iRMX® 1
Nucleus System Calls
Reference Manual
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 after 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:

<table>
<thead>
<tr>
<th>Mark</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Above iLBX</td>
<td>iLBX</td>
</tr>
<tr>
<td>BITBUS</td>
<td>iPSC</td>
</tr>
<tr>
<td>COMMputer</td>
<td>iRMX</td>
</tr>
<tr>
<td>CREDIT</td>
<td>iBSC</td>
</tr>
<tr>
<td>Data Pipeline</td>
<td>iSBX</td>
</tr>
<tr>
<td>Genius</td>
<td>iSDM</td>
</tr>
<tr>
<td>i</td>
<td>iSSB</td>
</tr>
<tr>
<td>i2ICE</td>
<td>Ripplemode</td>
</tr>
<tr>
<td>ICE</td>
<td>iRMX/80</td>
</tr>
<tr>
<td>iCEL</td>
<td>RUPi</td>
</tr>
<tr>
<td>iCS</td>
<td>Seamless</td>
</tr>
<tr>
<td>iDBP</td>
<td>SLD</td>
</tr>
<tr>
<td>iDIS</td>
<td>UPI</td>
</tr>
<tr>
<td>iISP</td>
<td>VLSiCEL</td>
</tr>
<tr>
<td>iPSB</td>
<td>376</td>
</tr>
<tr>
<td>iSBC</td>
<td>MULTIMODULE</td>
</tr>
<tr>
<td>iSSB</td>
<td>MULTIMODULE</td>
</tr>
<tr>
<td>iSBX</td>
<td>MULTIMODULE</td>
</tr>
<tr>
<td>iSMMX</td>
<td>MULTIMODULE</td>
</tr>
<tr>
<td>iXMT</td>
<td>MULTIMODULE</td>
</tr>
<tr>
<td>iXMT</td>
<td>MULTIMODULE</td>
</tr>
<tr>
<td>iXMT</td>
<td>MULTIMODULE</td>
</tr>
<tr>
<td>iXMT</td>
<td>MULTIMODULE</td>
</tr>
<tr>
<td>iXMT</td>
<td>MULTIMODULE</td>
</tr>
<tr>
<td>iXMT</td>
<td>MULTIMODULE</td>
</tr>
<tr>
<td>iXMT</td>
<td>MULTIMODULE</td>
</tr>
<tr>
<td>iXMT</td>
<td>MULTIMODULE</td>
</tr>
<tr>
<td>iXMT</td>
<td>MULTIMODULE</td>
</tr>
<tr>
<td>iXMT</td>
<td>MULTIMODULE</td>
</tr>
<tr>
<td>iXMT</td>
<td>MULTIMODULE</td>
</tr>
<tr>
<td>iXMT</td>
<td>MULTIMODULE</td>
</tr>
<tr>
<td>iXMT</td>
<td>MULTIMODULE</td>
</tr>
<tr>
<td>iXMT</td>
<td>MULTIMODULE</td>
</tr>
<tr>
<td>iXMT</td>
<td>MULTIMODULE</td>
</tr>
<tr>
<td>iXMT</td>
<td>MULTIMODULE</td>
</tr>
<tr>
<td>iXMT</td>
<td>MULTIMODULE</td>
</tr>
<tr>
<td>iXMT</td>
<td>MULTIMODULE</td>
</tr>
<tr>
<td>iXMT</td>
<td>MULTIMODULE</td>
</tr>
<tr>
<td>iXMT</td>
<td>MULTIMODULE</td>
</tr>
<tr>
<td>iXMT</td>
<td>MULTIMODULE</td>
</tr>
<tr>
<td>iXMT</td>
<td>MULTIMODULE</td>
</tr>
<tr>
<td>iXMT</td>
<td>MULTIMODULE</td>
</tr>
<tr>
<td>iXMT</td>
<td>MULTIMODULE</td>
</tr>
<tr>
<td>iXMT</td>
<td>MULTIMODULE</td>
</tr>
<tr>
<td>iXMT</td>
<td>MULTIMODULE</td>
</tr>
<tr>
<td>iXMT</td>
<td>MULTIMODULE</td>
</tr>
<tr>
<td>iXMT</td>
<td>MULTIMODULE</td>
</tr>
<tr>
<td>iXMT</td>
<td>MULTIMODULE</td>
</tr>
<tr>
<td>iXMT</td>
<td>MULTIMODULE</td>
</tr>
<tr>
<td>iXMT</td>
<td>MULTIMODULE</td>
</tr>
<tr>
<td>iXMT</td>
<td>MULTIMODULE</td>
</tr>
<tr>
<td>iXMT</td>
<td>MULTIMODULE</td>
</tr>
<tr>
<td>iXMT</td>
<td>MULTIMODULE</td>
</tr>
<tr>
<td>iXMT</td>
<td>MULTIMODULE</td>
</tr>
<tr>
<td>iXMT</td>
<td>MULTIMODULE</td>
</tr>
<tr>
<td>iXMT</td>
<td>MULTIMODULE</td>
</tr>
<tr>
<td>iXMT</td>
<td>MULTIMODULE</td>
</tr>
<tr>
<td>iXMT</td>
<td>MULTIMODULE</td>
</tr>
<tr>
<td>iXMT</td>
<td>MULTIMODULE</td>
</tr>
<tr>
<td>iXMT</td>
<td>MULTIMODULE</td>
</tr>
<tr>
<td>iXMT</td>
<td>MULTIMODULE</td>
</tr>
<tr>
<td>iXMT</td>
<td>MULTIMODULE</td>
</tr>
<tr>
<td>iXMT</td>
<td>MULTIMODULE</td>
</tr>
<tr>
<td>iXMT</td>
<td>MULTIMODULE</td>
</tr>
<tr>
<td>iXMT</td>
<td>MULTIMODULE</td>
</tr>
<tr>
<td>iXMT</td>
<td>MULTIMODULE</td>
</tr>
<tr>
<td>iXMT</td>
<td>MULTIMODULE</td>
</tr>
<tr>
<td>iXMT</td>
<td>MULTIMODULE</td>
</tr>
<tr>
<td>iXMT</td>
<td>MULTIMODULE</td>
</tr>
<tr>
<td>iXMT</td>
<td>MULTIMODULE</td>
</tr>
<tr>
<td>iXMT</td>
<td>MULTIMODULE</td>
</tr>
<tr>
<td>iXMT</td>
<td>MULTIMODULE</td>
</tr>
<tr>
<td>iXMT</td>
<td>MULTIMODULE</td>
</tr>
<tr>
<td>iXMT</td>
<td>MULTIMODULE</td>
</tr>
<tr>
<td>iXMT</td>
<td>MULTIMODULE</td>
</tr>
<tr>
<td>iXMT</td>
<td>MULTIMODULE</td>
</tr>
<tr>
<td>iXMT</td>
<td>MULTIMODULE</td>
</tr>
</tbody>
</table>

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, PC/XT, and PC/AT are registered trademarks of International Business Machines. Soft-Scope is a registered trademark of Concurrent Sciences.

<table>
<thead>
<tr>
<th>REV.</th>
<th>REVISION HISTORY</th>
<th>DATE</th>
</tr>
</thead>
<tbody>
<tr>
<td>-001</td>
<td>Original Issue.</td>
<td>02/89</td>
</tr>
</tbody>
</table>

iii/iv
This manual documents the system calls of the Nucleus, the main subsystem of the iRMX® I 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 iRMX® I Nucleus User's Guide and with the PL/M-86 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 blue ink; subsequent appearances are in black.

Throughout this manual, system calls are shown using a generic shorthand (such as CREATE$TASK instead of RQ$CREATE$TASK). This convention is used to allow easier alphabetic arrangement of the calls. The actual PL/M-86 external-procedure names must be used in all calling sequences.

NOTE

The values NIL and SELECTOR$OF(NIL) are used throughout this manual. You may also use a value of zero in place of NIL and SELECTOR$OF(NIL). However, Intel recommends that you use NIL and SELECTOR$OF(NIL) in your iRMX I code to maintain upward compatibility with the iRMX II Operating System. For a description of the SELECTOR$OF and NIL built-in functions, refer to the PL/M user's guide.

You can also invoke the system calls from assembly language, but you must obey the PL/M-86 calling sequences when doing so. For more information on these calling sequences refer to the iRMX® I Programming Techniques Reference Manual.
Chapter 1. iRMX® I Nucleus System Calls

Introduction .................................................................................................................. 1
System Call Dictionary ................................................................................................. 3
ACCEPT$CONTROL ........................................................................................................ 7
ALTER$COMPOSITE ................................................................................................. 10
CATALOG$OBJECT .................................................................................................... 12
CREATE$COMPOSITE ............................................................................................... 15
CREATE$EXTENSION ................................................................................................. 18
CREATE$JOB .............................................................................................................. 21
CREATE$MAILBOX ...................................................................................................... 29
CREATE$REGION ....................................................................................................... 33
CREATE$SEGMENT ..................................................................................................... 36
CREATE$SEMAPHORE ............................................................................................... 39
CREATE$TASK ............................................................................................................ 42
DELETE$COMPOSITE .............................................................................................. 47
DELETE$EXTENSION ............................................................................................... 49
DELETE$JOB ............................................................................................................ 52
DELETE$MAILBOX ..................................................................................................... 55
DELETE$REGION ....................................................................................................... 58
DELETE$SEGMENT ..................................................................................................... 61
DELETE$SEMAPHORE ............................................................................................... 64
DELETE$TASK ............................................................................................................ 67
DISABLE ..................................................................................................................... 71
DISABLE$DELETION ................................................................................................. 81
ENABLE ..................................................................................................................... 77
ENABLE$DELETION ................................................................................................. 84
END$INIT$TASK ....................................................................................................... 84
ENTER$INTERRUPT .................................................................................................... 85
EXIT$INTERRUPT ....................................................................................................... 89
FORCE$DELETE ........................................................................................................ 92
GET$EXCEPT$HANDLER ......................................................................................... 95
GET$LEVEL ................................................................................................................ 97
GET$POOL$ATTRIB ................................................................................................. 99
GET$PRIORITY ......................................................................................................... 102
GET$SIZE ................................................................................................................ 105
GET$TASK$TOKENS ............................................................................................... 108
GET$TYPE ................................................................................................................. 110
INSPECT$COMPOSITE ......................................................................................... 113
LOOKUP$OBJECT ................................................................................................. 115
## Chapter 1. iRMX® I Nucleus System Calls (continued)

<table>
<thead>
<tr>
<th>Call</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>OFFSPRING</td>
<td>118</td>
</tr>
<tr>
<td>RECEIVE$CONTROL</td>
<td>121</td>
</tr>
<tr>
<td>RECEIVE$MESSAGE</td>
<td>124</td>
</tr>
<tr>
<td>RECEIVE$UNITS</td>
<td>128</td>
</tr>
<tr>
<td>RESET$INTERUPT</td>
<td>131</td>
</tr>
<tr>
<td>RESUME$TASK</td>
<td>135</td>
</tr>
<tr>
<td>SEND$CONTROL</td>
<td>139</td>
</tr>
<tr>
<td>SEND$MESSAGE</td>
<td>142</td>
</tr>
<tr>
<td>SEND$UNITS</td>
<td>146</td>
</tr>
<tr>
<td>SET$EXCEPTION$HANDLER</td>
<td>148</td>
</tr>
<tr>
<td>SET$INTERUPT</td>
<td>154</td>
</tr>
<tr>
<td>SET$OS$EXTENSION</td>
<td>159</td>
</tr>
<tr>
<td>SET$POOL$MIN</td>
<td>162</td>
</tr>
<tr>
<td>SET$PRIORITY</td>
<td>165</td>
</tr>
<tr>
<td>SIGNAL$EXCEPTION</td>
<td>169</td>
</tr>
<tr>
<td>SIGNAL$INTERUPT</td>
<td>172</td>
</tr>
<tr>
<td>SLEEP</td>
<td>177</td>
</tr>
<tr>
<td>SUSPEND$TASK</td>
<td>179</td>
</tr>
<tr>
<td>UNCATALOG$OBJECT</td>
<td>182</td>
</tr>
<tr>
<td>WAIT$INTERUPT</td>
<td>186</td>
</tr>
</tbody>
</table>
INTRODUCTION

This manual lists the iRMX® I Nucleus system calls in alphabetical order and provides a detailed description of each one.

The calling sequence for each call is the same as for the PL/M-86 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-86 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-86 data types such as BYTE, WORD, POINTER and SELECTOR are used. In addition, the iRMX I data types TOKEN and STRING are used. (See the iRMX® I Nucleus User's Guide for more information on these data types.) They are always capitalized, and their definitions are found in Appendix A of the iRMX® Extended I/O System User's Guide. 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-86 data type, you must declare it to be literally a SELECTOR or a WORD every place you use it. The word "token" in lowercase refers to a value that the iRMX I 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 I Operating System. Refer to the iRMX® I Programming Techniques Manual for additional information on creating this INCLUDE statement. Further, there is also a literal declaration for TOKEN, which is used in the examples. For the sake of simplicity, the examples assume that an established exception handler is to deal with exceptional conditions. Consequently, they do not illustrate in-line exception processing.
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.
# SYSTEM CALL DICTIONARY

<table>
<thead>
<tr>
<th>Call</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>CREATE$JOB</td>
<td>Creates a job with a task and returns a token for the job.</td>
<td>21</td>
</tr>
<tr>
<td>DELETE$JOB</td>
<td>Deletes a childless job that contains no extension objects (extension</td>
<td>52</td>
</tr>
<tr>
<td></td>
<td>objects are described in the iRMX® <em>Nucleus User's Guide</em>).</td>
<td></td>
</tr>
<tr>
<td>OFFSPRING</td>
<td>Provides a segment containing tokens of the child jobs of the specified job.</td>
<td>118</td>
</tr>
<tr>
<td>CREATE$TASK</td>
<td>Creates a task and returns a token for it.</td>
<td>42</td>
</tr>
<tr>
<td>DELETE$TASK</td>
<td>Deletes a task that is not an interrupt task.</td>
<td>67</td>
</tr>
<tr>
<td>GET$PRIORITY</td>
<td>Returns the static priority of a task.</td>
<td>102</td>
</tr>
<tr>
<td>GET$TASK$TOKENS</td>
<td>Returns to the caller a token for either itself, its job, its job's</td>
<td>108</td>
</tr>
<tr>
<td></td>
<td>parameter object, or the root job.</td>
<td></td>
</tr>
<tr>
<td>RESUME$TASK</td>
<td>Decreases a task's suspension depth by one; resumes (unsuspends) the task if</td>
<td>135</td>
</tr>
<tr>
<td></td>
<td>the suspension depth becomes zero.</td>
<td></td>
</tr>
<tr>
<td>SET$PRIORITY</td>
<td>Changes a task’s priority.</td>
<td>165</td>
</tr>
<tr>
<td>SLEEP</td>
<td>Places the calling task in the asleep state for a specified amount of time.</td>
<td>178</td>
</tr>
<tr>
<td>SUSPEND$TASK</td>
<td>Increases a task’s suspension depth by one; suspends the task if it is not</td>
<td>180</td>
</tr>
<tr>
<td></td>
<td>already suspended.</td>
<td></td>
</tr>
<tr>
<td>CREATE$MAILBOX</td>
<td>Creates a mailbox and returns a token for it.</td>
<td>29</td>
</tr>
<tr>
<td>DELETE$MAILBOX</td>
<td>Deletes a mailbox.</td>
<td>55</td>
</tr>
<tr>
<td>RECEIVE$MESSAGE</td>
<td>Allows the calling task to receive an object; the task has the option of</td>
<td>124</td>
</tr>
<tr>
<td></td>
<td>waiting if no objects are present.</td>
<td></td>
</tr>
<tr>
<td>SEND$MESSAGE</td>
<td>Sends an object to a mailbox.</td>
<td>142</td>
</tr>
<tr>
<td>Call</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>----------------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>CREATE$SEMAPHORE</td>
<td>Creates a semaphore and returns a token for it.</td>
<td>39</td>
</tr>
<tr>
<td>DELETE$SEMAPHORE</td>
<td>Deletes a semaphore.</td>
<td>64</td>
</tr>
<tr>
<td>RECEIVE$UNITS</td>
<td>Asks for a specific number of units from a semaphore.</td>
<td>128</td>
</tr>
<tr>
<td>SEND$UNITS</td>
<td>Adds a specific number of units to a semaphore.</td>
<td>146</td>
</tr>
</tbody>
</table>

