collection of user ID-access mask pairs, where each mask defines the access rights the corresponding user ID has to the file. When the program calls DQ$CREATE to create a file or DQ$ATTACH to get another connection to a file, the resulting connection receives all access rights corresponding to user IDs that are both associated with the file and in the default user. The purpose of the DQ$CHANGE$ACCESS system call is to change, for a particular file, the access rights associated with a particular user ID. This has the effect of changing the access granted when the program makes subsequent calls to DQ$ATTACH to get further connections to the file.

In the UDI subset of the extended iRMX II environment, a default user has two IDs. One of them, called the owner ID, is associated with the program. The other, called the WORLD, is associated universally with all programs. DQ$CHANGE$ACCESS can change, for the file, the access mask of either the owner ID or the WORLD.

Changing the access rights for a user ID has no effect on connections already obtained by the program. However, all subsequently obtained connections reflect the changed access rights.

For more information about user IDs, default users, access masks, WORLD, access rights, owner IDs, and how connections are related to all of these entities, refer to the *Extended iRMX II Basic I/O System User's Guide*.

NOTE

DQ$CHANGE$ACCESS affects only connections made after the call is issued. It does not affect existing connections to the file.

Condition Codes

E$OK	000H	No exceptional conditions.
E$SUPPORT	0023H	The value specified for the user parameter is greater than two. You tried to set bits 5-7 of the access parameter.
E$FACCESS	0026H	Access to the specified file is denied.

In addition to the condition codes listed above, DQ$CHANGE$ACCESS can return the same condition codes as the Extended I/O System call RQSCHANGE$ACCESS. See the *Extended iRMX II Extended I/O System Calls Reference Manual* for details.

DQ$CHANGE$EXTENSION changes or adds the extension at the end of a file name stored in memory (not the file name on the mass storage volume).

```
CALL DQ$CHANGE$EXTENSION (path$ptr, extension$ptr, except$ptr);
```

INPUT PARAMETERS

path$ptr	A POINTER to a STRING containing a pathname of the file to be renamed.
extension$ptr	A POINTER to a series of three bytes containing the characters to be added to the pathname. This is not a STRING. You must include three bytes, even if some are blank.

Output Parameter

except$ptr	A POINTER to a WORD where the system places the condition code.

Description

This is a facility for editing strings that represent file names in memory. If the existing file name has an extension, DQ$CHANGE$EXTENSION replaces that extension with the specified three characters. Otherwise, DQ$CHANGE$EXTENSION adds the three characters as an extension.

For example, a compiler can use DQ$CHANGE$EXTENSION to edit a string containing the name, such as :AFD1:FILE.SRC, of a source file to the name, such as :AFD1:FILE.OBJ, of an object file, and then create the object file.

Note that extended iRMX II file names may contain multiple periods, but if they do, the extension, if any, consists of the characters following the last period. Note also that an extension may contain more than three characters, but any extension created or changed by DQ$CHANGE$EXTENSION has at most three (non-blank) characters.

The three-character extension may not contain delimiters recognized by DQGETARGUMENT but may contain trailing blanks. If the first character pointed to by extension$ptr is a space, DQ$CHANGE$EXTENSION deletes the existing extension including the period, if any, preceding the extension.

DQ$CHANGE$EXTENSION

Condition Codes

E$OK	000H	No exceptional conditions.
E$STRING$BUFFER	0081H	The filename is more than 14 characters.

DQ$CLOSE waits for completion of I/O operations (if any) taking place on the file, empties the output buffers, and frees all buffers associated with the connection.

```
CALL DQ$CLOSE (connection$t, except$ptr);
```

Input Parameter

connection$t	A TOKEN for a file connection that is currently open.

Output Parameter

except$ptr	A POINTER to a WORD where the system places the condition code.

Description

The DQ$CLOSE system call closes a connection that has been opened by the DQ$OPEN system call. It performs the following actions, in order:

1. Waits until all currently running I/O operations for the connection are completed.
2. Ensures that information, if any, in a partially filled output buffer is written to the file.
3. Releases all buffers associated with the connection.
4. Closes the connection. The connection is still valid, and can be re-opened if necessary.

Condition Codes

E$OK	000H	No exceptional conditions.

In addition to the condition code listed above, DQ$CLOSE can return the same condition codes associated with the Extended I/O System call RQSCLOSE. See the *Extended iRMX II Extended I/O System Calls Reference Manual* for details.

DQ$CREATE creates a new file and establishes a connection to the file.

```
connection$t = DQ$CREATE (path$ptr, except$ptr);
```

Input Parameter

path$ptr	A POINTER to a STRING containing a pathname for the file to be created.

Output Parameters

connection$t	A TOKEN for the connection to the file.
except$ptr	A POINTER to a WORD where the system places the condition code.

Description

This call creates a new file with the name you specify and returns a connection to it. If a file of the same name already exists, it is truncated to a length of zero and the data in it is destroyed.

To prevent accidentally destroying a file, call DQ$ATTACH before calling DQ$CREATE. If the file does not exist, DQ$ATTACH returns an E$FNEXIST exception code.

Condition Codes

E$OK	0000H	No exceptional conditions.
E$MEM	0002H	Insufficient memory remains to complete the call.
E$SPACE	0029H	Insufficient space exists on a direct-access device.
E$SUPPORT	0023H	An unsupported operation was attempted.

In addition to the condition codes listed above, DQ$CREATE can return the condition codes associated with the Extended I/O system calls RQSCREATE$FILE and RQ$S$DELETE$FILE. See the *Extended iRMX II Extended I/O System Calls Reference Manual* for details.

DQ$DECODE$EXCEPTION translates an exception code into its mnemonic.

```
CALL DQ$DECODE$EXCEPTION (exception$code, buff$ptr, except$ptr);
```

Input Parameter

exception$code	A WORD containing the numeric exception code that is to be translated.

Output Parameters

buff$ptr	A POINTER to a STRING (at least 81 bytes long) into which the system returns the mnemonic.
except$ptr	A POINTER to a WORD where the system places the condition code.

Description

Your program can call DQ$DECODE$EXCEPTION to exchange a numeric exception code for its hexadecimal equivalent followed by its mnemonic. For example, if you pass DQ$DECODE$EXCEPTION a value of 2 in the except$code parameter, the system returns the following string to the area pointed to by the buff$ptr parameter:

```
0002H:  E$MEM
```

The hexadecimal values and mnemonics for condition codes are listed in Table 1. This system call can decode any extended iRMX II exception code value. See the *Operator's Guide To The Extended iRMX II Human Interface* for more details.

Condition Codes

E$OK	0000H	No exceptional conditions.

In addition to the condition code listed above, DQ$DECODE$EXCEPTION can return the condition codes associated with the Human Interface system call, RQCFORMAT$EXCEPTION. See the *Extended iRMX II Human Interface System Calls Reference Manual* for details.

DQ$DECODE$TIME returns the current system time and date as ASCII date and time strings. You can also use DQ$DECODE$TIME to return the current time and date in binary format or as a decoded ASCII string.

```
CALL  DQ$DECODE$TIME (date$time$ptr, except$ptr);
```

Output Parameters

date$time$ptr — A POINTER to a structure of the following form:

```
DECLARE DT STRUCTURE(
                SYSTEM$TIME DWORD,
                DATE (8)    BYTE,
                TIME (8)    BYTE);
```

where

SYSTEM$TIME is an operating-system-dependent DWORD containing the current time and date. To get the current time and date, the value in SYSTEM$TIME must be zero when the DQ$DECODE$TIME call is issued. To decode a binary time value, the time value must be stored in SYSTEM$TIME before making the call. (See the following Description section for format information.)

