intel®

iRMX® I Programming Techniques Reference Manual

Order Number: 462931-001

iRMX® I Programming Techniques Reference Manual

Order Number: 462931-001

Intel Corporation
3065 Bowers Avenue
Santa Clara, California 95051

Copyright © 1980, 1989, Intel Corporation, All Rights Reserved

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:

Above	iLBX	iPSC	Plug-A-Bubble
BITBUS	i$_m$	iRMX	PROMPT
COMMputer	iMDDX	iSBC	Promware
CREDIT	iMMX	iSBX	QUEST
Data Pipeline	Insite	iSDM	QueX
Genius	int$_e$l	iSSB	Ripplemode
$\overset{\triangle}{\text{i}}$	Intel376	iSXM	RMX/80
i	Intel386	Library Manager	RUPI
I^{2}ICE	int$_e$lBOS	MCS	Seamless
ICE	Intelevision	Megachassis	SLD
iCEL	int$_e$ligent Identifier	MICROMAINFRAME	UPI
iCS	int$_e$ligent Programming	MULTIBUS	VLSiCEL
iDBP	Intellec	MULTICHANNEL	376
iDIS	Intellink	MULTIMODULE	386
	iOSP	OpenNET	386SX
	iPDS	ONCE	
	iPSB		

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

Copyright© 1980, 1989, Intel Corporation. All Rights Reserved.

REV.	REVISION HISTORY	DATE
-001	Original Issue.	03/89

PREFACE

This manual summarizes techniques that will be useful to you as you produce an application system based on the iRMX® I Operating System. A typical development process goes through these stages:

- Dividing the application into jobs and tasks
- Writing the code for tasks
- Writing interrupt handlers
- Configuring and starting up the system
- Debugging an application

This manual should be used as a reference guide when developing your application system. The techniques described here will help you save time and avoid problems during the development process.

CONVENTIONS

The following conventions are used throughout this manual:

- Information appearing as UPPERCASE characters when shown in keyboard examples must be entered or coded exactly as shown. You may, however, mix lower and uppercase characters when entering the text.
- Fields appearing as lowercase characters within angle brackets (< >) when shown in keyboard examples indicate variable information. You must enter an appropriate value or symbol for variable fields.
- User input appears in one of the following forms:

```
as blue text
```

```
as bolded text within a screen
```

- The term "iRMX I" refers to the iRMX I (iRMX 86) Operating System.
- All numbers, unless otherwise stated, are assumed to be decimal. Hexadecimal numbers include the "H" radix character (for example, 0FFH).

CONTENTS

Figures

Tables

SELECTING A PL/M-86 SIZE CONTROL 1

1.1 INTRODUCTION

You should read this chapter only if you will be programming iRMX® I tasks using PL/M-86. You should be familiar with the following concepts:

- The PL/M-86 programming language
- PL/M-86 models of segmentation
- iRMX I jobs, tasks, and segments

Whenever you invoke the PL/M-86 compiler, you must specify (either explicitly or by default) a program size control (SMALL, COMPACT, MEDIUM, or LARGE). This size control determines which model of segmentation the compiler uses and thus affects how much memory you need to store your application's object code.

The following sections explain which size control you should use to produce the smallest object program while still satisfying the requirements of the system.

1.2 MAKING THE SELECTION

Since you cannot use the SMALL size control, you must choose COMPACT, MEDIUM, or LARGE. The algorithm for selecting a size control is presented later in this chapter; note, however, that your choice can place restrictions on the system.

1.2.1 Ramifications of Your Selection

If you use the COMPACT or MEDIUM size controls, the capabilities of the system will be slightly restricted. Only the LARGE size control preserves all features of the system.

Restrictions Associated with MEDIUM

If you use PL/M-86 MEDIUM, you lose the option of having the iRMX I Operating System dynamically allocate stacks for tasks that are created dynamically. Therefore, you must anticipate the stack requirements of each task, and you must explicitly reserve memory for each stack during configuration.

1.2.2 Decision Algorithm

Before using the flowchart (Figure 1-1) to make your decision, note that three of the boxes are numbered. Each of these three boxes asks you to derive a quantity that represents a memory requirement for a job. To derive each quantity, follow the directions provided below, matching the numbers with Figure 1-1.

1. COMPUTE MEMORY REQUIREMENTS FOR STATIC DATA

   Box 1 asks for the memory required to store the static data for all the tasks of an iRMX I job. Static data consists of all variables other than

   - parameters in a procedure call
   - variables local to a re-entrant PL/M-86 procedure
   - PL/M-86 structures declared to be BASED

To obtain an accurate estimate of this quantity, use the COMPACT size control to compile the code for each task in your job. For each compilation, do the following steps:

a. Find the MODULE INFORMATION area at the end of the listing. In this area is a value labeled VARIABLE AREA SIZE and another labeled CONSTANT AREA SIZE.

b. Compute the static data size for each compilation by adding the VARIABLE AREA SIZE to the CONSTANT AREA SIZE.

c. Once you have computed the static data size for each compilation in the job, add them to obtain the static data size for the entire job.

2. **COMPUTE MEMORY REQUIREMENTS FOR CODE**

Box 2 asks for the memory required to store the code for all the tasks of an iRMX I job. To obtain this estimate, do the following steps:

a. Using the COMPACT size control, compile the code for each task in your job.

b. For each compilation, find the MODULE INFORMATION area at the end of the listing. In this area is a value labeled CODE AREA SIZE. This is how much memory you need to store the code generated by this compilation.

c. Add the code requirements for all the compilations in the job. The sum is the code requirement for the entire job.