**SEMAPHORES**

<table>
<thead>
<tr>
<th>Call</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>CREATE$SEGMENT</td>
<td>Creates a segment and returns a token for it.</td>
<td>35</td>
</tr>
<tr>
<td>DELETE$SEGMENT</td>
<td>Returns a segment to the memory pool from which it was allocated.</td>
<td>61</td>
</tr>
<tr>
<td>GET$POOL$ATTRIBUTES</td>
<td>Returns the following memory pool attributes of the caller’s job:</td>
<td>99</td>
</tr>
<tr>
<td></td>
<td>pool minimum, pool maximum, initial size, number of allocated</td>
<td></td>
</tr>
<tr>
<td></td>
<td>16-byte paragraphs, number of available 16-byte paragraphs.</td>
<td></td>
</tr>
<tr>
<td>GET$SIZE</td>
<td>Returns the size, in bytes, of a segment.</td>
<td>105</td>
</tr>
<tr>
<td>SET$POOL$MIN</td>
<td>Changes the minimum attribute of the memory pool of the caller’s job.</td>
<td>162</td>
</tr>
</tbody>
</table>

**SEGMENTS AND MEMORY POOLS**

<table>
<thead>
<tr>
<th>Call</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>CATALOG$OBJECT</td>
<td>Places an object in an object directory.</td>
<td>12</td>
</tr>
<tr>
<td>GET$TYPE</td>
<td>Accepts a token for an object and returns its type code.</td>
<td>110</td>
</tr>
<tr>
<td>LOOKUP$OBJECT</td>
<td>Accepts a cataloged name of an object and returns a token for it.</td>
<td>115</td>
</tr>
<tr>
<td>UNCATALOG$OBJECT</td>
<td>Removes an object from an object directory.</td>
<td>183</td>
</tr>
</tbody>
</table>

**ALL OBJECTS**

<table>
<thead>
<tr>
<th>Call</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>GET$EXCEPTION-$HANDLER</td>
<td>Returns the current values of the caller’s exception handler</td>
<td>95</td>
</tr>
<tr>
<td></td>
<td>and exception mode attributes.</td>
<td></td>
</tr>
<tr>
<td>SET$EXCEPTION-$HANDLER</td>
<td>Sets the exception handler and exception mode attributes of the caller.</td>
<td>148</td>
</tr>
</tbody>
</table>

**EXCEPTION HANDLERS**
### INTERRUPT HANDLERS, TASKS, AND LEVELS

(* indicates the system calls that an interrupt handler can make)

<table>
<thead>
<tr>
<th>Call</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>*DISABLE</td>
<td>Disables an interrupt level.</td>
<td>71</td>
</tr>
<tr>
<td>ENABLE</td>
<td>Enables an interrupt level.</td>
<td>77</td>
</tr>
<tr>
<td>ENDINITTASK</td>
<td>Informs the root task that a synchronous initialization process has completed.</td>
<td>84</td>
</tr>
<tr>
<td>*ENTERINTERRUPT</td>
<td>Sets up a previously designated data segment base address for the calling interrupt handler.</td>
<td>85</td>
</tr>
<tr>
<td>*EXITINTERRUPT</td>
<td>Used by interrupt handlers to send an end-of-interrupt signal to hardware.</td>
<td>89</td>
</tr>
<tr>
<td>*GETLEVEL</td>
<td>Returns the interrupt level of highest priority for which an interrupt handler has started but has not yet finished processing.</td>
<td>97</td>
</tr>
<tr>
<td>RESETINTERRUPT</td>
<td>Cancels the assignment of an interrupt handler to a level and, if applicable, deletes the interrupt task for that level.</td>
<td>131</td>
</tr>
<tr>
<td>SETINTERRUPT</td>
<td>Assigns an interrupt handler and, if desired, an interrupt task to an interrupt level.</td>
<td>154</td>
</tr>
<tr>
<td>*SIGNALINTERRUPT</td>
<td>Used by interrupt handlers to invoke interrupt tasks.</td>
<td>173</td>
</tr>
<tr>
<td>WAITINTERRUPT</td>
<td>Puts the calling interrupt task to sleep until it is called into service by an interrupt handler.</td>
<td>187</td>
</tr>
</tbody>
</table>

### COMPOSITE OBJECTS

<table>
<thead>
<tr>
<th>Call</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALTERCOMPOSITE</td>
<td>Replaces components of composite objects.</td>
<td>10</td>
</tr>
<tr>
<td>CREATECOMPOSITE</td>
<td>Creates a composite object and returns a token for it.</td>
<td>15</td>
</tr>
<tr>
<td>DELETECOMPOSITE</td>
<td>Deletes a composite object.</td>
<td>47</td>
</tr>
<tr>
<td>INSPECTCOMPOSITE</td>
<td>Returns a list of the component tokens contained in a composite object.</td>
<td>113</td>
</tr>
</tbody>
</table>

### EXTENSION OBJECTS

<table>
<thead>
<tr>
<th>Call</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>CREATEEXTENSION</td>
<td>Creates a new object type and returns a token for it.</td>
<td>18</td>
</tr>
<tr>
<td>DELETEEXTENSION</td>
<td>Deletes an extension object and all composites of that type.</td>
<td>49</td>
</tr>
</tbody>
</table>
## NUCLEUS SYSTEM CALL DICTIONARY

### DELETION CONTROL

<table>
<thead>
<tr>
<th>Call</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>DISABLE$DELETION</td>
<td>Makes an object immune to ordinary deletion.</td>
<td>74</td>
</tr>
<tr>
<td>ENABLE$DELETION</td>
<td>Makes an object susceptible to ordinary deletion. Required only if the object has had its deletion disabled.</td>
<td>81</td>
</tr>
<tr>
<td>FORCE$DELETE</td>
<td>Deletes objects whose disabling depths are zero or one.</td>
<td>92</td>
</tr>
</tbody>
</table>

### OPERATING SYSTEM EXTENSIONS

<table>
<thead>
<tr>
<th>Call</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>SET$OS$EXTENSION</td>
<td>Either enters the address of an entry (or function) procedure in the Interrupt Vector Table or deletes such an entry.</td>
<td>159</td>
</tr>
<tr>
<td>SIGNAL$EXCEPTION</td>
<td>Used by OS extensions to signal the occurrence of an exception.</td>
<td>169</td>
</tr>
</tbody>
</table>

### REGIONS

<table>
<thead>
<tr>
<th>Call</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACCEPT$CONTROL</td>
<td>Causes the calling task to accept control from the region only if control is immediately available. If control is not available, the calling task does not wait at the region.</td>
<td>7</td>
</tr>
<tr>
<td>CREATE$REGION</td>
<td>Creates a region and returns a token for it.</td>
<td>33</td>
</tr>
<tr>
<td>DELETE$REGION</td>
<td>Deletes a region.</td>
<td>58</td>
</tr>
<tr>
<td>RECEIVE$CONTROL</td>
<td>Causes the calling task to wait at the region until the task receives control.</td>
<td>121</td>
</tr>
<tr>
<td>SEND$CONTROL</td>
<td>Relinquishes control to the next task waiting at the region.</td>
<td>139</td>
</tr>
</tbody>
</table>
The ACCEPT$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 I 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';
/* if your PL/M compiler does not support this variable type, 
declare TOKEN a WORD */

CERT$EXT declares all nucleus system calls */
$INCLUDE(:RMX:INC/CERT$EXT)

DECLARE region$token TOKEN;
DECLARE priority$queue LITERALLY '1'; /* tasks wait in 
priority order */
DECLARE status WORD;

SAMPLEPROCEDURE:
    PROCEDURE;

    •
    • Typical PL/M-86 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-86 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-86 Statements

    •
When the task is ready to relinquish access to the data protected by the region, it invokes the SEND$CONTROL system call.

CALL RQS$SEND$CONTROL (@status);

Typical PL/M-86 Statements

END SAMPLEPROCEDURE;

Condition Codes

<table>
<thead>
<tr>
<th>Code</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>E$OK</td>
<td>0000H</td>
<td>No exceptional conditions.</td>
</tr>
<tr>
<td>E$BUSY</td>
<td>0003H</td>
<td>Another task currently has access to the protected data.</td>
</tr>
<tr>
<td>E$CONTEXT</td>
<td>0005H</td>
<td>The calling task currently has access to the region in question.</td>
</tr>
<tr>
<td>E$EXIST</td>
<td>0006H</td>
<td>The region parameter is not a token for an existing object.</td>
</tr>
<tr>
<td>E$NOT$CONFIGURED</td>
<td>0008H</td>
<td>This system call is not part of the present configuration.</td>
</tr>
<tr>
<td>E$TYPE</td>
<td>8002H</td>
<td>The region parameter is a token for an object that is not a region.</td>
</tr>
</tbody>
</table>
The ALTER$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$obj**
  A TOKEN for the replacement component object. A value of SELECTOR$OF(NIL) or zero represents no object.

**Output Parameter**

- **except$ptr**
  A POINTER to a WORD to which the iRMX I 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) or zero 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 iRMX® I Nucleus User's Guide.

Condition Codes

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>E$OK</td>
<td>0000H  No exceptional conditions.</td>
</tr>
<tr>
<td>E$CONTEXT</td>
<td>0005H  The composite parameter is not compatible with the extension parameter.</td>
</tr>
<tr>
<td>E$EXIST</td>
<td>0006H  The extension, composite, or object parameter(s) is not a token for an existing object.</td>
</tr>
<tr>
<td>E$NOT$CONFIGURED</td>
<td>0008H</td>
</tr>
<tr>
<td>E$TYPE</td>
<td>8002H  One or both of the extension or composite parameters is a token for an object that is not of the correct object type.</td>
</tr>
<tr>
<td>E$PARAM</td>
<td>8004H  The component$index parameter refers to a nonexistent position in the component object list.</td>
</tr>
</tbody>
</table>
CATALOG$OBJECT

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) or zero, it indicates that the object is to be cataloged in the object directory of the job to which the calling task belongs.

- If not SELECTOR$OF(NIL) or zero, 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) or zero 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 I 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';
/** if your PL/M compiler does not support this variable type, declare TOKEN a WORD */

/** NUCLUS.EXT declares all nucleus system calls */
$INCLUDE(:RMX:INC/NUCLUS.EXT)

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

SAMPLEPROCEDURE:
PROCEDURE;

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-86 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-86 Statements
CATALOG$OBJECT

************************************************************************
* After creating the mailbox, the calling task catalogues 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-86 Statements

END SAMPLEPROCEDURE;

Condition Codes

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>E$OK</td>
<td>0000H  No exceptional conditions.</td>
</tr>
<tr>
<td>E$CONTEXT</td>
<td>0005H At least one of the following is true:</td>
</tr>
<tr>
<td></td>
<td>• The name being cataloged is already in the designated object directory.</td>
</tr>
<tr>
<td></td>
<td>• The directory's maximum allowable size is 0.</td>
</tr>
<tr>
<td>E$EXIST</td>
<td>0006H Either the job parameter, which is not SELECTOR$OF(NIL) or zero, or the object parameter is not a token for an existing object.</td>
</tr>
<tr>
<td>E$LIMIT</td>
<td>0004H The designated object directory is full.</td>
</tr>
<tr>
<td>E$NOT$CONFIGURED</td>
<td>0008H This system call is not part of the present operating system configuration.</td>
</tr>
<tr>
<td>E$PARAM</td>
<td>8004H The first BYTE of the STRING pointed to by the name parameter contains a zero or a value greater than 12.</td>
</tr>
<tr>
<td>E$TYPE</td>
<td>8002H The job parameter is a token for an object which is not a job or is not SELECTOR$OF(NIL) or zero.</td>
</tr>
</tbody>
</table>
CREATE$COMPOSITE

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$slots**
      - Number of elements in the component objects list that the composite object will contain. This number represents the maximum number of component objects that the composite object can handle. If num$slots is greater than num$used, the values in the extra elements should be set to zero.

    - **num$used**
      - Number of token elements to include in the composite. If num$used is greater than num$slots, the extra components are ignored.

    - **tokens(*)**
      - Tokens that will actually constitute the composite object.
CREATE$COMPOSITE

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 I 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$slots) indicates the number of tokens in the data structure; that is, the maximum number of component objects that can be part of a composite. Because you may need to specify that not all of the tokens in the token list be used in the composite object, the second element (num$used) indicates how many of those tokens are actually included in the composite. 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$used) is less than the number of tokens in the token$list (num$slots), CREATE$COMPOSITE sets the remaining elements to zero.

If, on the other hand, the number of tokens in the token list (num$slots) is less than the number of token elements to include in the composite (num$used), CREATE$COMPOSITE ignores the extra components in the token list.

Example


Condition Codes

<table>
<thead>
<tr>
<th>Code</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>E$OK</td>
<td>0000H</td>
<td>No exceptional conditions.</td>
</tr>
<tr>
<td>E$EXIST</td>
<td>0006H</td>
<td>The extension parameter or one or more of the non-zero token$list parameters is not a token for an existing object.</td>
</tr>
<tr>
<td>E$LIMIT</td>
<td>0004H</td>
<td>The calling task's job has already reached its object limit.</td>
</tr>
<tr>
<td>Code</td>
<td>Description</td>
<td></td>
</tr>
<tr>
<td>--------</td>
<td>----------------------------------------------------------------------------</td>
<td></td>
</tr>
<tr>
<td>E$MEM</td>
<td>0002H The memory available to the calling task's job is insufficient to create a composite.</td>
<td></td>
</tr>
<tr>
<td>E$NOT$CONFIGURED</td>
<td>0008H This system call is not part of the present configuration.</td>
<td></td>
</tr>
<tr>
<td>E$PARAM</td>
<td>8004H The specified number of components is zero.</td>
<td></td>
</tr>
<tr>
<td>E$TYPE</td>
<td>8002H The extension parameter is a token for an object that is not an extension object.</td>
<td></td>
</tr>
</tbody>
</table>
CREATE$EXTENSION

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) or zero 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$ptr**: A POINTER to a WORD to which the iRMX I Operating System will return the condition code generated by this system call.
CREATE$EXTENSION

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) or zero. 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 the "Initialization" section of Chapter 10 in the iRMX® I Nucleus User's Guide.