SYSTEM$TIME receives the time as the number of seconds since midnight, January 1, 1978.

DATE receives the date portion of the time, in the form of ASCII characters.

TIME receives the time-of-day portion of the time, in the form of ASCII characters.

If the value in SYSTEM$TIME is not 0 when DQ$DECODE$TIME is called, DQ$DECODE$TIME accepts that value as the number of seconds since midnight, January 1, 1978, decodes the value, and returns it in the DATE and TIME fields.

except$ptr — A POINTER to a WORD where the system places the condition code.

Description

This system call returns the indicated date and time, each as a series of ASCII bytes. (Note that they are not STRINGs.)

DATE has the form MM/DD/YY for month, day, and year. The two slashes (/) are in the third and sixth bytes. For example, the date January 15th of 1982 would be returned as

```
01/15/82
```

TIME has the form HH:MM:SS for hours, minutes, and seconds, with separating colons (:). The value for hours ranges from 0 through 23. For example, the time 20 seconds past 3:12 PM would be returned as

```
15:12:20
```

If, when you call DQ$DECODE$TIME, the SYSTEM$TIME parameter is zero, the call first gets the system time (number of seconds since midnight, January 1, 1978) and then decodes it into the series of bytes as just described.

But if SYSTEM$TIME is not zero on input, DQ$DECODE$TIME uses it as the time to decode.

One thing your program can do with DQ$DECODE$TIME is first to call DQ$FILE$INFO to get two DWORD values associated with a file (the last time the file was updated and the time the file was created). Then the program can call DQ$DECODE$TIME to interpret the times.

Condition Codes

E$OK	0000H	No exceptional conditions.
E$SUPPORT	0023H	An unsupported operation was attempted.

In addition to the condition code listed above, DQ$DECODE$TIME can return the condition codes associated with the Basic I/O System call RQGETTIME, see the *Extended iRMX II Basic I/O System Calls Reference Manual* for details.

DQ$DELETE deletes an existing file.

```
CALL DQ$DELETE (path$ptr, except$ptr);
```

Input Parameter

path$ptr	A POINTER to a STRING containing a pathname of the file to be deleted.

Output Parameter

except$ptr	A POINTER to a WORD where the system places the condition code.

Description

A program can use this system call to delete a file. The immediate action this call takes is to mark the file for deletion. It does this rather than abruptly deleting the file, because it will not delete any file as long as there are existing connections to the file. DQ$DELETE will delete the file only when there are no longer any connections to the file, that is, when all existing connections have been detached. On the other hand, once the file is marked for deletion, no more connections may be obtained for the file by way of DQ$ATTACH.

Condition Codes

E$OK	0000H	No exceptional conditions.
E$FNEXIST	0021H	The specified file does not exist.
E$FACCESS	0026H	Access to the specified file is denied.

In addition to the condition codes listed above, DQ$DELETE can return the condition codes associated with the Extended I/O System call RQSDELETE$FILE. See the *Extended iRMX II Extended I/O System Calls Reference Manual* for details.

DQ$DETACH deletes a connection (but not the file) established by DQ$ATTACH or DQ$CREATE.

```
CALL DQ$DETACH (connection$t, except$ptr);
```

Input Parameter

connection$t	A TOKEN for the file connection to be deleted.

Output Parameter

except$ptr	A POINTER to a WORD where the system places the condition code.

Description

This system call deletes a file connection. If the connection is open, the DQ$DETACH system call automatically closes it first (see DQ$CLOSE). DQ$DETACH also deletes the file if the file has been marked for deletion, and this is the last existing connection to the file. The results of specifying an invalid connection are operating-system-dependent.

Condition Codes

E$OK	0000H	No exceptional conditions.

In addition to the condition code listed above, DQ$DETACH can return the condition codes associated with the Universal Development Interface system call DQ$CLOSE and the Extended I/O system call RQSDELETE$CONNECTION. See the DQ$CLOSE system call in this manual, or the *Extended iRMX II Extended I/O System Calls Reference Manual* for details.

DQ$EXIT transfers control from your program to the iRMX II Operating System. It does not return any value to the calling program, not even a condition code.

```
CALL DQ$EXIT (completion$code);
```

Input Parameter

completion$code — A WORD containing the encoded reason for termination of the program. See the following description for information about this value.

Description

DQ$EXIT terminates a program. Before the actual termination, all of the program's connections are closed and detached, and all memory allocated to the program by DQ$ALLOCATE is returned to the memory pool.

DQ$EXIT does not return a condition code to the calling program.

If the calling program is running as an I/O job, the calling task, normally the command line interpreter (CLI), receives an extended iRMX II condition code based on the value your program supplied in the end$code field when it called DQ$EXIT. This assumes the following sequence of events:

1. The CLI calls RQ$CREATE$IO$JOB, specifying a response mailbox in the call.
2. Your program, running as a task in the created I/O job, performs its duties and then calls DQ$EXIT, specifying an end$code value.
3. DQ$EXIT converts the end$code value into an extended iRMX II condition code, as follows:

end$code Value	iRMX II Condition Code	Associated Mnemonic	Meaning
0	0000H	E$OK	Termination was normal.
1	0C1H	E$WARNING$EXIT	Warning messages were issued.
2	0C2H	E$ERROR$EXIT	Errors were detected.
3	0C3H	E$FATAL$EXIT	Fatal errors were detected.
4	0C4H	E$ABORT$EXIT	The job was aborted.
5-65535	0C0H	E$UNKNOWN$EXIT	Cause of termination not known.

4. DQ$EXIT calls RQ$EXITIOJOB, specifying the extended iRMX II condition code in the user$fault$code field.

5. RQ$EXIT$IO$JOB places the condition code into the user$fault$code field of a message. Then RQ$EXIT$IO$JOB sends the message to the response mailbox set up by the earlier call to RQ$CREATE$IO$JOB.
6. The CLI, when it obtains the message from the response mailbox, can take appropriate actions. Note that it can call DQ$DECODE$EXCEPTION first, to convert the condition code into its associated mnemonic.

The CLI program supplied with the extended iRMX II Operating System ignores these UDI condition codes when they are returned in the user$fault$code field of the response message. These condition codes are ignored because the UDI is not required to be in the extended iRMX II Operating System, so the extended iRMX II CLI assumes that it is not. Therefore, if you want the CLI to take actions based on that code, you must provide your own CLI.

For more information about RQ$CREATE$IO$JOB, RQ$EXITIOJOB see the *Extended iRMX II Extended I/O System Reference Manual*; for more information on the format of the response message, see the *Extended iRMX II Extended I/O System User's Guide*.

DQ$FILE$INFO returns information about a file.

```
CALL  DQ$FILE$INFO (connection$t, mode, file$info$ptr, except$ptr);
```

INPUT PARAMETERS

connection$t — A TOKEN containing a connection for the file.

mode — An encoded BYTE specifying whether DQ$FILE$INFO is to return the User ID of the owner of the file. Encode as follows:

Value	Meaning
0	Do not return owner's User ID.
1	Return the owner's User ID.
2-255	Return E$SUPPORT exception.