Condition Codes

<table>
<thead>
<tr>
<th>Code</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>E$OK</td>
<td>0000H</td>
<td>No exceptional conditions.</td>
</tr>
<tr>
<td>E$CONTEXT</td>
<td>0005H</td>
<td>The calling task's job is being deleted.</td>
</tr>
<tr>
<td>E$EXIST</td>
<td>0006H</td>
<td>The deletion$mailbox parameter is not a token for an existing object.</td>
</tr>
<tr>
<td>E$LIMIT</td>
<td>0004H</td>
<td>The calling task's job has reached its object limit.</td>
</tr>
<tr>
<td>E$MEM</td>
<td>0002H</td>
<td>The memory available to the calling task's job is not sufficient to create an extension.</td>
</tr>
<tr>
<td>E$NOT$CONFIGURED</td>
<td>0008H</td>
<td>This system call is not part of the present configuration.</td>
</tr>
<tr>
<td>Code</td>
<td>Description</td>
<td></td>
</tr>
<tr>
<td>-------</td>
<td>-------------------------------------------------------</td>
<td></td>
</tr>
<tr>
<td>E$PARAM 8004H</td>
<td>The type parameter is invalid.</td>
<td></td>
</tr>
<tr>
<td>E$TYPE 8002H</td>
<td>The deletion mailbox parameter is a token for an object that is not a mailbox.</td>
<td></td>
</tr>
</tbody>
</table>
The CREATE$JOB system call creates a job with a single task.

```
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 $0FF00$.
- **param$obj**: A TOKEN indicating the presence or absence of a parameter object. See the *iRMX® Nucleus User's Guide* for an explanation of parameter objects.
  - If not SELECTOR$OF(NIL) or zero, it must contain a token for the new job's parameter object.
  - If set to SELECTOR$OF(NIL) or zero, it indicates that the new job has no parameter object.
- **pool$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 ($20H$ bytes). If the stack$ptr parameter has a base value of SELECTOR$OF(NIL) or zero, 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.
CREATE$JOB

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

- If not OFFFFH, it contains the maximum number of tasks that can exist simultaneously in the new job.
- If OFFFFH, 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:

<table>
<thead>
<tr>
<th>Value</th>
<th>To Exception Handler</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Never</td>
</tr>
<tr>
<td>1</td>
<td>On programmer errors only</td>
</tr>
<tr>
<td>2</td>
<td>On environmental conditions only</td>
</tr>
<tr>
<td>3</td>
<td>On all exceptional conditions</td>
</tr>
</tbody>
</table>
CREATE$JOB

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:

<table>
<thead>
<tr>
<th>Bits</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>15-2</td>
<td>Reserved bits that should be set to zero.</td>
</tr>
<tr>
<td>1</td>
<td>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. 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.</td>
</tr>
<tr>
<td>0</td>
<td>Reserved bit that should be set to zero.</td>
</tr>
</tbody>
</table>

task$priority  A BYTE that controls the priority of the new job's initial task.

- 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 not SELECTOR$OF(NIL) or zero, it is the base address of the data segment of the new job's initial task.

- If SELECTOR$OF(NIL) or zero, it indicates that the new job's initial task assigns its own data segment. Refer to the Guide to the iRMX® I Interactive Configuration Utility and the iRMX® I Interactive Configuration Utility Reference Manual for more information about data segment allocation.
CREATE$JOB

stack$ptr

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

- If the base portion is not NIL or zero, the pointer points to the base of the user-provided stack of the new job's initial task.
- If the base portion is NIL or zero, 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. The Nucleus increases specified values that are not multiples of 16 up to the next higher multiple of 16. Refer to the iRMX® I Programming Techniques Manual for further information on estimating stack sizes.

If you set the stack$ptr parameter to indicate a user-provided stack, setting the stack$size parameter causes the Nucleus to fill the user-provided stack with special characters which the iRMX 86 Debugger uses to detect stack overflow. Because of this situation, never specify a stack$size value that is larger than size of the user-provided stack.

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:

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

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 I Operating System will return the condition code generated by this system call.
**Description**

The CREATEJOB 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 *iRMX®/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.

**Example**

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

DECLARE TOKEN LITERALLY 'SELECTOR';
   /* if your PL/M compiler does not support this variable type, declare TOKEN a WORD */

/* NUCLUS.EXT declares all nucleus system calls */
$INCLUDE(:RMX: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 = 0;  /* 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 = 0AH;  /* 0AH tasks can exist simultaneously */
    max$priority = 0;  /* inherit max priority of parent */
    except$handler = NIL;  /* use system default except handler */
    job$flags = 0;  /* no flags set */
    task$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-86 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-86 Statements
    *

END SAMPLEPROCEDURE;
### Condition Codes

<table>
<thead>
<tr>
<th>Code</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>E$OK</td>
<td>0000H</td>
<td>No exceptional conditions.</td>
</tr>
<tr>
<td>E$CONTEXT</td>
<td>0005H</td>
<td>The job containing the calling task is in the process of being deleted.</td>
</tr>
<tr>
<td>E$EXIST</td>
<td>0006H</td>
<td>The param$obj parameter is not SELECTOR$OF(NIL) or zero and is not a token for an existing object.</td>
</tr>
<tr>
<td>E$LIMIT</td>
<td>0004H</td>
<td>At least one of the following is true:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• max$objects is larger than the unused portion of the object allotment in the calling task’s job.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• max$tasks is larger than the unused portion of the task allotment in the calling task’s job.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• max$priority is greater (numerically smaller) than the maximum allowable task priority in the calling task’s job.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• directory$size is larger than 0FF0H.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• The initial task would exceed the object limit in the new job. That is, the max$objects parameter is set to zero.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• The initial task would exceed the task limit in the new job. The max$tasks parameter is set to zero.</td>
</tr>
<tr>
<td>E$MEM</td>
<td>0002H</td>
<td>At least one of the following is true:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• The memory available to the new job is not sufficient to create a job descriptor (an internal data structure) and the object directory.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• The memory available to the new job is not sufficient to satisfy the pool$min parameter.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• The memory available to the new job is not sufficient to create the task as specified.</td>
</tr>
</tbody>
</table>
CREATE$JOB

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.
- `pool$max` is zero.
- the exception handler mode is not valid.

Nucleus System Calls
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:

<table>
<thead>
<tr>
<th>Bits</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>15-5</td>
<td>Reserved bits which should be set to zero.</td>
</tr>
<tr>
<td>4-1</td>
<td>A value that, when multiplied by four, specifies the number of objects that can be queued on the high performance object queue. Additional objects are queued on the slower, overflow queue. Four is the minimum size for the high performance queue; that is, specifying zero or one in these bits results in a high performance queue that holds four objects.</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Value</th>
<th>Queuing Scheme</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>First-in/first-out</td>
</tr>
<tr>
<td>1</td>
<td>Priority based</td>
</tr>
</tbody>
</table>

### 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 I Operating System will return the condition code generated by this system call.
CREATE$MAILBOX

Description

This system call creates a mailbox, an exchange that tasks can use to exchange tokens for objects. To send an object, 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.

When you set up a mailbox, 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.

When more objects than the high-performance queue can hold are queued at a mailbox, the objects overflow into a slower queue whose size is limited only by the amount of memory in the job containing the mailbox. No space is allocated to the overflow queue until the space is needed to contain objects.

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.
Example

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

DECLARE   TOKEN    LITERALLY 'SELECTION';
/* if your PL/M compiler does not support this variable type,    *
declarate TOKEN a WORD */

/* NUCLUS.EXT declares all nucleus system calls */
$INCLUDE(:RMX:INC/NUCLUS.EXT)

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

SAMPLEPROCEDURE:
PROCEDURE;

mbx$flags = 0;   /* designates four objects to be queued on the high performance object queue; designates a first-in/first-out task queue. */

•

• Typical PL/M-86 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-86 Statements

•

END SAMPLEPROCEDURE;
**CREATE$MAILBOX**

**Condition Codes**

<table>
<thead>
<tr>
<th>Code</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>E$OK</td>
<td>0000H</td>
<td>No exceptional conditions.</td>
</tr>
<tr>
<td>E$LIMIT</td>
<td>0004H</td>
<td>The calling task's job has already reached its object limit.</td>
</tr>
<tr>
<td>E$MEM</td>
<td>0002H</td>
<td>The memory available to the calling task's job is not sufficient to create a mailbox.</td>
</tr>
<tr>
<td>E$NOT$CONFIGURED</td>
<td>0008H</td>
<td>This system call is not part of the present configuration.</td>
</tr>
</tbody>
</table>
The CREATE$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 I Operating System will return the condition code generated by this system call.

**Description**

The CREATE$REGION system call creates a region and returns a token for the region.
CREATE$REGION

Example

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

DECLARE TOKEN LITERALLY 'SELECTOR';
/* if your PL/M compiler does not support this variable type,
declare TOKEN a WORD */

/* NUCLUS.EXT declares all nucleus system calls */
$INCLUDE(:RMX: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-86 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-86 Statements

•

END SAMPLEPROCEDURE;
### Condition Codes

<table>
<thead>
<tr>
<th>Code</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>E$OK</td>
<td>0000H</td>
<td>No exceptional conditions.</td>
</tr>
<tr>
<td>E$LIMIT</td>
<td>0004H</td>
<td>The calling task’s job has reached its object limit.</td>
</tr>
<tr>
<td>E$MEM</td>
<td>0002H</td>
<td>The memory pool of the calling task’s job does not contain a sufficiently large block to satisfy the request.</td>
</tr>
<tr>
<td>E$NOT$CONFIGURED</td>
<td>0008H</td>
<td>This system call is not part of the present configuration.</td>
</tr>
</tbody>
</table>
The CREATE$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 the size parameter is not a multiple of 16, it will be rounded up to the nearest higher multiple of 16 before the request is processed by the Nucleus.
- 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 I 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 a pointer by setting the base portion equal to the segment's TOKEN and the offset portion equal to zero. If you have a PL/M-86 compiler that supports the SELECTOR data type, you can accomplish the same thing by basing the array or structure on the SELECTOR.
Example

MAINPROC: DO;
************************************************************************
* This example illustrates how the CREATE$SEGMENT system call can be used. *
************************************************************************/
DECLARE TOKEN LITERALLY 'SELECTOR';
/* if your PL/M compiler does not support this variable type, declare TOKEN a WORD */

/* NUCLUS.EXT declares all nucleus system calls */
$INCLUDE(:RMX:INC/NUCLUS.EXT)

DECLARE seg TOKEN;
DECLARE seg WORD;
DECLARE status WORD;

SAMPLEPROCEDURE:
PROCEDURE;

seg size = 0100H;
/* the size of the requested segment is 256 bytes */

Typical PL/M-86 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-86 Statements

END SAMPLEPROCEDURE;
END MAINPROC;
## CREATE$SEGMENT

### Condition Codes

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>E$OK</td>
<td>0000H  No exceptional conditions.</td>
</tr>
<tr>
<td>E$LIMIT</td>
<td>0004H  The calling task's job has already reached its object limit.</td>
</tr>
<tr>
<td>E$MEM</td>
<td>0002H  The memory available to the calling task's job is not sufficient to create a segment of the specified size.</td>
</tr>
<tr>
<td>E$NOT$CONFIGURED</td>
<td>0008H This system call is not part of the present configuration.</td>
</tr>
</tbody>
</table>
The CREATE$SEMAPHORE system call creates a semaphore.

\[
\text{semaphore} = \text{RQ$CREATE$SEMAPHORE} (\text{initial$value}, \text{max$value}, \\
\text{semaphore$flags, except$ptr});
\]

**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:

<table>
<thead>
<tr>
<th>Value</th>
<th>Queuing Scheme</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>First-in/first-out</td>
</tr>
<tr>
<td>1</td>
<td>Priority based</td>
</tr>
</tbody>
</table>

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 I 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'; /* if your PL/M compiler does not support this variable type, declare TOKEN a WORD */

/* NUCLUS.EXT declares all nucleus system calls */
$INCLUDE(:RMX: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-86 Statements

sem$token = RQ$CREATE$SEMAPHORE (init$value,
max$value,
sem$flags,
@status);

* Typical PL/M-86 Statements

END SAMPLEPROCEDURE;

40 Nucleus System Calls
## Condition Codes

<table>
<thead>
<tr>
<th>Code</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>E$OK</td>
<td>0000H</td>
<td>No exceptional conditions.</td>
</tr>
<tr>
<td>E$LIMIT</td>
<td>0004H</td>
<td>The calling task's job has already reached its object limit.</td>
</tr>
<tr>
<td>E$MEM</td>
<td>0002H</td>
<td>The memory available to the calling task's job is not sufficient to create a semaphore.</td>
</tr>
<tr>
<td>E$NOT$CONFIGURED</td>
<td>0008H</td>
<td>This system call is not part of the present configuration.</td>
</tr>
<tr>
<td>E$PARAM</td>
<td>8004H</td>
<td>At least one of the following is true:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• The initial$value parameter is larger than the max$value parameter.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• The max$value parameter is 0.</td>
</tr>
</tbody>
</table>
CREATE$TASK

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$seg**: A TOKEN that specifies the new task’s data segment.
  - If not SELECTOR$OF(NIL) or zero, the TOKEN contains the base address of the new task’s data segment.
  - If set to SELECTOR$OF(NIL) or zero, the TOKEN indicates that the new task assigns its own data segment. Refer to the Guide To The iRMX® I Interactive Configuration Utility and the iRMX® I Interactive Configuration Utility Reference Manual for further information on data segment allocation.

- **stack$ptr**: A POINTER that specifies the location of the stack for the new task.
  - If the base portion is not NIL or zero, the Nucleus uses the sum of the offset portion and the stack$size parameter (declared during the call to CREATE$TASK) as the value of the SP register (the stack pointer).
  - If the base portion is NIL or zero, the Nucleus allocates a stack to the new task. The length of the stack is equal to the value of the stack$size parameter.
CREATE$TASK

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. 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 bytes if the new task is going to make Nucleus system calls. Refer to the iRMX® I Programming Techniques Manual for further information on assigning stack sizes.

If you set the stack$ptr parameter to indicate a user-provided stack, setting the stack$size parameter causes the Nucleus to fill the user-provided stack with special characters which the iRMX I Debugger uses to detect stack overflow. Because of this situation, never specify a stack$size value that is larger than size of the user-provided stack.

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:

<table>
<thead>
<tr>
<th>Bits</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>15-1</td>
<td>Reserved bits which should be set to zero</td>
</tr>
<tr>
<td>0</td>
<td>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.</td>
</tr>
</tbody>
</table>

Output Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>task</td>
<td>A TOKEN to which the operating system will return a token for the new task.</td>
</tr>
<tr>
<td>except$ptr</td>
<td>A POINTER to a WORD to which the iRMX I Operating System will return the condition code generated by this system call.</td>
</tr>
</tbody>
</table>
CREATE$TASK

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';
/* if your PL/M compiler does not support this variable type, declare TOKEN a WORD */

/* NUCLUS.EXT declares all nucleus system calls */
$INCLUDE(:RMX:INC/NUCLUS.EXT)

TASKCODE: PROCEDURE EXTERNAL;
END TASKCODE;

DECLARE task$token TOKEN;
DECLARE priority$level$66 LITERALLY '66';
DECLARE start$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-86 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-86 Statements

•

END SAMPLEPROCEDURE;
CREATE$TASK

Condition Codes

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>E$OK</td>
<td>0000H No exceptional conditions.</td>
</tr>
<tr>
<td>E$LIMIT</td>
<td>0004H The calling task’s job has already reached its object limit or task limit.</td>
</tr>
<tr>
<td>E$MEM</td>
<td>0002H The memory available to the calling task’s job is not sufficient to create a task as specified (task descriptor, stack, and possibly NPX area).</td>
</tr>
<tr>
<td>E$NOT$CONFIGURED</td>
<td>0008H This system call is not part of the present configuration.</td>
</tr>
<tr>
<td>E$PARAM</td>
<td>8004H At least one of the following is true:</td>
</tr>
<tr>
<td></td>
<td>• The stack$size parameter is less than 16.</td>
</tr>
<tr>
<td></td>
<td>• The priority parameter is nonzero and greater (numerically smaller) than the maximum allowable priority for tasks in the calling task’s job.</td>
</tr>
</tbody>
</table>
The DELETE$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 I 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 the "Initialization" section of Chapter 10 in the iRMX® I Nucleus User's Guide.
## Condition Codes

<table>
<thead>
<tr>
<th>Code</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>E$OK</td>
<td>0000H</td>
<td>No exceptional conditions.</td>
</tr>
<tr>
<td>E$CONTEXT</td>
<td>0005H</td>
<td>The extension type does not match the composite parameter.</td>
</tr>
<tr>
<td>E$EXIST</td>
<td>0006H</td>
<td>One or both of the extension or composite parameters is not a token for an existing object.</td>
</tr>
<tr>
<td>E$MEM</td>
<td>0002H</td>
<td>The memory available to the calling task's job is not sufficient to complete this operation.</td>
</tr>
<tr>
<td>E$NOT$CONFIGURED</td>
<td>0008H</td>
<td>This system call is not part of the present configuration.</td>
</tr>
<tr>
<td>E$TYPE</td>
<td>8002H</td>
<td>One or both of the extension or composite parameters is a token for an object that is not of the correct type.</td>
</tr>
</tbody>
</table>
The DELETE$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).

```plaintext
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 I 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';
    /* if your PL/M compiler does not support this variable type, declare TOKEN a WORD */

/* NUCLUS.EXT declares all nucleus system calls */
$INCLUDE(:RMX: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 */

    ext$token = RQ$CREATE$EXTENSION (type$code,
                                    delete$mbx$token,
                                    @status);

    •
    •
    •

    Typical PL/M-86 Statements

CALL RQ$DELETE$EXTENSION (ext$token,
                           @status);

END SAMPLEPROCEDURE;
Condition Codes

<table>
<thead>
<tr>
<th>Condition Code</th>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>E$OK</td>
<td>0000H</td>
<td>No exceptional conditions.</td>
</tr>
<tr>
<td>E$EXIST</td>
<td>0006H</td>
<td>The extension parameter is not a token for an existing object.</td>
</tr>
<tr>
<td>E$MEM</td>
<td>0002H</td>
<td>The memory available to the calling task's job is not sufficient to complete this operation.</td>
</tr>
<tr>
<td>E$NOT$CONFIGURED</td>
<td>0008H</td>
<td>This system call is not part of the present configuration.</td>
</tr>
<tr>
<td>E$TYPE</td>
<td>8002H</td>
<td>The extension parameter is a token for an object that is not an extension object.</td>
</tr>
</tbody>
</table>
DELETE$JOB

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) or zero specifies the calling task's job.