Output Parameters

file$info$ptr — A POINTER to a structure into which the requested information is to be returned. The form of the structure is

```
DECLARE FILE$INFO STRUCTURE(
                          OWNER(15)       BYTE,
                          LENGTH          DWORD,
                          TYPE            BYTE,
                          OWNER$ACCESS    BYTE,
                          WORLD$ACCESS    BYTE,
                          CREATE$TIME     DWORD,
                          LAST$MOD$TIME   DWORD,
                          GROUP$ACCESS    BYTE,
                          RESERVED(19)    BYTE);
```

where

OWNER — A STRING containing (if requested) the User ID of the file's owner.

LENGTH — A DWORD that gives the size of the file in bytes.

TYPE — A value indicating the type of file, as follows:

Value	File Type
0	Data file
1	Directory file
2	System-specific file
3-255	Reserved

OWNER$ACCESS An encoded BYTE whose bits specify the type of access granted to the owner, as follows. When a bit is set, it means the type of access is granted; otherwise the type of access is denied. (Bit 0 is the low-order bit.)

Bit	Associated Access Type
0	Delete
1	Read (the data file) or Display (the directory)
2	Append (to the data file) or Add Entry (to the directory)
3	Update (read and write to the file) or Change Access (to the directory)
4	Execute the specified file. (Set to the value of bit 1 for compatibility with other operating systems.)
5-7	Reserved

WORLD$ACCESS An encoded BYTE whose bits specify the type of access granted to the WORLD (all users on the system). When a bit is set, it means the type of access is granted; otherwise the type of access is denied. (Bit 0 is the low-order bit.)

Bit	Associated Type of Access
0	Delete
1	Read (the data file) or Display (the directory)
2	Write (to the data file) or Add Entry (to the directory)
3	Update (read and write to the file) or Change Access (to the directory)
4	Execute the specified file. (Set to the value of bit 1 for compatibility with other operating systems.)
5-7	Reserved

CREATE$TIME The date and time that the file or directory was created, expressed as the number of seconds since midnight, January 1, 1978. (You can convert this date/time to ASCII characters by calling DQ$DECODE$TIME.)

LASTMODTIME The date and time that the file or directory was last modified. For data files, modified means written to or truncated; for directories, modified means an entry was changed or an entry was added. (You can convert this date/time to ASCII characters by calling DQ$DECODE$TIME.)

GROUP$ACCESS An encoded byte that is always set to the value of WORLD$ACCESS. The extended iRMX II UDI does not use GROUP$ACCESS, it is included for compatibility with other operating systems.

except$ptr A POINTER to a WORD where the system places the condition code.

Description

The DQ$FILE$INFO system call returns information, as described above, about a data file or a directory file.

Condition Codes

E$OK	0000H	No exceptional conditions.
E$SUPPORT	0023H	The mode parameter has a value greater than 1.

In addition to the condition codes listed above, DQ$FILE$INFO can return the condition codes associated with the Nucleus system calls RQ$CREATE$MAILBOX and RQ$RECEIVE$MESSAGE and the Basic I/O system call RQAGET$FILE$STATUS. See the *Extended iRMX II Nucleus System Calls Reference Manual* and the *Extended iRMX II Basic I/O System Calls Reference Manual* for details.

DQ$FREE returns to the system a segment of memory obtained earlier by DQ$ALLOCATE.

```
CALL DQ$FREE (seg$t, except$ptr);
```

Input Parameter

seg$t	A TOKEN containing the memory segment to be deleted. The TOKEN is returned by a DQ$ALLOCATE call and is no longer valid for this procedure once the call is made.

Output Parameter

except$ptr	A POINTER to a WORD where the system places the condition code.

Description

The DQ$FREE system call returns the specified segment to the memory pool from which it was allocated. A subsequent attempt to use this deleted segment may cause errors or unexpected results, since the memory may have been otherwise allocated.

Condition Codes

E$OK	0000H	No exceptional conditions.

In addition to the condition code listed above, DQ$FREE can return the condition codes associated with the Nucleus system call RQ$DELETE$SEGMENT. See the *Extended iRMX II Nucleus System Call Reference Manual* for details.

The DQGETARGUMENT system call returns arguments, one at a time, from a command line entered at the system console. This command line is either that which invoked the program containing the DQGETARGUMENT call or a command line entered while the program was running.

```
delimit$char = DQ$GET$ARGUMENT (argument$ptr, except$ptr);
```

Input Parameter

argument$ptr	A POINTER to a STRING (at least 81 bytes long) that will receive the argument.

Output Parameters

delimit$char	A BYTE which receives the delimiter character.
except$ptr	A POINTER to a WORD where the system places the condition code.

Description

Your program can call GET$ARGUMENT to get arguments from a command line. Each call returns an argument and the delimiter character following the argument.

Your program can use this command in two ways. One way is to get arguments from the command line used to invoke the program at the console. In this case, you can assume that the command line is already in a buffer that has automatically been provided for this purpose.

The other way to use this command is to get arguments from command lines that are entered in response to requests from your program. In this case, your program must supply a when calling DQ$READ. This is the buffer you want used when your program calls DQGETARGUMENT. To set this up, your program must call DQ$SWITCH$BUFFER before the call to DQGETARGUMENT.

A delimiter is returned only if the exception code is zero. The following delimiters are recognized by the extended iRMX II Operating System:

,) (= # ! % + - & ; < > [ ] \ ' | ~

as well as a space ( ) and all characters with ASCII values in the range 0 through 20H, or between 7FH and 0FFH.

Before returning arguments in response to DQGETARGUMENT, the system does the following editing on the contents of the command buffer:

- It strips out ampersands (&) and semicolons (;).
- Where multiple blanks are adjacent to each other between arguments, it replaces them with a single blank. (Tabs are treated as blanks.)
- It converts lowercase characters to uppercase unless they are part of a quoted string.
- It treats the command line and the buffer (after a DQ$SWITCH$BUFFER system call) as if they were preceded by a null delimiter.

When returning arguments in response to DQGETARGUMENT, the system considers strings enclosed between matching pairs of single or double quotes to be literals. The enclosing quotes are not returned as part of the argument.

Example

The following example illustrates the arguments and delimiters returned by successive calls to DQGETARGUMENT. The example assumes that the contents of the buffer are

```
PLM286 LINKER.PLM PRINT(:LP:) NOLIST
```

The following shows what is returned if DQGETARGUMENT is called five times.

Call Number	Argument Returned	Delimiter Returned
1	(06H)PLM286	space
2	(0AH)LINKER.PLM	space
3	(05H)PRINT	(
4	(04H):LP:	)
5	(06H)NOLIST	cr

Note that the argument returned has the form of an iRMX II string, with the first byte devoted to specifying the length of the string. In the second call, there are ten characters in the argument, so the first byte contains 0AH.

Note that the last delimiter for the example is a carriage return (cr). This is how your program can determine that there are no more arguments in the command line.

Condition Codes

E$OK	0000H	No exceptional conditions.
E$STRING$BUFFER	0081H	An argument has been found that is longer than 80 characters. This only indicates that another call to DQGETARGUMENT is needed to obtain the rest of the argument.

The DQGETCONNECTION$STATUS system call returns information about a file connection.

```
CALL DQ$GET$CONNECTION$STATUS (connection$t, info$ptr, except$ptr);
```

Input Parameter

connection$t — A TOKEN containing the connection whose status is desired.

Output Parameters

info$ptr — A POINTER to a structure into which the operating system is to place the status information. The structure has the following format:

```
DECLARE INFO STRUCTURE(
                    OPEN        BYTE,
                    ACCESS      BYTE,
                    SEEK        BYTE,
                    FILE$PTR    DWORD);
```

where

OPEN — A Boolean that is 0FFH (TRUE) if the connection is open; 000H (FALSE) otherwise.

ACCESS — Access privileges of the connection. The right is granted if the corresponding bit is set to 1. (Bit 0 is the low-order bit.)

Bit	Access
0	Delete
1	Read
2	Write
3	Update (read and write)
4	Execute (Set to the value of bit 1 for compatibility with other operating systems.)
5-7	Reserved

SEEK — Types of seek supported.

Value	Meaning
0	No seek allowed
3	Seek forward and backward

Other values are not meaningful.

FILE$PTR — This DWORD integer marks the current position in the file. The position is expressed as the number of bytes from the beginning of the file, the first byte being byte 0. This field is undefined if the file is not open or if seek is not supported by the device. (For example, seek operations are not valid for a line printer.)

except$ptr — A POINTER to a WORD where the system places the condition code.

Description

DQGETCONNECTION$STATUS returns information about a file. You might use this system call, for example, if your program has performed several read or write operations and you must determine where the file pointer is now located.

Condition Codes

E$OK	0000H	No exceptional conditions.
E$SUPPORT	0023H	An unsupported operation was attempted.

In addition to the condition code listed above, DQGETCONNECTION$STATUS can return the condition codes associated with the Extended I/O system call RQSGET$CONNECTION$STATUS. See the *Extended iRMX II Extended I/O System Calls Reference Manual* for details.

DQGETEXCEPTION$HANDLER returns the address of the current exception handler.

```
CALL DQ$GET$EXCEPTION$HANDLER (current$handler$ptr, except$ptr);
```

Output Parameters

current$handler$ptr	A POINTER to a STRUCTURE into which this system call returns the entry point of the current exception handler. This STRUCTURE has the same form as a long POINTER. DQ$TRAP$EXCEPTION specifies this entry point if it is called.
except$ptr	A POINTER to a WORD where the system places the condition code.

Description

DQGETEXCEPTION$HANDLER is a system call that returns the address of the current exception handler to your program. This is the address specified in the most recent call, if any, to DQ$TRAP$EXCEPTION. Otherwise, the value returned is the address of the system default exception handler.

This routine always returns a long POINTER, even if called from a program compiled under the SMALL model of segmentation. You can use this long POINTER in two ways:

- You can use it to make an indirect call to the current exception handler.
- After temporarily substituting another exception handler, you can use it to restore the current exception handler.

DQGETEXCEPTION$HANDLER is used in conjunction with DQ$TRAP$EXCEPTION and DQ$DECODE$EXCEPTION. See the descriptions of these calls for more information.

Condition Codes

E$OK	0000H	No exceptional conditions.

In addition to the condition code listed above, DQGETEXCEPTION$HANDLER can return the condition codes associated with the Nucleus system call RQGETEXCEPTION$HANDLER. See the *Extended iRMX II Nucleus System Calls Reference Manual* for details.

DQGETMSIZE returns the size, in BYTES, of the memory block specified.

```
size = DQ$GET$MSIZE(seg$ptr, exception$ptr);
```

Input Parameter

seg$ptr	A POINTER that indicates an area of memory that was allocated earlier by a call to DQ$MALLOCATE.

Output Parameters

size	A DWORD which receives the size (in BYTES) of the memory block previously allocated by DQ$MALLOCATE.
exception$ptr	A POINTER to a WORD where the system call places the condition code.

Description

The DQGETMSIZE system call returns the size, in bytes, of a segment allocated by the DQ$MALLOCATE system call. Okay folks! Does this call allocate memory in paragraphs and round up to the next highest multiple of 16 like DQGETSIZE? Are there any restrictions on who should use this call? Should I give any information about the memory being checked not always being in a new segment?

Condition Codes

E$OK	0000H	No exceptional conditions.
E$SUPPORT	0023H	An unsupported operation was attempted.

In addition to the condition codes listed above, DQGETMSIZE can return the condition codes associated with the Nucleus system call RQGETSIZE. See the *Extended iRMX II Nucleus System Calls Reference Manual* for details.

DQGETSIZE returns the size of a previously allocated memory segment.

```
size = DQ$GET$SIZE (seg$t, except$ptr);
```

Input Parameter

seg$t	A TOKEN for a segment of memory allocated by the DQ$ALLOCATE call.

Output Parameters

size	A WORD which, if not zero, contains the size, in bytes, of the segment identified by the seg$t parameter. if zero, indicates that the size of the segment is 65536 (64K) bytes.
except$ptr	A POINTER to a WORD where the system places the condition code.

Description

The GET$SIZE system call returns the size, in bytes, of a segment.

Condition Codes

E$OK	0000H	No exceptional conditions.

In addition to the condition code listed above, DQGETSIZE can return the condition codes associated with the Nucleus system call RQGETSIZE. See the *Extended iRMX II Nucleus System Calls Reference Manual* for details.

DQGETSYSTEM$ID returns the identity of the operating system providing the environment for the UDI.

```
CALL DQ$GET$SYSTEM$ID (id$ptr, except$ptr);
```

Output Parameters

id$ptr	A POINTER to a 21-BYTE buffer into which DQGETSYSTEM$ID places a STRING identifying the operating system.
except$ptr	A POINTER to a WORD where the system places the condition code.

Description

This system call returns the string

```
iRMX II
```

Condition Codes

E$OK	0000H	No exceptional conditions.

DQGETTIME returns the current date and time in character format. This procedure is obsolete.

```
CALL DQ$GET$TIME (date$time$ptr, except$ptr);
```

This system call is included only for compatibility with previous versions of the UDI. Use the more general DQ$DECODE$TIME system call for this function.

DQ$MALLOCATE requests that a specific amount of logically contiguous free memory be added to the existing memory available to the calling program.

```
seg$ptr = DQ$MALLOCATE (size, except$ptr);
```

Input Parameter

size	A DWORD that specifies the number of BYTES of memory being requested.

Output Parameters

seg$ptr	A POINTER that indicates the starting address of the acquired memory.
except$ptr	A POINTER to a word in which the system places the condition code.

Description

The DQ$MALLOCATE system call requests a specific amount of logically contiguous memory be added to the memory pool of the calling program. If the call is successful, the procedure returns a POINTER to the first byte of the acquired memory. If the call fails, the procedure returns a POINTER of undefined value and an exception code.

Multiple calls to DQ$MALLOCATE will result in multiple segments being allocated.

NOTE

DQ$MALLOCATE cannot be used in the PL/M-286 SMALL model of compilation.

Condition Codes

E$OK	0000H	No exceptional conditions.
E$MEM	0002H	Insufficient memory is available to fill the request.
E$SUPPORT	0023H	An unsupported operation was attempted.

In addition to the condition codes listed above, DQ$MALLOCATE can return the condition codes associated with the Nucleus system calls RQGETPOOL$ATTRIBUTES and RQ$CREATE$SEGMENT. See the *Extended iRMX II Nucleus System Calls Reference Manual* for details.