Output Parameter

except$ptr

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

Description

The DELETE$JOB system call deletes the specified job, all of the job's tasks, and all objects created by the tasks. Exceptions are that jobs and extension objects (see the iRMX® I Nucleus User's Guide) created by tasks in the target job must be deleted prior to the call to DELETE$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';
/* if your PL/M compiler does not support this variable type, declare TOKEN a WORD */

/* NUCLUS.EXT declares all nucleus system calls */
$INCLUDE(:RMX:INC/NUCLUS.EXT)

DECLARE calling$tasks$job LITERALLY 'SELECTOR$OF(NIL)';
DECLARE status WORD;

SAMPLEPROCEDURE:
   PROCEDURE;

   • Typical PL/M-86 Statements
   •

************************************************************************
* If you set the selection 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

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>E$OK 0000H</td>
<td>No exceptional conditions.</td>
</tr>
<tr>
<td>E$CONTEXT 0005H</td>
<td>At least one of the following is true:</td>
</tr>
<tr>
<td></td>
<td>• There are undeleted jobs or extension objects (see the iRMX® I Nucleus User's Guide) which have been created by tasks in the target job.</td>
</tr>
<tr>
<td></td>
<td>• The deleting task has access to data guarded by a region contained in the job to be deleted. (Refer to the iRMX® I Nucleus User's Guide for information concerning regions.)</td>
</tr>
<tr>
<td>E$EXIST 0006H</td>
<td>The job parameter is not a token for an existing object.</td>
</tr>
<tr>
<td>E$NOT$CONFIGURED 0008H</td>
<td>This system call is not part of the present configuration.</td>
</tr>
<tr>
<td>E$TYPE 8002H</td>
<td>The job parameter is a token for an object that is not a job.</td>
</tr>
</tbody>
</table>
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 I 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 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'; /* if your PL/M compiler does not 
/* if your PL/M compiler does not support this variable type, 

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

SAMPLEPROCEDURE:
PROCEDURE;

mbx$flags = 0; /* designates four objects to be queued 
/* designates four objects to be queued on the high performance object 
queue; designates a first-in/first-out task queue */ 

*/ Typical PL/M-86 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-86 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-86 Statements

•

END SAMPLEPROCEDURE;

Condition Codes

<table>
<thead>
<tr>
<th>Code</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>E$OK</td>
<td>0000H</td>
<td>No exceptional conditions.</td>
</tr>
<tr>
<td>E$EXIST</td>
<td>0006H</td>
<td>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.</td>
</tr>
<tr>
<td>E$NOT$CONFIGURED</td>
<td>0008H</td>
<td>This system call is not part of the present configuration.</td>
</tr>
<tr>
<td>E$TYPE</td>
<td>8002H</td>
<td>The mailbox parameter is a token for an object which is not a mailbox.</td>
</tr>
</tbody>
</table>
DELETE$REGION

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 I 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. *
************************************************************************/

DECLARE TOKEN LITERALLY 'SELECTOR';
   /* if your PL/M compiler does not support this variable type,
      declare TOKEN a WORD */

/* NUCLUS.EXT declares all nucleus system calls */
$INCLUDE(:RMX: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-86 Statements

   • Typical PL/M-86 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-86 Statements

   • Typical PL/M-86 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-86 Statements

END SAMPLEPROCEDURE;
### Condition Codes

<table>
<thead>
<tr>
<th>Code</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ESOK</td>
<td>0000H</td>
<td>No exceptional conditions.</td>
</tr>
<tr>
<td>E$CONTEXT</td>
<td>0005H</td>
<td>The deletion is being requested by a task that currently holds access to data protected by the region.</td>
</tr>
<tr>
<td>E$EXIST</td>
<td>0006H</td>
<td>The region parameter is not a token for an existing object.</td>
</tr>
<tr>
<td>E$NOT$CONFIGURED</td>
<td>0008H</td>
<td>This system call is not part of the present configuration.</td>
</tr>
<tr>
<td>E$TYPE</td>
<td>8002H</td>
<td>The region parameter is a token for an object that is not a region.</td>
</tr>
</tbody>
</table>
The DELETE$SEGMENT system call deletes a segment.

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

**Input Parameter**

segment  
A TOKEN for the segment to be deleted.

**Output Parameter**

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

**Description**

The DELETE$SEGMENT system call deletes iRMX I segments created via CREATE$SEGMENT. When deleting iRMX I 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.
Example

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

DECLARE TOKEN LITERALLY 'SELECTOR';
    /* if your PL/M compiler does not 
support this variable type, 
declare TOKEN a WORD */

/* NUCLUS.EXT declares all nucleus system calls */
$INCLUDE(:RMX: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-86 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-86 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-86 Statements

END SAMPLEPROCEDURE;
Condition Codes

<table>
<thead>
<tr>
<th>Condition Codes</th>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>E$OK</td>
<td>0000H</td>
<td>No exceptional conditions.</td>
</tr>
<tr>
<td>E$EXIST</td>
<td>0006H</td>
<td>At least one of the following is true:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• The segment parameter is not a token for an existing object.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• The segment parameter represents a segment whose job is being deleted.</td>
</tr>
<tr>
<td>E$NOT$CONFIGURED</td>
<td>0008H</td>
<td>This system call is not part of the present configuration.</td>
</tr>
<tr>
<td>E$TYPE</td>
<td>8002H</td>
<td>The segment parameter is a token for an object that is not a segment.</td>
</tr>
</tbody>
</table>
The **DELETE$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 I 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';
/* if your PL/M compiler does not support this variable type, declare TOKEN a WORD */

/* NUCLUS.EXT declares all nucleus system calls */
$INCLUDE(:RMX: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 */

sem$token = RQ$CREATE$SEMAPHORE (init$value,
max$value,
sem$flags,
@status);

Nucleus System Calls 65
DELETE$SEMAPHORE

/************************************************************************
* 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-86 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.

E$NOT$CONFIGURED 0008H This system call is not part of the present configuration.

E$TYPE 8002H The semaphore parameter is a token for an object that is not a semaphore.
The DELETE$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 not SELECTOR$OF(NIL) or zero, the TOKEN must contain a token for the task to be deleted.
- If SELECTOR$OF(NIL) or zero, this parameter indicates that the calling task should be deleted.

**Output Parameter**

**except$ptr**

A POINTER to a WORD to which the iRMX I 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.
DELETE$TASK

Example

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

DECLARE TOKEN LITERALLY 'SELECTOR';  /* if your PL/M compiler does not support this variable type, 
   declare TOKEN a WORD */

/* NUCLUS.EXT declares all nucleus system calls */
$INCLUDE(:RMX:INC/NUCLUS.EXT)

TASKCODE: PROCEDURE EXTERNAL;
END TASKCODE;

DECLARE task$token TOKEN;
DECLARE priority$level$66 LITERALLY '66';
DECLARE start$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-86 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$66,
start$address,
data$seg,
stack$pointer,
stack$size$512,
task$flags,
@status);

•
•

Typical PL/M-86 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-86 Statements

•

END SAMPLEPROCEDURE;
## Condition Codes

<table>
<thead>
<tr>
<th>Code</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>E$OK</td>
<td>0000H</td>
<td>No exceptional conditions.</td>
</tr>
<tr>
<td>E$CONTEXT</td>
<td>0005H</td>
<td>The task parameter is a token for an interrupt task.</td>
</tr>
<tr>
<td>E$EXIST</td>
<td>0006H</td>
<td>One of the following conditions has occurred:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- The task parameter is not a token for an existing object.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- The task parameter represents a task whose job is being deleted.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- More than one task is trying to delete a task which is in a region.</td>
</tr>
<tr>
<td>E$NOT$CONFIGURED</td>
<td>0008H</td>
<td>This system call is not part of the present configuration.</td>
</tr>
<tr>
<td>E$TYPE</td>
<td>8002H</td>
<td>The task parameter is a token for an object which is not a task.</td>
</tr>
</tbody>
</table>
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):

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>15-7</td>
<td>Reserved bits that should be set to zero.</td>
</tr>
<tr>
<td>6-4</td>
<td>First digit of the interrupt level (0-7).</td>
</tr>
<tr>
<td>3</td>
<td>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.</td>
</tr>
<tr>
<td>2-0</td>
<td>Second digit of the interrupt level (0-7), if bit 3 is zero.</td>
</tr>
</tbody>
</table>

### Output Parameter

**except$ptr**

A POINTER to a WORD to which the iRMX I 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 ESCONTEXT exception code.

You must not disable the level reserved for the system clock. You determine this level during system configuration (refer to the iRMX® I 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';
/* if your PL/M compiler does not support this variable type, 
declare TOKEN a WORD */

/* NUCLUS.EXT declares all nucleus system calls */
$INCLUDE(:RMX:INC/NUCLUS.EXT)

INTERRUPTHANDLER: PROCEDURE INTERRUPT 63 EXTERNAL;
END INTERRUPTHANDLER;

DECLARE interrupt$level$7 LITERALLY '0000000001111000B';
/* specifies master interrupt level 7 */
DECLARE interrupt$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 */
interrupt$handler = INTERRUPT$PTR (INTERRUPTHANDLER);
/* points to first instruction of interrupt handler */

• Typical PL/M-86 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$flag, interrupt$handler, data$segment, @status);
```

Typical PL/M-86 Statements

```
CALL RQ$DISABLE (interrupt$level$7, @status);
```

END SAMPLEPROCEDURE;

**Condition Codes**

<table>
<thead>
<tr>
<th>Condition Code</th>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>E$OK</td>
<td>0000H</td>
<td>No exceptional conditions.</td>
</tr>
<tr>
<td>E$CONTEXT</td>
<td>0005H</td>
<td>The level indicated by the level parameter is already disabled or has no interrupt handler assigned to it.</td>
</tr>
<tr>
<td>E$NOT$CONFIGURED</td>
<td>0008H</td>
<td>This system call is not part of the present configuration.</td>
</tr>
<tr>
<td>E$PARAM</td>
<td>8004H</td>
<td>The level parameter is invalid.</td>
</tr>
</tbody>
</table>
DISABLE$DELETION

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 iRMX I 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

```plm
/* This example illustrates how the DISABLE$DELETE system call can be used to make an object immune to ordinary deletion. */

DECLARE TOKEN LITERALLY 'SELECTOR';
  /* if your PL/M compiler does not support this variable type, declare TOKEN a WORD */

/* NUCLUS.EXT declares all nucleus system calls */
$INCLUDE(:RMX:INC/NUCLUS.EXT)

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

SAMPLEPROCEDURE:
  PROCEDURE;
    •
      • Typical PL/M-86 Statements
    •

/* In this example the calling task will be the object to become immune to ordinary 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-86 Statements
    •

/* Using its own token, the calling task invokes the DISABLE$DELETE system call to increase its own disabling depth by one. This makes the calling task immune to ordinary deletion. */
```
DISABLE$DELETION

CALL RQ$DISABLE$DELETION (task$token, @status);

•
•
• Typical PL/M-86 Statements

END SAMPLEPROCEDURE;

Condition Codes

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>E$OK 0000H</td>
<td>No exceptional conditions.</td>
</tr>
<tr>
<td>E$EXIST 0006H</td>
<td>The object parameter is not a token for an existing object.</td>
</tr>
<tr>
<td>E$LIMIT 0004H</td>
<td>The object's disabling depth is already 255.</td>
</tr>
<tr>
<td>E$NOT$CONFIGURED 0008H</td>
<td>This system call is not part of the present configuration.</td>
</tr>
</tbody>
</table>
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):

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>15-7</td>
<td>Reserved bits that should be set to zero.</td>
</tr>
<tr>
<td>6-4</td>
<td>First digit of the interrupt level (0-7).</td>
</tr>
</tbody>
</table>
| 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 I 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';
    /* if your PL/M compiler does not support this variable type, declare TOKEN a WORD */

/* NUCLUS.EXT declares all nucleus system calls */
$INCLUDE(:RMX:INC/NUCLUS.EXT)

INTERRUPTHANDLER: PROCEDURE INTERRUPT 63 EXTERNAL;
END INTERRUPTHANDLER;

DECLARE interrupt$level$7 LITERALLY '000000001111000B';
    /* specifies master interrupt level */
DECLARE interrupt$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 */

interrupt$handler = INTERRUPT$PTR (INTERRUPTHANDLER);  /* points to first instruction of interrupt handler */

• Typical PL/M-86 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$flag,
     interrupt$handler,
     data$segment,
     @status);
```

Typical PL/M-86 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-86 Statements

END SAMPLEPROCEDURE;
## Condition Codes

<table>
<thead>
<tr>
<th>Code</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>E$OK</td>
<td>0000H</td>
<td>No exceptional conditions.</td>
</tr>
<tr>
<td>E$CONTEXT</td>
<td>0005H</td>
<td>At least one of the following is true:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• A non-interrupt task tried to enable a level that was already enabled.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• There is not an interrupt handler assigned to the specified level.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• There has been an interrupt overflow on the specified level.</td>
</tr>
<tr>
<td>E$NOT$CONFIGURED</td>
<td>0008H</td>
<td>This system call is not part of the present configuration.</td>
</tr>
<tr>
<td>E$PARAM</td>
<td>8004H</td>
<td>The level parameter is invalid.</td>
</tr>
</tbody>
</table>
The ENABLE$DELETION system call enables the deletion of objects that have had deletion disabled.

**CAUTION**

Human Interface applications should not use the DISABLE$DELETION system call, and therefore, have no need to use the ENABLE$DELETION and FORCE$DELETE system calls. This is because DISABLE$DELETION increases the disabling depth of an object. A Human Interface application containing objects whose disabling depths are greater than one cannot be deleted asynchronously (via a CONTROL-C entered at a terminal).

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 iRMX I 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.
ENABLE$DELETION

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';

/************************************************************************
* if your PL/M compiler does not support this variable type, declare TOKEN a WORD */
************************************************************************/

/* NUCLUS.EXT declares all nucleus system calls */
$INCLUDE(:RMX:INC/NUCLUS.EXT)

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

SAMPLEPROCEDURE:
PROCEDURE;

•
• Typical PL/M-86 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-86 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-86 Statements

Nucleus System Calls
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-86 Statements

END SAMPLEPROCEDURE;

Condition Codes

- E$OK 0000H No exceptional conditions.
- E$CONTEXT 0005H The object's deletion is not disabled.
- E$EXIST 0006H The object parameter is not a token for an existing object.
- E$NOT$CONFIGURED 0008H This system call is not part of the present configuration.
END$INIT$TASK

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 of a first-level job 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 iRMX® I 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):
  
<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>15-7</td>
<td>Reserved bits that should be set to zero.</td>
</tr>
<tr>
<td>6-4</td>
<td>First digit of the interrupt level (0-7).</td>
</tr>
<tr>
<td>3</td>
<td>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.</td>
</tr>
<tr>
<td>2-0</td>
<td>Second digit of the interrupt level (0-7), if bit 3 is zero</td>
</tr>
</tbody>
</table>

**Output Parameter**

- **except$ptr**: A POINTER to a WORD to which the iRMX I 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';
   /* if your PL/M compiler does not
      support this variable type,
      declare TOKEN a WORD */
/
/* NUCLUS.EXT declares all nucleus system calls */
$INCLUDE(:RMX:INC/NUCLUS.EXT)

DECLARE the$first$word WORD;
DECLARE E$OK LITERALLY '00H';
DECLARE interrupt$level$7 LITERALLY '000000001111000B';
   /* 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 59 PUBLIC;   /* 59 is a
   placeholder value. ENTER$INTERRUPT establishes the actual level. */

  

*  Typical PL/M-86 Statements

  

86  Nucleus System Calls
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-86 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$status);
END INTERRUPT$HANDLER;

INLINEERRORPROCESS: PROCEDURE (int$status);
DECLARE int$status WORD;

IF int$status <> E$OK THEN
DO;
  •
  • Typical PL/M-86 Statements
  •
END;
END INLINEERRORPROCESS;

SAMPLEPROCEDURE: PROCEDURE;
    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$handler = INTERRUPT$PTR (INTERRUPTHANDLER);
    /* points to the first instruction of the interrupt handler */
    interrupt$task$flag = 0; /* indicates no interrupt task on level 7 */
ENTER$INTERRUPT

•

• Typical PL/M-86 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$flag,
interrupt$handler,
data$segment,
@status);

•

• Typical PL/M-86 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$INTERRUPT.
E$NOT$CONFIGURED 0008H  This system call is not included in the present configuration.
E$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):
  
<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>15-7</td>
<td>Reserved bits that should be set to zero.</td>
</tr>
<tr>
<td>6-4</td>
<td>First digit of the interrupt level (0-7).</td>
</tr>
<tr>
<td>3</td>
<td>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.</td>
</tr>
<tr>
<td>2-0</td>
<td>Second digit of the interrupt level (0-7), if bit 3 is zero.</td>
</tr>
</tbody>
</table>

**Output Parameter**

- **except$ptr**: A POINTER to a WORD to which the iRMX I 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**

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';
  /* if your PL/M compiler does not support this variable type, declare TOKEN a WORD */

/* NUCLUS.EXT declares all nucleus system calls */
$INCLUDE(:RMX:INC/NUCLUS.EXT)

DECLARE 
  interrupt$level$7 LITERALLY '000000001111000B';
  /* specifies master interrupt level 7 */
  E$OK LITERALLY '00h';
  interrupt$task$flag BYTE;
  interrupt$handler POINTER;
  data$segment TOKEN;
  status WORD;
  interrupt$status WORD;

INTERRUPTHANDLER: PROCEDURE INTERRUPT 59 PUBLIC; /* 59 is a placeholder value. ENTER$INTERRUPT establishes actual level */

  ... Typical PL/M-86 Statements ...