DQ$MFREE returns memory allocated, by DQ$MALLOCATE, to the available memory pool.

```
CALL DQ$MFREE (seg$ptr, exception$ptr);
```

INPUT PARAMETERS

seg$ptr	A POINTER to a block of memory that is to be returned to the available memory pool.

Output Parameters

exception$ptr	A POINTER to a WORD where the system places the condition code.

Description

The DQ$MFREE system call is used to return to available memory space a block of memory that was previously allocated using the DQ$MALLOCATE system call. Any memory freed by this call is no longer available to the calling program. Further attempts to use this area of memory may result in unexpected results since the memory referenced may have reallocated to another process.

In using the DQ$MFREE system call you must return an entire block of memory, it is not possible to return a portion of the memory allocated by a previous call to DQ$MALLOCATE.

Condition Codes

E$OK	0000H	No exceptional conditions.

In addition to the condition code listed above, DQ$MFREE can return the condition codes associated with the Nucleus system call RQ$DELETE$SEGMENT. See the *Extended iRMX II Nucleus System Call Reference Manual* for details.

The DQ$OPEN system call opens a file for I/O operations, specifies how the file will be accessed, and specifies the number of buffers needed to support the I/O operations.

```
CALL DQ$OPEN (connection$t, mode, num$buf, except$ptr);
```

INPUT PARAMETERS

connection$t — A TOKEN for the file connection to be opened.

mode — A BYTE specifying how the connection will be used to access the file. This value is encoded as follows:

Value	Meaning
1	Read only
2	Write only
3	Update (both reading and writing)
4	Reserved
5-7	Available for Xenix systems; ignored by iRMX II systems
8-255	Reserved

num$buf — A BYTE containing the number of buffers needed for this connection. Specifying a value larger than 0 implicitly requests that "double buffering" (that is, read-ahead and/or write-behind) is to be performed automatically. Specifying a value greater than 2, results in an E$SUPPORT error.

Output Parameter

except$ptr — A POINTER to a WORD where the system places the condition code.

Description

This system call prepares a connection for use with DQ$READ, DQ$WRITE, DQ$SEEK, and DQ$TRUNCATE commands. Your program can have up to six connections open simultaneously.

The DQ$OPEN system call does the following:

- Creates the requested buffers.

- Sets the connection's file pointer to zero. This a place marker that tells where in the file the next I/O operation is to begin.
- Starts reading ahead if num$buf is greater than zero and the access parameter is "Read only" or "Update."

Selecting Access Rights

The system does not allow reading using a connection open for writing only nor writing using a connection open for reading only. If you are not certain how the connection will be used, specify updating. However, if the specified connection does not support the specified type of access, an exception code is returned.

Selecting the Number of Buffers

The process of deciding how many buffers to request is based on three considerations--compatibility, memory, and performance.

COMPATIBILITY. If you expect to run your UDI program on other systems, which support the UDI, you should request no more than two buffers.

MEMORY. The amount of memory used for buffers is directly proportional to the number of buffers. You can save memory by using fewer buffers.

PERFORMANCE. The performance consideration is more complex. Up to a certain point, the more buffers you allocate, the faster your program can run. The actual break-even point, where more buffers don't improve performance, depends on many variables. Often, the only way to determine the break-even point is to experiment. However, the following statements are true of every system:

- To overlap I/O with computation, you must request at least two buffers.
- If performance is not at all important but memory is, request no buffers.

Requesting zero buffers means that no buffering is to occur. That is, each DQ$READ or DQ$WRITE is followed immediately by the physical I/O operation necessary to perform the requested reading or writing. Interactive programs should open :CI: and :CO: with a request for no buffers.

If your program normally calls DQ$SEEK before calling DQ$READ or DQ$WRITE, it should request one buffer.

Your program can use the DQ$RESERVE$IO$MEMORY call to reserve memory that the UDI can use for its internal data structures when the program calls DQ$ATTACH and for buffers when the program calls DQ$OPEN. The advantage of reserving memory is that the memory is guaranteed to be available when needed. If memory is not reserved, a call to DQ$OPEN might not be successful because of a memory shortage. See the description of DQ$RESERVE$IO$MEMORY later in this chapter for more information about reserving memory.

Condition Codes

E$OK	0000H	No exceptional conditions.
E$SUPPORT	0023H	At least one of the following is true: • The mode parameter is 4 or 8-255. • The num$buffs parameter is greater than two.
E$FACCESS	0026H	Access to the specified file is denied.
E$SHARE	0028H	The specified file may not be shared.
E$MEM	0002H	Insufficient memory remains to complete the call.

In addition to the condition codes listed above, DQ$OPEN can return the condition codes associated with the Extended I/O system call RQSOPEN. See the *Extended iRMX II Extended I/O System Calls Reference Manual* for details.

In systems using overlays, the root module calls DQ$OVERLAY to load an overlay module.

```
CALL DQ$OVERLAY (name$ptr, except$ptr);
```

Input Parameter

name$ptr	A POINTER to a STRING containing the name of an overlay module. The name must be in uppercase.

Output Parameter

except$ptr	A POINTER to a WORD where the system places the condition code.

Description

A root module, in an overlay system, calls DQ$OVERLAY each time it wants to load an overlay module.

If your assembly language or PL/M-286 program uses the DQ$OVERLAY procedure, you should ensure that you bind the UDI library to your program correctly. The *iAPX 286 Utilities* manual describes the OVL286 utility in detail. The following steps describe the process for loading iRMX II programs in overlay form.

1. Use BND286 to create linkable overlay files from compiled modules belonging to each overlay.
2. Use BND286 to create a nonpacked STL module from the linkable overlay files. The BND286 NOPACK control must be used.
3. Write an overlay definition file to describe the structure of the overlays in the program.
4. Use OVL286 to create an overlaid executable file from the linkable overlay files, the loadable module, and the overlay definition file.

To maintain portability to other operating systems that support the UDI, you should call no more than one level of overlay invoked only from the root of the application.

Condition Codes

E$OK	0000H	No exceptional conditions.
E$SUPPORT	0023H	An supported operation was attempted.

In addition to the condition code listed above, DQ$OVER$LAY can return the condition codes associated with the Extended I/O system call RQSOVERLAY. See the *Extended iRMX II Extended I/O System Calls Reference Manual* for details.

The DQ$READ system call copies bytes from a file into a buffer.

```
bytes$read = DQ$READ (connection$t, buff$ptr, count, except$ptr);
```

INPUT PARAMETERS

connection$t	A TOKEN for the connection to the file. This connection must be open for reading or for both reading and writing, and the file pointer of the connection must point to the first byte to be read.
buff$ptr	A POINTER to the buffer that is to receive the data from the file.
count	A WORD containing the requested number of bytes to be read from the file.

Output Parameters

bytes$read	A WORD containing the number of bytes actually read. This number is always equal to or less than count.
except$ptr	A POINTER to a WORD where the system places the condition code.

Description

This system call reads a collection of contiguous bytes from the file associated with the connection. The bytes are placed into the buffer specified in the call. If bytes$read is less than count and the exception code returned from the DQ$READ system call is E$OK, an end of file was encountered. If you type an interrupt or a terminate character from the console, for example a CONTROL-C, while the operating system performs a read operation, an E$OK exception code is returned and bytes$read is set to zero.

The Buffer

The buff$ptr parameter tells the operating system where to place the bytes when they are read. Your program must provide this buffer. DQ$READ copies as many bytes as it is instructed to copy (unless it encounters the end of the file). If the buffer is not long enough, copying continues beyond the end of the buffer.

Number of Bytes Read

The number of bytes that your program requests is the maximum number of bytes that DQ$READ copies into the buffer. However, there are circumstances under which the system reads fewer bytes.

- If the DQ$READ detects an end of file before reading the number of bytes requested, it returns only the bytes preceding the end of file. In this case, the bytes$read parameter is less than the count parameter, yet no exceptional condition is indicated.
- If an exceptional condition occurs during the reading operation, information in the buffer and the value of the bytes$read parameter are meaningless and should be ignored.
- If a CONTROL-C (interrupt or terminate) character is typed at the console (see description).

Connection Requirements

The connection must be open for reading or updating. If it is not, DQ$READ returns an exceptional condition.

Condition Codes

E$OK	0000H	No exceptional conditions.
E$SUPPORT	0023H	An unsupported operation was attempted.

In addition to the condition codes listed above, DQ$READ can return the condition codes associated with the Extended I/O system call RQSREAD$MOVE (except E$FLUSHING). See the *Extended iRMX II Extended I/O System Call Reference Manual* for details.

The DQ$RENAME system call changes the pathname of a file.

```
CALL DQ$RENAME (path$ptr, new$path$ptr, except$ptr);
```

Input Parameters

path$ptr	A POINTER to a STRING that specifies the pathname of the file to be renamed.
new$path$ptr	A POINTER to a STRING that specifies the new pathname for the file. This path must not refer to an existing file.

Output Parameter

except$ptr	A POINTER to a WORD where the system places the condition code.

Description

This system call allows your programs to change the pathname of a data or a directory file. Be aware that when you rename a directory, you are changing the pathnames of all files contained in the directory. When you rename a file to which a connection exists--this is permitted--the connection to the renamed file remains established.

A file's pathname may be changed in any way, provided the file or directory remains on the same volume. Successfully renaming a file without appropriate access permission depends on the operating system.

If your operating system does not allow renaming a file to another volume or storage device, an E$SUPPORT exception is returned.

Condition Codes

E$OK	000H	No exceptional conditions.
E$FEXIST	0020H	The file represented by new$path$ptr already exists.
E$SUPPORT	0023H	The file represented by new$path$ptr exists on another volume.
E$FNEXIST	0021H	The file represented by path$ptr does not exist.

In addition to these condition codes, DQ$RENAME can return the condition codes associated with the Extended I/O System call RQSRENAME$FILE. See the *Extended iRMX II Extended I/O System Calls Reference Manual* for details.

The DQ$RESERVE$IO$MEMORY system call lets your program reserve enough memory to ensure that it can open and attach the files it will be using.

```
CALL DQ$RESERVE$IO$MEMORY (number$files, number$buffers, except$ptr);
```

INPUT PARAMETERS

number$files — A WORD whose value indicates the maximum number of files the program will have attached simultaneously. This value must not be greater than 12. Moreover, no more than 6 of these files may be open simultaneously.

number$buffers — A WORD whose value indicates the total number of buffers (up to a maximum of 12) that will be needed at one time. For example, if your program will have two files open at the same time, and each of them has two buffers (specified when they are opened), number$files should be two and number$buffers four.

If you specify a value for number$files or number$buffers that exceeds the limits explained above, an E$SUPPORT exception will be returned. If you specify a zero for both number$files and number$buffers, the memory reserved earlier will be returned to the memory pool.

Output Parameter

except$ptr — A POINTER to a WORD where the system places the condition code.

Description

DQ$RESERVE$IO$MEMORY sets aside memory on behalf of the calling program, guaranteeing that it will be available when needed later for attaching and opening files. This memory is used for internal UDI data structures when the program requests file connections via DQ$ATTACH and for buffers when the program opens file connections via DQ$OPEN. Memory reserved in this way is not eligible to be allocated by DQ$ALLOCATE or DQ$MALLOCATE. Your program should call DQ$RESERVE$IO$MEMORY before making any calls to DQ$ALLOCATE or DQ$MALLOCATE.

DQ$RESERVE$IO$MEMORY

For an application to be portable across all operating systems that support UDI, it should not allow I/O without first explicitly reserving the memory by calling DQ$RESERVE$IO$MEMORY. In the call to DQ$RESERVE$IO$MEMORY, you may specify as many as 12 files (that can be attached using the reserved memory) and as many as 12 buffers (that can be requested when opening files).

NOTE

If a program calls DQ$RESERVE$IO$MEMORY after making one or more calls to DQ$ATTACH or DQ$OPEN, the memory used by those calls is immediately applied against the file and buffer counts specified in the DQ$RESERVE$IO$MEMORY call, possibly exhausting the memory supply being requested.

If your program calls DQ$RESERVE$IO$MEMORY more than once in a program, it simply increases or decreases the amount of memory reserved, unless your requests total more than 12 files or 12 buffers. If the requests exceed the maximum number of files or buffers, the maximum is reserved and no error is returned.

RESTRICTION

This system call is effective only if your program uses exclusively UDI system calls to communicate with the extended iRMX II Operating System.

Portability across operating systems that support the UDI cannot be guaranteed if your application requires more than 12 files attached simultaneously or a group of simultaneously open files whose total number of buffers exceeds 12.

Condition Codes

E$OK	0000H	No exceptional conditions.
E$MEM	0002H	Insufficient memory remains to complete the call.
E$SUPPORT	0023H	At least one of the following is true: • The value specified for number$files is greater than 12. • The value specified for number$buffers is greater than 12.

DQ$SEEK moves the file pointer associated with the specified connection.

```
CALL DQ$SEEK (connection$t, mode, offset, except$ptr)
```

Input Parameters

connection$t — A TOKEN for the open connection whose file pointer is to be moved.

mode — A BYTE indicating the type of file pointer movement being requested, as follows:

Mode	Meaning
1	Move the pointer backward by the specified move count. If the move count is large enough to position the pointer past the beginning of the file, the pointer is set to the first byte of the file (position zero).
2	Set the pointer to the position specified by the move count. Position zero is the first position in the file. Moving the pointer beyond the end of the file is permitted.
3	Move the file pointer forward by the specified move count. Moving the pointer beyond the end of the file is permitted.
4	First move the pointer to the end of the file and then move it backward by the specified move count. If the specified move count would position the pointer beyond the front of the file, the pointer is set to the first byte in the file (position zero).

offset — A DWORD specifying either how far, in bytes, the file pointer is to be moved, or the exact position in the file to which the pointer is to be moved.

Output Parameter

except$ptr — A POINTER to a WORD where the system places the condition code.

Description

When performing non-sequential I/O, your programs can use this system call to position the file pointer before using the DQ$READ, DQ$TRUNCATE, or DQ$WRITE system calls. The location of the file pointer specifies where in the file a DQ$READ, DQ$WRITE, or DQ$TRUNCATE operation is to begin. If your program is performing sequential I/O on a file, it need not use this system call.

You can position the file pointer beyond the end of a file. If your program does this and then invokes the DQ$READ system call, DQ$READ behaves as though the read operation began at the end of file. If your program calls DQ$WRITE when the file pointer is beyond the end of the file, the file is extended and the data is written as requested. A subsequent DQ$READ returns an end of file condition. Attempting a seek past the end of a file without performing an explicit DQ$WRITE call and subsequently expecting the file to be lengthened, will produce indeterminate results.