CALL RQ$EXIT$INTERRUPT (interrupt$level$7, 
  @interrupt$status);

IF interrupt$status <> E$OK THEN DO;
  ... Typical PL/M-86 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 */
   interrupt$handler = INTERRUPT$PTR (INTERRUPTHANDLER);  /* points to the
   first instruction of
   the interrupt handler */
   
   * Typical PL/M-86 Statements
   *
   
   CALL RQ$SET$INTERRUPT (interrupt$level$7,
                        interrupt$task$flag,
                        interrupt$handler,
                        data$segment,
                        @status);
   
   * Typical PL/M-86 Statements
   *
   
END SAMPLEPROCEDURE;

Condition Codes

<table>
<thead>
<tr>
<th>Condition Code</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>E$OK</td>
<td>0000H</td>
<td>No exceptional conditions.</td>
</tr>
<tr>
<td>E$CONTEXT</td>
<td>0005H</td>
<td>The SET$INTERRUPT system call has not been invoked for the specified level.</td>
</tr>
<tr>
<td>E$NOT$CONFIGURED</td>
<td>0008H</td>
<td>This system call is not part of the present configuration.</td>
</tr>
<tr>
<td>E$PARAM</td>
<td>8004H</td>
<td>The level parameter is invalid.</td>
</tr>
</tbody>
</table>

Nucleus System Calls
The FORCE$DELETE system call deletes objects whose disabling depths are zero or one.

**CAUTION**

Human Interface applications should not use the DISABLE$DELETION system call, and therefore, have no need to use the FORCE$DELETE and ENABLE$DELETION system calls. This is because DISABLE$DELETION increases the disabling depth of an object. A Human Interface application containing objects whose disabling depths are greater than one cannot be deleted asynchronously (via a CONTROL-C entered at a terminal).

```
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 must be SELECTOR$OF(NIL) or zero.

- **object**
  - A TOKEN for the object that is to be deleted.

**Output Parameter**

- **except$ptr**
  - A POINTER to a WORD to which the iRMX I 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 an task that has had deletion disabled.

```
DECLARE TOKEN LITERALLY 'SELECTOR';
    /* if your PL/M compiler does not support this variable type,
       declare TOKEN a WORD */

/* NUCLUS.EXT declares all nucleus system calls */
$INCLUDE(:RMX: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 */
```

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$token = RQS$CREATE$SEMAPHORE (init$value,  
    max$value,  
    sem$flags,  
    @status);
```

- Typical PL/M-86 Statements
FORCE$DELETE

/**************************************************************************/
* 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. *
**************************************************************************/

CALL RQ$FORCE$DELETE (SELECTOR$OF(NIL),
                        sem$token,
                        @status);

•
• Typical PL/M-86 Statements

END SAMPLEPROCEDURE;

Condition Codes

<table>
<thead>
<tr>
<th>Code</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>E$OK</td>
<td>0000H</td>
<td>No exceptional conditions.</td>
</tr>
<tr>
<td>E$CONTEXT</td>
<td>0005H</td>
<td>The wrong extension type was used in the extension parameter of the FORCE$DELETE system call.</td>
</tr>
<tr>
<td>E$EXIST</td>
<td>0006H</td>
<td>One or both of the object or extension parameters is not a token for an existing object.</td>
</tr>
<tr>
<td>E$MEM</td>
<td>0002H</td>
<td>The memory available to the calling task's job is not sufficient to complete this call.</td>
</tr>
<tr>
<td>E$NOT$CONFIGURED</td>
<td>0008H</td>
<td>This system call is not part of the present configuration.</td>
</tr>
<tr>
<td>E$TYPE</td>
<td>8002H</td>
<td>The extension parameter is a token for an object that is not an extension object.</td>
</tr>
</tbody>
</table>
The GET$EXCEPTION$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$HANDLER$OFFSET  WORD,
    EXCEPTION$HANDLER$BASE  TOKEN,
    EXCEPTION$MODE  BYTE);
```

where, after the call,

- EXCEPTION$HANDLER$OFFSET contains the offset of the first instruction of the exception handler.
- EXCEPTION$HANDLER$BASE contains a base for the segment containing the first instruction of the exception handler. If exception$handler$base is SELECTOR$OF(NIL) and exception$handler$offset is zero, the calling task’s exception handler is the system default exception handler.
- EXCEPTION$MODE contains an encoded indication of the calling task’s current exception mode. The value is interpreted as follows:

<table>
<thead>
<tr>
<th>Value</th>
<th>When to Pass Control to Exception Handler</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Never</td>
</tr>
<tr>
<td>1</td>
<td>On programmer errors only</td>
</tr>
<tr>
<td>2</td>
<td>On environmental conditions only</td>
</tr>
<tr>
<td>3</td>
<td>On all exceptional conditions</td>
</tr>
</tbody>
</table>

except$ptr
A POINTER to a WORD to which the iRMX I 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'; /* if your PL/M compiler does not support this variable type, declare TOKEN a WORD */

/* NUCLUS.EXT declares all nucleus system calls */
$INCLUDE(:RMX:INC/NUCLUS.EXT)

DECLARE x$handler STRUCTURE (x$handler$offset WORD, x$handler$base TOKEN, x$mode BYTE);

DECLARE status WORD;

SAMPLEPROCEDURE:
PROCEDURE;

•
• Typical PL/M-86 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-86 Statements
•

END SAMPLEPROCEDURE;

Condition Codes

ESOK 0000H No exceptional conditions.

ES$NOT$CONFIGURED 0008H This system call is not part of the present configuration.
The GET$LEVEL system call returns the number of the level of the highest priority interrupt being serviced.