Condition Codes

E$OK	0000H	No exceptional conditions.
E$SUPPORT	0023H	The mode parameter was set to 0 or 5-255.

In addition to the condition code listed above, DQ$SEEK can return the condition codes associated with the Extended I/O system call RQSSEEK. See the *Extended iRMX II Extended I/O System Calls Reference Manual* for details.

DQ$SPECIAL sets options or specifies actions to be performed in the program execution environment.

```
CALL DQ$SPECIAL (mode, parameter$ptr, except$ptr);
```

Input Parameters

mode — A BYTE used to specify the options to be set or the actions to be performed. Values and meanings of mode are

Value	Meaning
1	Transparent
2	Line editing (default value)
3	Polling
4-5	Reserved
6	Baud rate

Each of these modes is explained in the Description section.

parameter$ptr — A POINTER. See complete explanation in the Description section.

Output Parameter

except$ptr — A POINTER to a WORD where the system places the condition code.

Description

This system call changes the mode in which your program receives input from a console input device. When your system starts to run, the mode is line editing (mode 2). By using DQ$SPECIAL, you can change to either of the other two modes, or back to line editing.

The meanings of the mode parameter values are as follows:

Value	Meaning
1	Transparent. Interactive programs must often obtain characters from the console exactly as they are typed. Transparent mode makes this possible. In transparent mode, normal input characters are placed in the buffer specified by the call to DQ$READ. Two exceptions to this are (1) signal characters (e.g., the Human Interface CONTROL-C) set by specifying "set signal" in the spec$func parameter of A$SPECIAL or S$SPECIAL, and (2) any enabled output control characters or . DQ$READ returns control to the calling program when the number of characters entered equals the number of characters specified in the read request.
2	Line Editing. This option enables you to correct typing errors with special keys before the application program receives the characters typed. Characters used for editing are operating-system-dependent. The RETURN character is always converted to CARRIAGE-RETURN-LINE-FEED (CRLF).
3	Polling. This option is nearly the same as Transparent (1) mode, except that in Polling mode DQ$READ returns control to your program immediately after it is called, regardless of whether any characters have been typed since the last call to DQ$READ. If no characters have been typed, this is indicated by the bytes$read parameter of the DQ$READ call. Characters typed between successive calls to read the terminal are held in the "type-ahead" buffer. where parameter$ptr — A POINTER to a TOKEN for a connection to the :CI: file previously established by DQ$ATTACH.
4-5	Reserved, E$SUPPORT will be returned.

6	Baud Rate — Specifies baud rate selection for an asynchronous line.
	
	where
	
	parameter$ptr points to this structure:
	
	`DECLARE LINE BASED parameter$ptr STRUCTURE (`
	`conn TOKEN,`
	`in$baud$rate BYTE,`
	`out$baud$rate BYTE);`

where

LINE.CONN is a connection previously established by a DQ$ATTACH call.

LINE.in$baud$rate specifies the desired input baud rate.

LINE.out$baud$rate specifies the desired output baud rate.

These values specify baud rate:

Byte Value	Baud Rate
0	Unspecified
1	300
2	600
3	1200
4	2400
5	4800
6	9600
7	19200
8-255	Reserved

Condition Codes

E$OK	000H	No exceptional conditions.
E$SUPPORT	0023H	The mode parameter represents an unsupported mode.

In addition to the condition codes listed above, DQ$SPECIAL can return the condition codes associated with the Extended I/O system call RQSSPECIAL. See the *Extended iRMX II Extended I/O System Calls Reference Manual* for details.

DQ$SWITCH$BUFFER substitutes a new command line for the existing one.

```
char$offset = DQ$SWITCH$BUFFER (buff$ptr, except$ptr);
```

Input Parameter

buff$ptr	A POINTER to a buffer containing the "new" command line. That is, the one whose arguments are to be returned by subsequent calls to DQGETARGUMENT. The buffer must not exceed 32 K-bytes in length.

Output Parameters

char$offset	A WORD into which the UDI places a number. This number represents the number of bytes from the beginning of the "old" command line to the last character of the last argument so far processed by DQGETARGUMENT. In other words, the value in char$offset tells how many characters in the old command line have been processed by the time of this call.
except$ptr	A POINTER to a WORD where the system places the condition code.

Description

When your program is invoked from the console, the operating system places the invocation command into a buffer. Typically, your program will use DQGETARGUMENT to obtain the arguments in that command. If your program subsequently calls DQ$READ to obtain an additional command line from the console, it can call DQ$SWITCH$BUFFER to designate the buffer with the new command line as that from which arguments are to be obtained when DQGETARGUMENT is called.

You can use DQ$SWITCH$BUFFER any number of times to point to different strings in your program. However, you cannot use DQ$SWITCH$BUFFER to return to the command line that invoked the program, because only the operating system knows the location of that buffer. Therefore, you should use DQGETARGUMENT to obtain all arguments of the invocation command line before issuing the first call to DQ$SWITCH$BUFFER.

A second service of DQ$SWITCH$BUFFER is that it returns the location of the last byte of the last argument so far obtained from the old buffer by calls to DQGETARGUMENT. Therefore, in addition to using DQ$SWITCH$BUFFER to switch buffers, you can use it after one or more DQGETARGUMENT calls to determine where in the buffer the next argument starts. However, doing this "resets" the buffer, in the sense that the next call to DQGETARGUMENT would return the first argument in the buffer. To return to the desired point in the buffer, where you can continue to extract arguments, call DQ$SWITCH$BUFFER again, but when doing so, use the sum of the starting address of the buffer and the value returned by the previous call to DQ$SWITCH$BUFFER. The following is an example showing how to use the second service of DQ$SWITCH$BUFFER:

```
DECLARE
    E$OK    LITERALLY '0'
    E$FATAL$EXIT    LITERALLY '3'
    mybuffer$ptr        POINTER,
    buff$ptr            POINTER,
    arg$ptr             POINTER,
    buff                STRUCTURE(
                offset    WORD,
                segment   WORD) AT (@buff$ptr),
    next$char            WORD,
    char$offset          WORD,
    condition$code      WORD,
    delimit$char        BYTE;
          •
          •
          •

/* initialize buff$ptr and next$char */

    buff$ptr = mybuff$ptr;
    next$char = 0;
          •
          •
          •

/* determine where in the buffer the next argument starts */

    char$offset = DQ$SWITCH$BUFFER( buff$ptr, @condition$code );
        IF condition$code <> E$OK THEN          /* do error processing  */
            CALL DQ$EXIT(E$FATAL$EXIT)
    next$char = char$offset + next$char;
```

(Example continued on next page)

```
/* return to desired point in buffer */

    buff.offset = buff.offset + char$offset;
    char$offset = DQ$SWITCH$BUFFER( buff$ptr, @condition$code );
        IF condition$code <> E$OK THEN          /* do error processing  */
            CALL DQ$EXIT(E$FATAL$EXIT)

/* get next argument */

    delimit$char = DQ$GET$ARGUMENT( arg$ptr, @condition$ptr );
        IF condition$code <> E$OK THEN          /* do error
        processing  */
            CALL DQ$EXIT(E$FATAL$EXIT)
        •
        •
        •
```

Condition Codes

E$OK	0000H	No exceptional conditions.