```plaintext
level = RQ$GET$LEVEL (except$ptr);
```

**Output Parameters**

- **level**
  A WORD whose value is interpreted as follows (bit 15 is the high-order bit):

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>15-8</td>
<td>Reserved bits that are set to zero.</td>
</tr>
<tr>
<td>7</td>
<td>If zero, some level is being serviced and bits 6-0 are significant.</td>
</tr>
<tr>
<td></td>
<td>If one, no level is being serviced and bits 6-0 are not significant.</td>
</tr>
<tr>
<td>6-4</td>
<td>First digit of the interrupt level (0-7).</td>
</tr>
<tr>
<td>3</td>
<td>If one, the level is a master level and bits 6-4 specify the entire level number.</td>
</tr>
<tr>
<td></td>
<td>If zero, the level is a slave level and bits 2-0 specify the second digit.</td>
</tr>
<tr>
<td>2-0</td>
<td>Second digit of the interrupt level (0-7), if bit 3 is zero.</td>
</tr>
</tbody>
</table>

- **except$ptr**
  A POINTER to a WORD to which the iRMX I 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';
    /* if your PL/M compiler does not support this variable type, 
    declare TOKEN a WORD */

/* NUCLUS.EXT declares all nucleus system calls */
$INCLUDE(:RMX:INC/NUCLUS.EXT)

DECLARE interrupt$level WORD;
DECLARE status WORD;

SAMPLEPROCEDURE:
    PROCEDURE;
    
        •
        •
            Typical PL/M-86 Statements
        •

        /********************************************************************************
        * The GET$LEVEL system call returns to the calling task the number of *
        * the highest interrupt level being serviced. *
        ********************************************************************************/

        interrupt$level = RQ$GET$LEVEL (@status);

        •
        •
            Typical PL/M-86 Statements
        •

    END SAMPLEPROCEDURE;

Condition Codes

<table>
<thead>
<tr>
<th>Condition Code</th>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>E$OK</td>
<td>0000H</td>
<td>No exceptional conditions.</td>
</tr>
<tr>
<td>E$NOT$CONFIGURED</td>
<td>0008H</td>
<td>This system call is not part of the present configuration.</td>
</tr>
</tbody>
</table>
The GET$POOL$ATTRIB system call returns information about the memory pool of the calling task’s job.

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$ptr  A POINTER to a WORD to which the iRMX I Operating System will return the condition code generated by this system call.
GET$POOL$ATTRIB

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.

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';
   /* if your PL/M compiler does not */
   /* support this variable type, */
   declare TOKEN a WORD */

/* NUCLUS.EXT declares all nucleus system calls */
$INCLUDE(:RMX: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-86 Statements
   •
GET$POOL$ATTRIB

/************************************************************************
* 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-86 Statements

END SAMPLEPROCEDURE;

Condition Codes

E$OK 0000H  No exceptional conditions.
E$NOT$CONFIGURED 0008H  This system call is not part of the present configuration.
The GET$PRIORITY system call returns the priority of a task.

\[
priority = \text{RQ$GET$PRIORITY (task, except$ptr)};\]

### Input Parameter

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
</table>
| task      | A TOKEN that specifies the task whose priority is being requested.  
            - If not SELECTOR$OF(NIL) or zero, the TOKEN must contain a token for the task whose priority is being requested.  
            - If SELECTOR$OF(NIL) or zero, the calling task is asking for its own priority. |

### Output Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>priority</td>
<td>A BYTE in which the system call returns the priority of the task indicated by the task parameter.</td>
</tr>
<tr>
<td>except$ptr</td>
<td>A POINTER to a WORD to which the iRMX I Operating System will return the condition code generated by this system call.</td>
</tr>
</tbody>
</table>

### Description

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

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

DECLARE TOKEN LITERALLY 'SELECTOR'; /* if your PL/M compiler does not support this variable type,
declare TOKEN a WORD */

/* NUCLUS.EXT declares all nucleus system calls */
$INCLUDE(:RMX:INC/NUCLUS.EXT)

DECLARE priority BYTE;
DECLARE calling$tasks$priority LITERALLY 'SELECTOR$OF(NIL)';
DECLARE status WORD;

SAMPLEPROCEDURE:
PROCEDURE;

* Typical PL/M-86 Statements

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

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

* Typical PL/M-86 Statements

END SAMPLEPROCEDURE;
GET$PRIORITY

Condition Codes

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>E$OK</td>
<td>0000H  No exceptional conditions.</td>
</tr>
<tr>
<td>E$EXIST</td>
<td>0006H  The task parameter is not a token for an existing object.</td>
</tr>
<tr>
<td>E$NOT$CONFIGURED</td>
<td>0008H  This system call is not part of the present configuration.</td>
</tr>
<tr>
<td>E$TYPE</td>
<td>8002H  The task parameter is a token for an object that is not a task.</td>
</tr>
</tbody>
</table>
The GET$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 I Operating System will return the condition code generated by this system call.

### Description

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

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

DECLARE TOKEN LITERALLY 'SELECTOR';
    /* if your PL/M compiler does not support this variable type,
       declare TOKEN a WORD */

/* NUCLUS.EXT declares all nucleus system calls */
$INCLUDE(:RMX:INC/NUCLUS.EXT)

DECLARE mbx$token TOKEN;
DECLARE calling$tasks$job LITERALLY 'SELECTOR$OF(NIL)';
DECLARE wait$forever LITERALLY 'OFFFH';
DECLARE seg$token TOKEN;
DECLARE response TOKEN;
DECLARE size WORD;
DECLARE status WORD;

SAMPLEPROCEDURE:
PROCEDURE;
   •
   •
   • Typical PL/M-86 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. *
 *************************************************************************/

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

   •
   •
   • Typical PL/M-86 Statements
GET$SIZE

/************************************************************************
* 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-86 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-86 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.
    E$NOT$CONFIGURED  0008H  This system call is not part of the present
                             configuration.
    E$TYPE            8002H  The segment parameter is a token for an object
                             that is not a segment.
GET$TASK$TOKENS

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

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

Input Parameter

<table>
<thead>
<tr>
<th>selection</th>
<th>A BYTE that tells the iRMX I Operating System what information is desired. It is encoded as follows:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>Object for which a Token is Requested</td>
</tr>
<tr>
<td>0</td>
<td>The calling task.</td>
</tr>
<tr>
<td>1</td>
<td>The calling task’s job.</td>
</tr>
<tr>
<td>2</td>
<td>The parameter object of the calling task’s job.</td>
</tr>
<tr>
<td>3</td>
<td>The root job.</td>
</tr>
</tbody>
</table>

Output Parameters

| object    | A TOKEN to which the iRMX I Operating System will return the requested token.                  |
| except$ptr| A POINTER to a WORD to which the iRMX I 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';
    /* if your PL/M compiler does not support this variable type,    *
       declare TOKEN a WORD */

/* NUCLUS.EXT declares all nucleus system calls */
$INCLUDE(:RMX:INC/NUCLUS.EXT)

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

SAMPLEPROCEDURE:
PROCEDURE;
    
    Typical PL/M-86 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-86 Statements
    
END SAMPLEPROCEDURE;

Condition Codes

E$OK 0000H No exceptional conditions.

ESPARAM 8004H The selection parameter is greater than 3.
GET$TYPE

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

\[ \text{type$code} = \text{RQ$GET$TYPE (object, except$ptr)}; \]

**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 I objects are encoded as follows:

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

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

**except$ptr**
A POINTER to a WORD to which the iRMX I 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';
    /* if your PL/M compiler does not support this variable type,
    declare TOKEN a WORD */

/* NUCLUS.EXT declares all nucleus system calls */
$INCLUDE(:RMX:INC/NUCLUS.EXT)

DECLARE type$code WORD;
DECLARE mbx$token TOKEN;
DECLARE calling$tasks$job LITERALLY 'SELECTOR$OF(NIL)';
DECLARE wait$forever LITERALLY 'OFFFH';
DECLARE object$token TOKEN;
DECLARE response TOKEN;
DECLARE status WORD;

SAMPLEPROCEDURE:
    PROCEDURE;

    •
    • Typical PL/M-86 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).                                              *
************************************************************************/

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

    •
    • Typical PL/M-86 Statements
GET$TYPE

/********************_syscallname_*******************
* 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-86 Statements

/********************_*******************************/
* Using the type code returned by the GET$TYPE system call, the *
* calling task can find out if the object is a job, task, *
* mailbox, region, segment, semaphore, port, or extension. *
/********************_*******************************/
type$code = RQ$GET$TYPE (object$token,
@status);

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$NOT$CONFIGURED 0008H This system call is not part of the present configuration.
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$list$ptr 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$used Number of component tokens making up the composite object.
**INSPECT$COMPOSITE**

- **tokens(*)** The tokens that actually constitute the composite object.
- **except$ptr** A POINTER to a WORD to which the iRMX I 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$sots value in the data structure pointed to by the token$list parameter. The Nucleus fills in the remaining fields in that structure. If num$sots is set to zero, the Nucleus will fill in only the num$used field.

If the num$sots value is smaller than the actual number of component tokens, only that number (num$sots) of tokens will be returned.

**Example**

See the "DELETE_RING_BUFFER Procedure" example in the *iRMX® I Nucleus User's Guide.*

**Condition Codes**

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>E$OK</td>
<td>0000H No exceptional conditions.</td>
</tr>
<tr>
<td>E$CONTEXT</td>
<td>0005H The composite parameter is not compatible with the extension parameter.</td>
</tr>
<tr>
<td>E$EXIST</td>
<td>0006H The composite and/or extension parameter(s) is not a token for an existing object.</td>
</tr>
<tr>
<td>E$NOT$CONFIGURED</td>
<td>0008H This system call is not part of the present configuration.</td>
</tr>
<tr>
<td>E$TYPE</td>
<td>8002H One or both of the extension or composite parameters is a token for an object that is not of the correct type.</td>
</tr>
</tbody>
</table>
The LOOKUP$OBJECT system call returns a token for a cataloged object.

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

### Input Parameters
- **job**: A TOKEN indicating the object directory to be searched.
  - If not SELECTOR$OF(NIL) or zero, the TOKEN must contain a token for the job whose object directory is to be searched.
  - If SELECTOR$OF(NIL) or zero, 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 *irMX® I 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 I Operating System will return the condition code generated by this system call.

### Description
The LOOKUP$OBJECT system call returns the token for an object after searching for its name in the specified object directory. Because it is possible that the object is not cataloged at the time of the call, the calling task has the option of waiting, either indefinitely or for a specific period of time, for another task to catalog the object.
LOOKUP$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';
    /* if your PL/M compiler does not
       support this variable type,
       declare TOKEN a WORD */

/* NUCLUS.EXT declares all nucleus system calls */
$INCLUDE(:RMX:INC/NUCLUS.EXT)

DECLARE mbx$token TOKEN;
DECLARE calling$tasks$job LITERALLY 'SELECTOR$OF(NIL)';
DECLARE wait$forever LITERALLY '0FFFF';
DECLARE status WORD;

SAMPLEPROCEDURE:
    PROCEDURE;

      •
      •
      • Typical PL/M-86 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'.
************************************************************************/

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

      •
      • Typical PL/M-86 Statements
      •

END SAMPLEPROCEDURE;


Condition Codes

<table>
<thead>
<tr>
<th>Condition Code</th>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>E$OK</td>
<td>0000H</td>
<td>No exceptional conditions.</td>
</tr>
<tr>
<td>E$CONTEXT</td>
<td>0005H</td>
<td>The specified job has an object directory of size 0.</td>
</tr>
</tbody>
</table>
| E$EXIST        | 0006H| At least one of the following is true:  
- The specified job was deleted while the task was waiting.  
- The job parameter (which is not SELECTOR$OF(NIL) or zero) 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. |
| E$NOT$CONFIGURED | 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. |
OFFSPRING

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) or zero specifies the calling task's job.

Output Parameter

token$list

A TOKEN that indicates the children of the specified job.

- If not SELECTOR$OF(NIL) or zero, 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) or zero, the specified job has no children.

except$ptr

A POINTER to a WORD to which the iRMX I 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';
    /* if your PL/M compiler does not * 
    support this variable type,     *
    declare TOKEN a WORD */

/* NUCLUS.EXT declares all nucleus system calls */
$INCLUDE(:RMX:INC/NUCLUS.EXT)

DECLARE token$list TOKEN;
DECLARE calling$tasks$job LITERALLY 'SELECTOR$OF(NIL)';
DECLARE status WORD;

SAMPLEPROCEDURE:
    PROCEDURE;
    
    Typical PL/M-86 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.     * 
    **********************************************************/

    token$list = RQ$OFFSPRING (calling$tasks$job,
      @status);
    
    Typical PL/M-86 Statements

END SAMPLEPROCEDURE;
## Condition Codes

<table>
<thead>
<tr>
<th>Code</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>E$OK</td>
<td>0000H</td>
<td>No exceptional conditions.</td>
</tr>
<tr>
<td>E$EXIST</td>
<td>0006H</td>
<td>The job parameter is not a token for an existing object.</td>
</tr>
<tr>
<td>E$LIMIT</td>
<td>0004H</td>
<td>The calling task's job has already reached its object limit.</td>
</tr>
<tr>
<td>E$MEM</td>
<td>0002H</td>
<td>The memory available to the specified job is not sufficient to complete this call.</td>
</tr>
<tr>
<td>E$NOT$CONFIGURED</td>
<td>0008H</td>
<td>This system call is not part of the present configuration.</td>
</tr>
<tr>
<td>E$TYPE</td>
<td>8002H</td>
<td>The job parameter contains a token for an object that is not a job.</td>
</tr>
</tbody>
</table>
RECEIVE$CONTROL

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 I 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.
RECEIVE$CONTROL

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';
    /* if your PL/M compiler does not
     * support this variable type,
     * declare TOKEN a WORD */

/* NUCLUS.EXT declares all nucleus system calls */
$INCLUDE(:RMX: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-86 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-86 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-86 Statements
    •

END SAMPLEPROCEDURE;
## Condition Codes

<table>
<thead>
<tr>
<th>Code</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>E$OK</td>
<td>0000H</td>
<td>No exceptional conditions.</td>
</tr>
<tr>
<td>E$CONTEXT</td>
<td>0005H</td>
<td>The region parameter refers to a region already accessed by the calling task.</td>
</tr>
<tr>
<td>E$EXIST</td>
<td>0006H</td>
<td>The region parameter is not a token for an existing object.</td>
</tr>
<tr>
<td>E$NOT$CONFIGURED</td>
<td>0008H</td>
<td>This system call is not part of the present configuration.</td>
</tr>
<tr>
<td>E$TYPE</td>
<td>8002H</td>
<td>The region parameter contains a token for an object that is not a region.</td>
</tr>
</tbody>
</table>
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 0xFFFF, indicates that the task will wait as long as is necessary.
  
  - If between 0 and 0xFFFF, indicates the number of clock intervals that the task is willing to wait. The length of a clock interval is configurable. Refer to the iRMX® I 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 not NIL, 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.

**CAUTION**

Response$ptr points to a location for the sending task to use. If you specify a constant value for response$ptr, be careful to ensure that the value does not conflict with system requirements.

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

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

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';
    /* if your PL/M compiler does not 
       support this variable type, 
       declare TOKEN a WORD */

/* NUCLUS.EXT declares all nucleus system calls */
$INCLUDE(:RMX:INC/NUCLUS.EXT)

DECLARE mbx$token TOKEN;
DECLARE calling$tasks$job LITERALLY 'SELECTOR$OF(NIL)';
DECLARE wait$forever LITERALLY 'OFFFH';
DECLARE seg$token TOKEN;
DECLARE response TOKEN;
DECLARE status WORD;

SAMPLEPROCEDURE:
PROCEDURE;

    •
    •
    •
    •
    
    Typical PL/M-86 Statements

    •
    •

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

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

    •
    •
    •
    •
    
    Typical PL/M-86 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. **/ 

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

• Typical PL/M-86 Statements

END SAMPLEPROCEDURE;

**Condition Codes**

<table>
<thead>
<tr>
<th>Condition Code</th>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>E$OK</td>
<td>0000H</td>
<td>No exceptional conditions.</td>
</tr>
<tr>
<td>E$EXIST</td>
<td>0006H</td>
<td>At least one of the following is true:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• The mailbox parameter is not a token for an existing object.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• The mailbox was deleted while the task was waiting.</td>
</tr>
<tr>
<td>E$NOT$CONFIGURED</td>
<td>0008H</td>
<td>This system call is not part of the present configuration.</td>
</tr>
<tr>
<td>E$TIME</td>
<td>0011H</td>
<td>One of the following is true:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• The calling task was not willing to wait and there was not a token available.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• The task waited in the task queue and its designated waiting period elapsed before the task got the desired token.</td>
</tr>
<tr>
<td>E$TYPE</td>
<td>8002H</td>
<td>The mailbox parameter contains a token for an object that is not a mailbox.</td>
</tr>
</tbody>
</table>
RECEIVE$UNITS

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

\[
\text{value} = \text{RQ$RECEIVE$UNITS} \text{(semaphore, units, time$limit, except$ptr)};
\]

**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 0xFFFF, the WORD indicates that the task will wait as long as is necessary.
  - If between 0 and 0xFFFF, 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 *iRMX® I 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 I Operating System will return the condition code generated by this system call.

**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';  
    /* if your PL/M compiler does not support this variable type, 
       declare TOKEN a WORD */

/* NUCLUS.EXT declares all nucleus system calls */
$INCLUDE(:RMX:INC/NUCLUS.EXT)

DECLARE sem$token            TOKEN;
DECLARE calling$tasks$job    LITERALLY 'SELECTOR$OF(NIL)';
DECLARE wait$forever         LITERALLY 'OFFFH';
DECLARE seg$token            TOKEN;
DECLARE units$remaining      WORD;
DECLARE units$requested      WORD;
DECLARE status               WORD;

SAMPLEPROCEDURE:
    PROCEDURE;

        •

        •

        Typical PL/M-86 Statements

        •

        •

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

sem$token = RQ$LOOKUP$OBJECT (calling$tasks$job, 
    @5,'SEMA4'),
    wait$forever,
    @status);

        •

        •

        Typical PL/M-86 Statements


RECEIVE$UNITS

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

units$remaining = RQ$RECEIVE$UNITS (sem$token, 
units$requested, 
wait$forever, 
@status);

•
•

Typical PL/M-86 Statements

END SAMPLEPROCEDURE;

Condition Codes

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>E$OK</td>
<td>0000H  No exceptional conditions.</td>
</tr>
<tr>
<td>E$EXIST</td>
<td>0006H  At least one of the following is true:</td>
</tr>
<tr>
<td></td>
<td>• The semaphore parameter is not a token for an existing object.</td>
</tr>
<tr>
<td></td>
<td>• The semaphore was deleted while the task was waiting.</td>
</tr>
<tr>
<td>E$LIMIT</td>
<td>0004H  The units parameter is greater than the maximum value specified for</td>
</tr>
<tr>
<td></td>
<td>the semaphore when it was created.</td>
</tr>
<tr>
<td>E$NOT$CONFIGURED</td>
<td>0008H</td>
</tr>
<tr>
<td>E$TIME</td>
<td>0001H  One of the following is true:</td>
</tr>
<tr>
<td></td>
<td>• The calling task was not willing to wait and the requested units were not</td>
</tr>
<tr>
<td></td>
<td>available.</td>
</tr>
<tr>
<td></td>
<td>• The task waited in the task queue and its designated waiting period elapsed</td>
</tr>
<tr>
<td></td>
<td>before the requested units were available.</td>
</tr>
<tr>
<td>E$TYPE</td>
<td>8002H  The semaphore parameter is a token for an object that is not a semaphore.</td>
</tr>
</tbody>
</table>
The RESET$INTERRUPT system call cancels the assignment of an interrupt handler to a level.

CALL RQ$RESET$INTERRUPT (level, except$ptr);

**Input Parameter**

<table>
<thead>
<tr>
<th>level</th>
<th>A WORD specifying an interrupt level. This word must be encoded as follows (bit 15 is the high-order bit):</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>15-7</td>
<td>Reserved bits that should be set to zero.</td>
</tr>
<tr>
<td>6-4</td>
<td>First digit of the interrupt level (0-7).</td>
</tr>
<tr>
<td>3</td>
<td>If one, the level is a master level and bits 6-4 specify the entire level number.</td>
</tr>
<tr>
<td></td>
<td>If zero, the level is a slave level and bits 2-0 specify the second digit.</td>
</tr>
<tr>
<td>2-0</td>
<td>Second digit of the interrupt level (0-7), if bit 3 is zero.</td>
</tr>
</tbody>
</table>

**Output Parameter**

| except$ptr | A POINTER to a WORD to which the iRMX I 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 iRMX® I Interactive Configuration Utility Reference Manual for further information).
Example

This example illustrates how the \texttt{RESET$\text{INTERRUPT}} system call can be used to cancel the assignment of an interrupt handler to an interrupt level.

```plaintext
DECLARE \texttt{TOKEN} \texttt{LITERALLY 'SELECTOR'};
  /* if your PL/M compiler does not support this variable type,
     declare \texttt{TOKEN} a \texttt{WORD} */

/* \texttt{NUCLUS.EXT} declares all nucleus system calls */
$\texttt{INCLUDE(:RMX:INC/NUCLUS.EXT)}$

\texttt{INTERRUPTHANDLER: PROCEDURE INTERRUPT 63 EXTERNAL;}
\texttt{END INTERRUPTHANDLER;}

DECLARE \texttt{task$token} \texttt{TOKEN;}
DECLARE \texttt{priority$level$66} \texttt{LITERALLY '66'};
DECLARE \texttt{start$address} \texttt{POINTER;}
DECLARE \texttt{data$segment} \texttt{TOKEN;}
DECLARE \texttt{stack$pointer} \texttt{POINTER;}
DECLARE \texttt{stack$size$512} \texttt{LITERALLY '512'; /* new task's stack size is 512 bytes*/}

DECLARE \texttt{task$flags} \texttt{WORD;}
DECLARE \texttt{interrupt$level$7} \texttt{LITERALLY '000000001111000B';}
  /* specifies master interrupt level 7 */
DECLARE \texttt{interrupt$task$flag} \texttt{BYTE;}
DECLARE \texttt{interrupt$handler} \texttt{POINTER;}
DECLARE \texttt{interrupt$status} \texttt{WORD;}
DECLARE \texttt{status} \texttt{WORD;}

\texttt{INTERRUPTTASK: PROCEDURE PUBLIC;}

\texttt{interrupt$task$flags = 001H;}  /* indicates that calling task is to be interrupt task */
\texttt{data$segment = SELECTOR$OF(NIL);}  /* use own data segment */
\texttt{interrupt$handler = INTERRUPT$PTR (INTERRUPTHANDLER);}  /* points to the first instruction of the interrupt handler */

/* The first system call in this example, \texttt{SET$\text{INTERRUPT}}, makes the *
  calling task (\texttt{INTERRUPTTASK}) the interrupt task for the interrupt *
  level. */

CALL \texttt{RQ$SET$\text{INTERRUPT}} (interrupt$level$7, interrupt$task$flag, 
interrupt$handler, data$segment, 
@interrupt$status);
```

132  Nucleus System Calls
The second system call, \texttt{WAIT\$INTERRUPT}, is used by the interrupt task to signal its readiness to service an interrupt.

\texttt{CALL \textit{RQ\$WAIT\$INTERRUPT} (interrupt\$level\$7, @interrupt\$status);}

\texttt{CALL \textit{RQ\$RESET\$INTERRUPT} (interrupt\$level\$7, @interrupt\$status);}

\texttt{END INTERRUPTTASK;}

\texttt{SAMPLEPROCEDURE: PROCEDURE;}

\texttt{start\$address = @INTERRUPTTASK; /* 1st instruction of interrupt task */}
\texttt{stack\$pointer = NIL; /* automatic stack allocation */}
\texttt{task\$flags = 0; /* indicates no floating-point instructions */}
\texttt{data\$segment = SELECTOR$OF(NIL); /* use own data segment */}

\texttt{END SAMPLEPROCEDURE;}
### Condition Codes

<table>
<thead>
<tr>
<th>Code</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>E$OK</td>
<td>0000H</td>
<td>No exceptional conditions.</td>
</tr>
<tr>
<td>E$CONTEXT</td>
<td>0005H</td>
<td>There is not an interrupt handler assigned to the specified level.</td>
</tr>
<tr>
<td>E$NOT$CONFIGURED</td>
<td>0008H</td>
<td>This system call is not part of the present configuration.</td>
</tr>
<tr>
<td>E$PARAM</td>
<td>8004H</td>
<td>The level parameter is invalid.</td>
</tr>
</tbody>
</table>
The RESUME$TASK system call decreases by one the suspension depth of a task.

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

Input Parameter
\begin{itemize}
  \item \textbf{task} \quad A \textsc{token} for the task whose suspension depth is to be decremented.
\end{itemize}

Output Parameter
\begin{itemize}
  \item \textbf{except$ptr} \quad A \textsc{pointer} to a \textsc{word} to which the iRMX I Operating System will return the condition code generated by this system call.
\end{itemize}

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';
   /* if your PL/M compiler does not support this variable type, declare TOKEN a WORD */

/* NUCLUS.EXT declares all nucleus system calls */
$INCLUDE(:RMX: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; /* indicates no floating-point instructions */

•
•
•

Typical PL/M-86 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-86 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-86 Statements

Using the token for the suspended task (whose code is labeled TASKCODE), the calling task invokes RESUME$TASK to decrease by one the suspension depth of the suspended task.

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

Typical PL/M-86 Statements

END SAMPLEPROCEDURE;
RESUME$TASK

Condition Codes

<table>
<thead>
<tr>
<th>Code</th>
<th>Value</th>
<th>Message</th>
</tr>
</thead>
<tbody>
<tr>
<td>E$OK</td>
<td>0000H</td>
<td>No exceptional conditions.</td>
</tr>
<tr>
<td>E$CONTEXT</td>
<td>0005H</td>
<td>The task indicated by the task parameter is an interrupt task.</td>
</tr>
<tr>
<td>E$EXIST</td>
<td>0006H</td>
<td>The task parameter is not a token for an existing object.</td>
</tr>
<tr>
<td>E$STATE</td>
<td>0007H</td>
<td>The task indicated by the task parameter was not suspended when the call was made.</td>
</tr>
<tr>
<td>E$TYPE</td>
<td>8002H</td>
<td>The task parameter is a token for an object that is not a task.</td>
</tr>
</tbody>
</table>
The SEND$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 I 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$CONTROL has had its priority boosted while it had access through a region, its priority is restored when it relinquishes the access.
SEND$CONTROL

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';
    /* if your PL/M compiler does not support this variable type, 
       declare TOKEN a WORD */

/* NUCLUS.EXT declares all nucleus system calls */
$INCLUDE(:RMX:INC/NUCLUS.EXT)

DECLARE region$token TOKEN;
DECLARE priority$queue LITERALLY '1'; /* tasks wait in priority order*/
DECLARE status WORD;

•

Typical PL/M-86 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-86 Statements

CALL RQ$RECEIVE$CONTROL (region$token, @status);

•

Typical PL/M-86 Statements

Nucleus System Calls
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-86 Statements

END SAMPLEPROCEDURE;

Condition Codes

<table>
<thead>
<tr>
<th>Code</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>E$OK</td>
<td>0000H</td>
<td>No exceptional conditions.</td>
</tr>
<tr>
<td>E$CONTEXT</td>
<td>0005H</td>
<td>The calling task does not have access to data protected by any region.</td>
</tr>
<tr>
<td>E$NOT$CONFIGURED</td>
<td>0008H</td>
<td>This system call is not part of the present configuration.</td>
</tr>
</tbody>
</table>
**SEND$MESSAGE**

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.

- **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 not SELECTOR$OF(NIL) or zero, contains a token for the desired response mailbox or semaphore.
    - If SELECTOR$OF(NIL) or zero, indicates that no response is requested.

**Output Parameter**

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

**Description**

The SEND$MESSAGE system call sends the specified object token 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.
Example

 HeaderComponent

 segment token to a mailbox.

 DECLARE TOKEN LITERALLY 'SELECTOR';
 /* if your PL/M compiler does not
 support this variable type,
 declare TOKEN a WORD */

 /* NUCLUS.EXT declares all nucleus system calls */
 $INCLUDE(:RMX: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-86 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 some way communicate) the mailbox token if another task is to receive the message.

CALL RQ$CATALOG$OBJECT
(jobj$token,
mbx$token,
@3, 'MBX'),
@status);

Typical PL/M-86 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
(mb$x$token,
seg$token,
no$response,
@status);

Typical PL/M-86 Statements

END SAMPLEPROCEDURE;
### Condition Codes

<table>
<thead>
<tr>
<th>Code</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>E$OK</td>
<td>0000H</td>
<td>No exceptional conditions.</td>
</tr>
<tr>
<td>E$EXIST</td>
<td>0006H</td>
<td>One or more of the input parameters is not a token for an existing object.</td>
</tr>
<tr>
<td>E$MEM</td>
<td>0002H</td>
<td>The high performance queue is full and the calling task's job does not contain sufficient memory to complete the call.</td>
</tr>
<tr>
<td>E$NOT$CONFIGURED</td>
<td>0008H</td>
<td>This system call is not part of the present configuration.</td>
</tr>
<tr>
<td>E$TYPE</td>
<td>8002H</td>
<td>At least one of the following is true:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• The mailbox parameter is a token for an object that is not a mailbox.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• The response parameter is a token for an object that is neither a mailbox nor a semaphore.</td>
</tr>
</tbody>
</table>
SEND$UNITS

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 I 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';
    /* if your PL/M compiler does not support this variable type, 
     declare TOKEN a WORD */

/* NUCLUS.EXT declares all nucleus system calls */
$INCLUDE(:RMX: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 */

    •
    •
    •

/*************************************************************************/
/* 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,

    •
    •

Typical PL/M-86 Statements
SEND$UNITS

/************************************************************************
** 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-86 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-86 Statements

END SAMPLEPROCEDURE;

Condition Codes

<table>
<thead>
<tr>
<th>Condition Code</th>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>E$OK</td>
<td>0000H</td>
<td>No exceptional conditions.</td>
</tr>
<tr>
<td>E$EXIST</td>
<td>0006H</td>
<td>The semaphore parameter is not a token for an existing object.</td>
</tr>
<tr>
<td>E$LIMIT</td>
<td>0004H</td>
<td>The number of units that the calling task is trying to send would cause the semaphore's supply of units to exceed its maximum allowable supply.</td>
</tr>
<tr>
<td>E$NOT$CONFIGURED</td>
<td>0008H</td>
<td>This system call is not part of the present configuration.</td>
</tr>
<tr>
<td>E$TYPE</td>
<td>8002H</td>
<td>The semaphore parameter is a token for an object that is not a semaphore.</td>
</tr>
</tbody>
</table>
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$HANDLER$OFFSET WORD,
    EXCEPTION$HANDLER$BASE TOKEN,
    EXCEPTION$MODE BYTE);
  ```

  where:
  - `exception$handler$offset` contains the offset of the first instruction of the exception handler.
  - `exception$handler$base` contains the base of the CPU segment containing 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$handler$offset` is SELECTOR$OF(NIL) and `exception$handler$base` is zero, the exception handler of the calling task's parent job is assigned.

**Output Parameter**

- **except$ptr**: A POINTER to a WORD to which the iRMX I Operating System will return the condition code generated by this system call.
SET$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-86:

```
DECLARE X STRUCTURE (OFFSET BASE MODE
   WORD TOKEN BYTE); /* establish a structure for exception handlers */

DECLARE Y POINTER AT (@X);

DECLARE EXCEPTION WORD;

Y = @RQDEBUGGEREX; /*@RQDEBUGGER" is the public symbol for the Debugger, 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';
   /* if your PL/M compiler does not
      support this variable type,
      declare TOKEN a WORD */

/* NUCLUS.EXT declares all nucleus system calls */
$INCLUDE(:RMX:INC/NUCLUS.EXT)

EXCEPTIONHANDLER: PROCEDURE EXTERNAL;
END EXCEPTIONHANDLER;

DECLARE X$HANDLER$STRUCTURE LITERALLY 'STRUCTURE( offset WORD,
   base TOKEN,
   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;
SET$EXCEPTION$HANDLER

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-86 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-86 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-86 Statements
•

Nucleus System Calls
/********************************************************************
* No longer needing the new exception handler, the calling task uses *
* the address and mode of the old exception handler to return Exception_handler.* *
* exception handling to its original exception handler.              *
*********************************************************************/

CALL RQ$SET$EXCEPTION$HANDLER (@x$handler,  
  @status);

•
• Typical PL/M-86 Statements
•

END SAMPLEPROCEDURE;

**Condition Codes**

<table>
<thead>
<tr>
<th>Code</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>E$OK</td>
<td>0000H</td>
<td>No exceptional conditions.</td>
</tr>
<tr>
<td>E$NOT$CONFIGURED</td>
<td>0008H</td>
<td>This system call is not part of the present configuration.</td>
</tr>
<tr>
<td>E$PARAM</td>
<td>8004H</td>
<td>The exception$mode parameter is greater than 3.</td>
</tr>
</tbody>
</table>
The SET$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$INTERRUPT requests that can exist. When this limit is reached, the associated interrupt level is disabled. The maximum value for this parameter is 255 decimal. The *iRMX*® *I Nucleus User’s Guide* describes this feature in more detail.
  - If zero, indicates that no interrupt task is to be associated with the special level and that the new interrupt handler will not call SIGNAL$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$handler A POINTER to the first instruction of the interrupt handler. To obtain the proper start address for interrupt handlers written in PL/M-86, place the following instruction before the call to SET$INTERRUPT:

interrupt$handler = interrupt$ptr (inter);

where interrupt$ptr is a PL/M-86 built-in procedure and inter is the name of your interrupt handling procedure.

interrupt$handler$ds A TOKEN that specifies the interrupt handler's data segment.

- If not SELECTOR$OF(NIL) or zero, it contains the base address of the interrupt handler's data segment. See the description of ENTER$INTERRUPT in this manual for information concerning the significance of this parameter.

- If SELECTOR$OF(NIL) or zero, the parameter indicates that the interrupt handler will load its own data segment and may not invoke ENTER$INTERRUPT.

It is often desirable for an interrupt handler to pass information to the interrupt task that it calls. The following PL/M-86 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$INTERRUPT:
SET$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 I Operating System
will return the condition code generated by this system call.

Description

The SET$INTERRUPT system call is used to inform the Nucleus that the specified
interrupt handler is to service interrupts which come in at the specified level. In a call to
SE T$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 iRMX® I 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';
            /* if your PL/M compiler does not support this variable type, 
               declare TOKEN a WORD */

/* NUCLUS.EXT declares all nucleus system calls */
$INCLUDE(:RMX:INC/NUCLUS.EXT)

INTERRUPTHANDLER: PROCEDURE INTERRUPT 63 EXTERNAL;
END INTERRUPTHANDLER;

DECLARE interrupt$level$7 LITERALLY '000000001111000B';
            /* 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 the interrupt handler will load its own data segment */

    interrupt$handler = INTERRUPT$PTR (INTERRUPTHANDLER);
             /* points to the first instruction of the interrupt handler */

    •
         • Typical PL/M-86 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$flag, 
    interrupt$handler, 
    data$segment, 
    @status);

Nucleus System Calls 157
SET$INTERRUPT

• Typical PL/M-86 Statements

END SAMPLEPROCEDURE;

Condition Codes

<table>
<thead>
<tr>
<th>Code</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>E$OK</td>
<td>0000H</td>
<td>No exceptional conditions.</td>
</tr>
<tr>
<td>E$CONTEXT</td>
<td>0005H</td>
<td>One of the following is true:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• The task is already an interrupt task.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• The specified level already has an interrupt handler assigned to it.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• The job containing the calling task or the calling task itself is in the process of being deleted.</td>
</tr>
<tr>
<td>E$NOT$CONFIGURED</td>
<td>0008H</td>
<td>This system call is not part of the present configuration</td>
</tr>
<tr>
<td>E$PARAM</td>
<td>8004H</td>
<td>One of the following is true:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• The level parameter is invalid or would cause the task to have a priority not allowed by its job.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• The programmable interrupt controller (PIC) corresponding to the specified level is not part of the hardware configuration.</td>
</tr>
</tbody>
</table>
The SET$OS$EXTENSION system call either enters the address of an entry (or function) procedure in the interrupt vector table or it deletes such an entry.

CAUTION

This system call should not be used by Human Interface applications that are loaded into dynamic memory. If such an application is deleted (via a CONTROL-C entered at a terminal), subsequent interrupts to the vector table entry set by this system call could cause unpredictable results.

CALL RQ$SET$OS$EXTENSION (os$extension, start$address, except$ptr);

Input Parameters

os$extension A BYTE designating the entry of the interrupt vector table to be set or reset. This value must be between 192 and 255 (decimal), inclusive. The values in the range 0 to 191 are valid, but are reserved for Intel use.

start$address A POINTER to the first instruction of an entry (or function) procedure. If start$address contains a NIL or zero value, the specified interrupt vector table entry is being reset (deallocated).

Output Parameter

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

Description

The SET$OS$EXTENSION system call sets or resets any one of the 32 operating system extension entries in the interrupt vector table. An entry must be reset before its contents can be changed. An attempt to set an already set entry causes an E$CONTEXT exceptional condition.
Example

/*************************************************************************/
* This example illustrates how the SET$OS$EXTENSION system call can * *
* be used to reset an entry in the Interrupt Vector Table. The * *
* example assumes that the entry for the level (number 250) was set * *
* earlier by another procedure. * *
*************************************************************************/

DECLARE TOKEN LITERALLY 'SELECTOR';
    /* if your PL/M compiler does not 
       support this variable type, 
       declare TOKEN a WORD */

/* NUCLUS.EXT declares all nucleus system calls */
$INCLUDE(:RMX:INC/NUCLUS.EXT)

DECLARE vector$entry$250 LITERALLY '250';
DECLARE reset LITERALLY '0';
DECLARE status WORD;

SAMPLEPROCEDURE:
    PROCEDURE;

    •

    • Typical PL/M-86 Statements

    •

    /*************************************************************************/
* The calling task invokes the SET$OS$EXTENSION system call to reset * *
* entry 250 (decimal) of the Interrupt Vector Table. * *
*************************************************************************/

CALL RQ$SET$OS$EXTENSION (vector$entry$250, reset,
@status);

    •

    • Typical PL/M-86 Statements

    •

END SAMPLEPROCEDURE;
**Condition Codes**

<table>
<thead>
<tr>
<th>Code</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>E$OK</td>
<td>0000H</td>
<td>No exceptional conditions.</td>
</tr>
<tr>
<td>E$CONTEXT</td>
<td>0005H</td>
<td>The entry is already set. Before you can set the entry again, you must first reset it (call SET$OS$EXTENSION and specify a 0 for the start$address parameter).</td>
</tr>
<tr>
<td>E$NOT$CONFIGURED</td>
<td>0008H</td>
<td>This system call is not part of the present configuration.</td>
</tr>
<tr>
<td>E$PARAM</td>
<td>8004H</td>
<td>The OS$extension byte value is less than 192.</td>
</tr>
</tbody>
</table>
The SET$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 I 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';
/* if your PL/M compiler does not support this variable type,
   declare TOKEN a WORD */

/* NUCLUS.EXT declares all nucleus system calls */
$INCLUDE(:RMX:INC/NUCLUS.EXT)

DECLARE new$min WORD;
DECLARE status WORD;

SAMPLEPROCEDURE:
PROCEDURE;

new$min = OFFFFH; /* sets pool$min attribute of calling
       task's job equal to job's pool$max
       attribute */

   •
   • Typical PL/M-86 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-86 Statements
   •

END SAMPLEPROCEDURE;
```
## Condition Codes

<table>
<thead>
<tr>
<th>Code</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>E$OK</td>
<td>0000H</td>
<td>No exceptional conditions.</td>
</tr>
<tr>
<td>E$LIMIT</td>
<td>0004H</td>
<td>The new min parameter is not OFFFFH, but it is greater than the pool max attribute of the calling task's job.</td>
</tr>
<tr>
<td>E$NOT$CONFIGURED</td>
<td>0008H</td>
<td>This system call is not part of the present configuration.</td>
</tr>
</tbody>
</table>
The SET$PRIORIY 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 I Operating System can be degraded when a task uses this system call to lower its own priority.

CALL RQ$SET$PRIORIY (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) or zero 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 I Operating System will return the condition code generated by this system call.

Description

The SET$PRIORIY 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';
    /* if your PL/M compiler does not support this variable type,
    declare TOKEN a WORD */

/* NUCLUS.EXT declares all nucleus system calls */
$INCLUDE(:RMX:INC/NUCLUS.EXT)

TASKCODE: PROCEDURE EXTERNAL;
END TASKCODE;

DECLARE task$token TOKEN;
DECLARE priority$level$66 LITERALLY '66';
DECLARE priority$level$0 LITERALLY '0';
DECLARE start$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-86 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.

```c
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.

```c
CALL RQ$CATALOG$OBJECT (job$token,
                               task$token,
                               @(8,'TASKCODE'),
                               @status);
```

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.