In addition to the condition code listed above, DQ$SWITCH$BUFFER can return the condition codes associated with the Human Interface system call RQCSET$PARSE$BUFFER. See the *Extended iRMX II Human Interface System Calls Reference Manual* for details.

DQ$TRAP$CC lets you specify a procedure that gains control if an operator enters an interrupt character (such as CONTROL-C) at the console.

```
CALL DQ$TRAP$CC (cc$routine$ptr, except$ptr);
```

Input Parameter

cc$routine$ptr	A POINTER to the entry point of your interrupt procedure.

Output Parameter

except$ptr	A POINTER to a WORD where the system places the condition code.

Description

The action the default interrupt procedure takes depends on the operating system. Using the DQ$TRAP$CC system call, lets you substitute an alternate interrupt procedure that will automatically receive control when you enter an interrupt character on the console. (See the *Extended iRMX II Human Interface User's Guide* for more information.) The context of the program executing at the time you invoke DQ$TRAP$CC must be saved by your operating system. Due to this context switch, the contents of the CPU registers at the time the interrupt procedure receives control may not be those associated with your program. The CPU registers may contain values for an internal task that was executing when the interrupt character was entered.

To ensure portability across other operating systems, a GOTO statement (PL/M, C, FORTRAN, etc.) must not branch outside the DQ$TRAP$CC procedure's routine.

Condition Codes

E$OK	0000H	No exceptional conditions.

DQ$TRAP$EXCEPTION substitutes an alternate exception handler for the default exception handler provided by the operating system.

```
CALL DQ$TRAP$EXCEPTION (handler$ptr, except$ptr);
```

Input Parameter

handler$ptr — A POINTER to a STRUCTURE containing a long pointer to the entry point of the alternate exception handler. The STRUCTURE has the form

```
DECLARE handler$ptr STRUCTURE (
                    offset  WORD,
                    base    TOKEN);
```

Output Parameter

except$ptr — A POINTER to a WORD where the system places the condition code.

Description

DQ$TRAP$EXCEPTION designates an alternate exception handler as the one to which control should pass when an exceptional condition occurs. The DQ$TRAP$EXCEPTION routine should restore the default exception handler before it terminates. Therefore, your program should call DQGETEXCEPTION$HANDLER before calling DQ$TRAP$EXCEPTION to get the default exception handler address.

See the section Condition Codes and Exception-Handling Calls at the beginning of this manual for an explanation of the conditions of the stack when your alternate exception handler receives control.

Condition Codes

E$OK	0000H	No exceptional conditions.

In addition to the condition code listed above, DQ$TRAP$EXCEPTION can return the condition codes associated with the Nucleus system call RQSETEXCEPTION$HANDLER. See the *Extended iRMX II Nucleus System Calls Reference Manual* for details.

DQ$TRUNCATE moves the end-of-file to the current position of a named file connection's file pointer, thereby freeing the portion of the file lying beyond the file pointer.

```
CALL DQ$TRUNCATE (connection$t, except$ptr);
```

Input Parameter

connection$t	A TOKEN for an open connection to the named data file that is to be truncated. The file pointer of this connection marks the place where truncation is to occur. The byte indicated by the file pointer is the first byte to be dropped from the file.

Output Parameter

except$ptr	A POINTER to a WORD where the system places the condition code.

Description

This system call truncates a file at the current setting of the file pointer and releases all file space beyond the pointer for reallocation to other files. If the pointer is at or beyond the end of file, no truncation is performed. Unless the file pointer is already at the proper location, your program should use the DQ$SEEK system call to position the pointer before calling DQ$TRUNCATE.

The connection should have write, or read and write access rights, established when the connection was opened.

Condition Codes

E$OK	0000H	No exceptional conditions.
E$SUPPORT	0023H	An unsupported operation was attempted.

In addition to the condition codes listed above, DQ$TRUNCATE can return the condition codes associated with the Extended I/O system call RQSTRUNCATE$FILE. See the *Extended iRMX II Extended I/O System Calls Reference Manual* for details.

The DQ$WRITE system call copies a collection of bytes from a buffer into a file.

```
CALL DQ$WRITE (connection$t, buff$ptr, count, except$ptr);
```

INPUT PARAMETERS

connection$t	A TOKEN containing the connection to the file into which the information is to be written.
buff$ptr	A POINTER to a buffer containing the data to be written to the specified file.
count	A WORD containing the number of bytes to be written from the buffer to the file.

Output Parameter

except$ptr	A POINTER to a WORD where the system places the condition code.

Description

This system call causes the operating system to write the specified number of bytes from the buffer to the file.

Number of Bytes Written

Occasionally, DQ$WRITE writes fewer bytes than requested by the calling program. This happens under the following two circumstances:

- When DQ$WRITE encounters an I/O error.
- When the volume to which your program is writing becomes full.

Where the Bytes Are Written

DQ$WRITE starts writing at the location specified by the connection's file pointer. After the writing operation is completed, the file pointer points to the byte immediately following the last byte written.

If your program must reposition the file pointer before writing, it can do so by using the DQ$SEEK system call.

Condition Codes

E$OK	0000H	No exceptional conditions.
E$SUPPORT	0023H	An unsupported operation was attempted.
E$SPACE	0029H	Inadequate memory space remains to complete the write.

In addition to the condition code listed above, DQ$WRITE can return the condition codes associated with the Extended I/O system call RQSWRITE$MOVE. See the *Extended iRMX II Extended I/O System Calls Reference Manual* for details.

Intel®

INDEX

A

B

C

D

E

F

I

L

M

O

P

R

S

T

U

INTERNATIONAL SALES OFFICES

INTEL CORPORATION
3065 Bowers Avenue
Santa Clara, California 95051

BELGIUM
Intel Corporation SA
Rue des Cottages 65
B-1180 Brussels

DENMARK
Intel Denmark A/S
Glentevej 61-3rd Floor
dk-2400 Copenhagen

ENGLAND
Intel Corporation (U.K.) LTD.
Piper's Way
Swindon, Wiltshire SN3 1RJ

FINLAND
Intel Finland OY
Ruosilante 2
00390 Helsinki

FRANCE
Intel Paris
1 Rue Edison-BP 303
78054 St.-Quentin-en-Yvelines Cedex

ISRAEL
Intel Semiconductors LTD.
Atidim Industrial Park
Neve Sharet
P.O. Box 43202
Tel-Aviv 61430

ITALY
Intel Corporation S.P.A.
Milandfiori, Palazzo E/4
20090 Assago (Milano)

JAPAN
Intel Japan K.K.
Flower-Hill Shin-machi
1-23-9, Shinmachi
Setagaya-ku, Tokyo 15

NETHERLANDS
Intel Semiconductor (Netherland B.V.)
Alexanderpoort Building
Marten Meesweg 93
3068 Rotterdam

NORWAY
Intel Norway A/S
P.O. Box 92
Hvamveien 4
N-2013, Skjetten

SPAIN
Intel Iberia
Calle Zurbaran 28-IZQDA
28010 Madrid

SWEDEN
Intel Sweden A.B.
Dalvaegen 24
S-171 36 Solna

SWITZERLAND
Intel Semiconductor A.G.
Talackerstrasse 17
8125 Glattbrugg
CH-8065 Zurich

WEST GERMANY
Intel Semiconductor G.N.B.H.
Seidlestrasse 27
D-8000 Munchen