```c
CALL RQ$SET$PRIORITY (task$token,
                          priority$level$0,
                          @status);
```

---

**Nucleus System Calls**

167
SET$PRIORITY

************************************************************************
* 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-86 Statements
•

END SAMPLEPROCEDURE;

Condition Codes

<table>
<thead>
<tr>
<th>Code</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>E$OK</td>
<td>0000H</td>
<td>No exceptional conditions.</td>
</tr>
<tr>
<td>E$CONTEXT</td>
<td>0005H</td>
<td>The specified task is an interrupt task. You cannot set the priority of an interrupt task dynamically.</td>
</tr>
<tr>
<td>E$EXIST</td>
<td>0006H</td>
<td>The task parameter is not a token for an existing object.</td>
</tr>
<tr>
<td>E$LIMIT</td>
<td>0004H</td>
<td>The priority parameter contains a priority value that is higher than the maximum priority of the specified task's containing job.</td>
</tr>
<tr>
<td>E$NOT$CONFIGURED</td>
<td>0008H</td>
<td>This system call is not part of the present configuration.</td>
</tr>
<tr>
<td>E$TYPE</td>
<td>8002H</td>
<td>The task parameter is a token for an object that is not a task.</td>
</tr>
</tbody>
</table>
The SIGNAL$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$code**: A WORD containing the code (see list in the iRMX® I Nucleus User's Guide) for the exceptional condition detected.
- **param$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 iRMX® I 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**: 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$ptr**: A POINTER to a WORD to which the iRMX I Operating System will return the condition code generated by this system call.
SIGNAL$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';
    /* if your PL/M compiler does not support this variable type,     
      declare TOKEN a WORD */

/* NUCLUS.EXT declares all nucleus system calls */
$INCLUDE(:RMX: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 = 0;          /* return control to instruction      
                             following call */

•
  • Typical PL/M-86 Statements
•

170 Nucleus System Calls
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-86 Statements

END SAMPLEPROCEDURE;

Condition Codes

E$OK 0000H No exceptional conditions.
The SIGNAL$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):

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>15-7</td>
<td>Reserved bits that should be set to zero.</td>
</tr>
<tr>
<td>6-4</td>
<td>First digit of the interrupt level (0-7).</td>
</tr>
<tr>
<td>3</td>
<td>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.</td>
</tr>
<tr>
<td>2-0</td>
<td>Second digit of the interrupt level (0-7), if bit 3 is zero.</td>
</tr>
</tbody>
</table>

Output Parameter

except$ptr A POINTER to a WORD to which the iRMX I 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.

DECLARE TOKEN LITERALLY 'SELECTOR';
/* if your PL/M compiler does not support this variable type, declare TOKEN a WORD */

/* NUCLUS.EXT declares all nucleus system calls */
$INCLUDE(:RMX:INC/NUCLUS.EXT)

DECLARE interrupt$level$7 LITERALLY '000000001111000B';
/* specifies master interrupt level 7 */
DECLARE E$OK LITERALLY '00H';
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 59 PUBLIC; /* 59 is meaningless value. ENTER$INTER-
RUPT establishes actual level */

Typical PL/M-86 Statements
SIGNAL$interrupt

*************************************************************************
* 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-86 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-86 Statements

•

END;

END INLINEERRORPROCESS;

174 Nucleus System Calls
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$handler = INTERRUPT$PTR (INTERRUPTHANDLER);
  /* points to the first instruction of the interrupt handler */
interrupt$task$flag = O1H; /* indicates that calling task is to be interrupt task */

  •
  •
  •

Typical PL/M-86 Statements

CALL RQ$SET$Interrupt
(interrupt$level$7,
interrupt$task$flag,
interrupt$handler,
data$segment,
@status);

  •
  •

Typical PL/M-86 Statements

END SAMPLEPROCEDURE;
### SIGNAL$INTERRUPT

#### Condition Codes

<table>
<thead>
<tr>
<th>Code</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>E$OK</td>
<td>0000H</td>
<td>No exceptional conditions.</td>
</tr>
<tr>
<td>E$CONTEXT</td>
<td>0005H</td>
<td>No interrupt task is assigned to the specified level.</td>
</tr>
<tr>
<td>E$INTERRUPT$OVERFLOW</td>
<td>000AH</td>
<td>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$INTERRUPT.</td>
</tr>
<tr>
<td>E$INTERRUPT$SATURATION</td>
<td>0009H</td>
<td>The interrupt task has accumulated the maximum allowable number of SIGNAL$INTERRUPT requests. This is an informative message only. It does not indicate an error.</td>
</tr>
<tr>
<td>E$LIMIT</td>
<td>0004H</td>
<td>An overflow has occurred because the interrupt task has received more than 255 SIGNAL$INTERRUPT requests.</td>
</tr>
<tr>
<td>E$NOT$CONFIGURED</td>
<td>0008H</td>
<td>This system call is not part of the present configuration.</td>
</tr>
<tr>
<td>E$PARAM</td>
<td>8004H</td>
<td>The level parameter is invalid.</td>
</tr>
</tbody>
</table>
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 iRMX I 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 I 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.
SLEEP

Example

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

DECLARE TOKEN LITERALLY 'SELETO';
    /* if your PL/M compiler does not support this variable type, 
       declare TOKEN a WORD */

/* NUCLUS.EXT declares all nucleus system calls */
$INCLUDE(:RMX:INC/NUCLUS.EXT)

DECLARE time$limit WORD;
DECLARE status WORD;

SAMPLEPROCEDURE:
    PROCEDURE;

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

    CALL RQ$SLEEP
        (time$limit,
         @status);

END SAMPLEPROCEDURE;

Condition Codes

E$OK 0000H No exceptional conditions.

E$NOT$CONFIGURED 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 not SELECTOR$OF(NIL) or zero, contains a token for the task whose suspension depth is to be incremented.
  • if SELECTOR$OF(NIL) or zero, indicates that the calling task is suspending itself.

Output Parameter

(except$ptr) A POINTER to a WORD to which the iRMX I 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';
    /* if your PL/M compiler does not support this variable type,
       declare TOKEN a WORD */

/* NUCLUS.EXT declares all nucleus system calls */
$INCLUDE(:RMX:INC/NUCLUS.EXT)

TASKCODE: PROCEDURE EXTERNAL;
END TASKCODE;

DECLARE task$token TOKEN;
DECLARE priority$level$LITERALLY '200';
DECLARE start$address POINTER;
DECLARE data$seg TOKEN;
DECLARE stack$pointer POINTER;
DECLARE stack$size$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-86 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);
```

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);
```

**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$TYPE** 8002H The task parameter is a token for an object that is not a task.
UNCATALOG$OBJECT

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

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>job</td>
<td>A TOKEN indicating the job of the object directory from which an entry is to be deleted.</td>
</tr>
<tr>
<td></td>
<td>• If not SELECTOR$OF(NIL) or zero, the TOKEN contains a token for the job from whose object directory the specified entry is to be deleted.</td>
</tr>
<tr>
<td></td>
<td>• If SELECTOR$OF(NIL) or zero, the entry is to be deleted from the object directory of the calling task’s job.</td>
</tr>
<tr>
<td>name</td>
<td>A POINTER to a STRING containing the name of the object whose entry is to be deleted.</td>
</tr>
</tbody>
</table>

Output Parameter

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>except$ptr</td>
<td>A POINTER to a WORD to which the iRMX I Operating System will return the condition code generated by this system call.</td>
</tr>
</tbody>
</table>

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';
/* if your PL/M compiler does not support this variable type,
declare TOKEN a WORD */

/* NUCLUS.EXT declares all nucleus system calls */
$INCLUDE(:RMX: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-86 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-86 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-86 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-86 Statements

END SAMPLEPROCEDURE;
### Condition Codes

<table>
<thead>
<tr>
<th>Code</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>E$OK</td>
<td>0000H</td>
<td>No exceptional conditions.</td>
</tr>
<tr>
<td>E$CONTEXT</td>
<td>0005H</td>
<td>The specified object directory does not contain an entry with the designated name.</td>
</tr>
<tr>
<td>E$EXIST</td>
<td>0006H</td>
<td>The job parameter is neither zero nor a token for an existing object.</td>
</tr>
<tr>
<td>E$NOT$CONFIGURED</td>
<td>0008H</td>
<td>This system call is not part of the present configuration.</td>
</tr>
<tr>
<td>E$PARAM</td>
<td>8004H</td>
<td>The first byte of the STRING pointed to by the name parameter contains a value greater than 12 or equal to 0.</td>
</tr>
<tr>
<td>E$TYPE</td>
<td>8002H</td>
<td>The job parameter is a token for an object that is not a job.</td>
</tr>
</tbody>
</table>
The WAIT$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**

<table>
<thead>
<tr>
<th>level</th>
<th>A WORD specifying an interrupt level which is encoded as follows (bit 15 is the high-order bit):</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bits</td>
<td>Value</td>
</tr>
<tr>
<td>15-7</td>
<td>Reserved bits that should be set to zero.</td>
</tr>
<tr>
<td>6-4</td>
<td>First digit of the interrupt level (0-7).</td>
</tr>
<tr>
<td>3</td>
<td>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.</td>
</tr>
<tr>
<td>2-0</td>
<td>Second digit of the interrupt level (0-7), if bit 3 is zero.</td>
</tr>
</tbody>
</table>

**Output Parameter**

| except$ptr | A POINTER to a WORD to which the iRMX I 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 SETS$INTERRUPT system call. If the associated interrupt level is enabled, all SIGNALS$INTERRUPT calls that the handler makes (up to the limit specified with SETS$INTERRUPT) are logged. If this count of SIGNALS$INTERRUPT calls is greater than zero when the interrupt task calls WAITS$INTERRUPT, the task is not suspended. Instead it continues processing the next SIGNALS$INTERRUPT request.

If the associated interrupt level is disabled while the interrupt task is running and the number of outstanding SIGNALS$INTERRUPT requests is less than the user-specified limit, the call to WAITS$INTERRUPT enables that level.

**Example**

```plaintext
/* This example illustrates how the WAITS$INTERRUPT system call can be used to signal a task's readiness to service an interrupt. */

DECLARE TOKEN LITERALLY 'SELECTOR';
   /* if your PL/M compiler does not support this variable type,
      declare TOKEN a WORD */

/* NUCLUS.EXT declares all nucleus system calls */
$INCLUDE(:RMX:INC/NUCLUS.EXT)

INTERRUPTHANDLER: PROCEDURE INTERRUPT 63 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 '000000001111000B'; /* specifies master interrupt level 7 */

DECLARE interrupt$task$flag BYTE;
DECLARE interrupt$handler POINTER;
DECLARE interrupt$status WORD;
DECLARE status WORD;
```
WAIT$INTERRUPT

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 */
interrupt$handler = INTERRUPT$PTR (INTERRUPTHANDLER);
/* points to the first instruction of
the interrupt handler */

/***************************************************************
* 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$flag,
   interrupt$handler,
   data$segment,
   @interrupt$status);

  * Typical PL/M-86 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-86 Statements
  *
/***************************************************************
* When the interrupt task invokes the RESET$INTERRUPT system call, *
* the assignment of the current interrupt handler to interrupt level *
* 7 is cancelled and, because an interrupt task has also been *
* assigned to the line, the interrupt task is deleted. *
***************************************************************
CALL RQ$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 */
WAIT$INTERCEPT

```
task$flags = 0;                          /* designates no floating-point
                                           instructions */
data$segment = SELECTOR$OF(NIL);     /* use own data segment */

•                                           
•                                           
• Typical PL/M-86 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-86 Statements

END SAMPLEPROCEDURE;

**Condition Codes**

<table>
<thead>
<tr>
<th>Code</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>E$OK</td>
<td>0000H</td>
<td>No exceptional conditions.</td>
</tr>
<tr>
<td>E$CONTEXT</td>
<td>0005H</td>
<td>The calling task is not the interrupt task for the given level.</td>
</tr>
<tr>
<td>E$NOT$CONFIGURED</td>
<td>0008H</td>
<td>This system call is not part of the present configuration.</td>
</tr>
<tr>
<td>E$PARAM</td>
<td>8004H</td>
<td>The level parameter is invalid.</td>
</tr>
</tbody>
</table>
INDEX

A
ACCEP$CONTROL 7
ALTER$COMPOSITE 10

C
CATALOG$OBJECT 12
CREATE$COMPOSITE 15
CREATE$EXTENSION 18
CREATE$JOB 21
CREATE$MAILBOX 29
CREATE$REGION 33
CREATE$SEGMENT 36
CREATE$SEMAPHORE 39
CREATE$TASK 42

D
DELETE$COMPOSITE 47
DELETE$EXTENSION 49
DELETE$JOB 52
DELETE$MAILBOX 55
DELETE$REGION 58
DELETE$SEGMENT 61
DELETE$SEMAPHORE 64
DELETE$TASK 67
DISABLE 71
DISABLE$DELETION 74

E
ENABLE 77
ENABLE$DELETION 81
Encoded meanings for object types 110
Encoding of interrupt levels 77
END$INIT$TASK 84
ENTER$INTERRUPT 85

Nucleus System Calls

Index-1
Index

E (continued)

Examples

- ACCEPT$CONTROL 8
- CATALOG$OBJECT 13
- CREATE$JOB 25
- CREATE$MAILBOX 31
- CREATE$REGION 34
- CREATE$SEGMENT 37
- CREATE$SEMAPHORE 40
- CREATE$TASK 44
- DELETE$EXTENSION 50
- DELETE$JOB 53
- DELETE$MAILBOX 56
- DELETE$REGION 59
- DELETE$SEGMENT 62
- DELETE$SEMAPHORE 65
- DELETE$TASK 68
- DISABLE 72
- DISABLE$DELETION 75
- ENABLE 78
- ENABLE$DELETION 82
- ENTER$INTERRUPT 86
- EXIT$INTERRUPT 90
- FORCE$DELETE 93
- GET$EXCEPTION$HANDLER 96
- GET$LEVEL 98
- GET$POOL$ATTRIB 100
- GET$PRIORITY 103
- GET$SIZE 106
- GET$TASK$TOKENS 109
- GET$TYPE 111
- LOOKUP$OBJECT 116
- OFFSPRING 119
- RECEIVE$CONTROL 122
- RECEIVE$MESSAGE 126
- RECEIVE$UNITS 129
- RESET$INTERRUPT 132
- RESUME$TASK 136
- SEND$CONTROL 140
- SEND$MESSAGE 143
- SEND$UNITS 147
- SET$EXCEPTION$HANDLER 151
- SET$INTERRUPT 157
E (continued)

SET$OS$EXTENSION 160
SET$POOL$MIN 163
SET$PRIORITY 166
SIGNAL$EXCEPTION 170
SIGNAL$INTERRUPT 173
SLEEP 178
SUSPEND$TASK 180
UNCATALOG$OBJECT 183
WAIT$INTERRUPT 187
EXIT$INTERRUPT 89

F

FORCE$DELETE 92

G

GET$EXCEPTION$HANDLER 95
GET$LEVEL 97
GET$POOL$ATTRIB 99
GET$PRIORITY 102
GET$SIZE 105
GET$TASK$TOKENS 108
GET$TYPE 110

I

INSPECT$COMPOSITE 113

L

LOOKUP$OBJECT 115

M

Mailbox$flags
  specifying information when creating a mailbox 29
Meaning of the encoded interrupt level WORD 97

O

OFFSPRING 118
INDEX

Q
Queuing scheme of a semaphore  39

R
RECEIVE$CONTROL  121
RECEIVE$MESSAGE  124
RECEIVE$UNITS  128
Required top 5 words of stack for SIGNAL$EXCEPTION  169
RESET$INTERUPT  131
RESUME$TASK  135

S
SEND$CONTROL  139
SEND$MESSAGE  142
SEND$UNITS  146
SET$EXCEPTION$HANDLER  148
SET$INTERUPT  154
SET$OS$EXTENSION  159
SET$POOL$MIN  162
SET$PRIORITY  165
SIGNAL$EXCEPTION  169
SIGNAL$INTERUPT  172
SLEEP  177
Structures
  exception handler  22
  extracting the DS register used by an interrupt task  156
  for assigning as exception handler  149
  information about the exception handler  95
  pool attributes for GET$POOL$ATTRIBUTES  99
  token$list for CREATE$COMPOSITE  15
  token$list for INSPECT$COMPOSITE  113
SUSPEND$TASK  179

U
UNCATALOG$OBJECT  182

V
Values for GET$TASK$TOKENS selection parameter  108
W
WAI$INTERRUPT 186
REQUEST FOR READER’S COMMENTS

Intel’s Technical Publications Departments attempt to provide publications that meet the needs of all Intel product users. This form lets you participate directly in the publication process. Your comments will help us correct and improve our publications. Please take a few minutes to respond.

Please restrict your comments to the usability, accuracy, organization, and completeness of this publication. If you have any comments on the product that this publication describes, please contact your Intel representative.

1. Please describe any errors you found in this publication (include page number).
   ____________________________________________________
   ____________________________________________________
   ____________________________________________________

2. Does this publication cover the information you expected or required? Please make suggestions for improvement.
   ____________________________________________________
   ____________________________________________________
   ____________________________________________________

3. Is this the right type of publication for your needs? Is it at the right level? What other types of publications are needed?
   ____________________________________________________
   ____________________________________________________
   ____________________________________________________

4. Did you have any difficulty understanding descriptions or wording? Where?
   ____________________________________________________
   ____________________________________________________
   ____________________________________________________

5. Please rate this publication on a scale of 1 to 5 (5 being the best rating).
   ________________

NAME __________________________________________ DATE __________
TITLE __________________________________________
COMPANY NAME/DEPARTMENT ____________________________
ADDRESS __________________________________________ PHONE ( )
CITY __________________________ STATE _______________ ZIP CODE __________
   (COUNTRY)

Please check here if you require a written reply. □
WE'D LIKE YOUR COMMENTS . . .

This document is one of a series describing Intel products. Your comments on the back of this form will help us produce better manuals. Each reply will be carefully reviewed by the responsible person. All comments and suggestions become the property of Intel Corporation.

If you are in the United States, use the preprinted address provided on this form to return your comments. No postage is required. If you are not in the United States, return your comments to the Intel sales office in your country. For your convenience, international sales office addresses are printed on the last page of this document.
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 Sharef
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