3. **COMPUTE MEMORY REQUIREMENTS FOR STACK**

Box 3 asks for the memory required to store the stacks of all the tasks in an iRMX I job. If you plan to have the iRMX I Operating System create your stacks dynamically, your stack requirement (for the purpose of the flowchart) is zero.

If you plan to create the stacks yourself, you can estimate the memory requirements by doing the following steps:

a. Refer to the MODULE INFORMATION AREA of the compilation listings you obtained while working with Box 2. In this area is a value labeled MAXIMUM STACK SIZE.

b. To this number, add the system stack requirement that you can determine by following the procedure in Chapter 8.

c. The result is an estimate of the stack requirement for one compilation.

To compute the requirements for the entire job, add the requirements for all the compilations in the job.

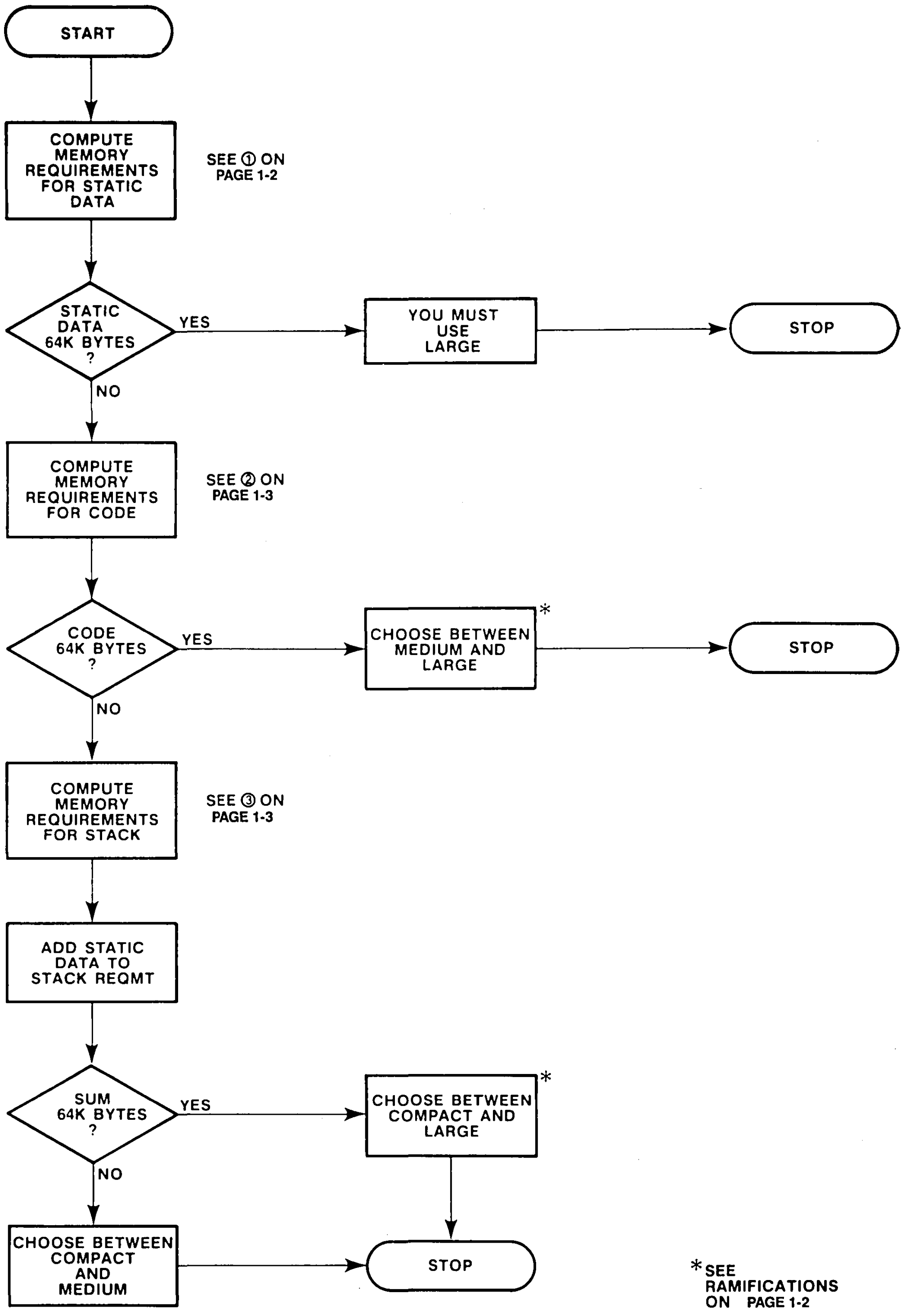
START
COMPUTE MEMORY REQUIREMENTS FOR STATIC DATA
SEE ① ON PAGE 1-2
STATIC DATA 64K BYTES ?
YES
NO
YOU MUST USE LARGE
STOP
COMPUTE MEMORY REQUIREMENTS FOR CODE
SEE ② ON PAGE 1-3
CODE 64K BYTES ?
YES
NO
CHOOSE BETWEEN MEDIUM AND LARGE *
STOP
COMPUTE MEMORY REQUIREMENTS FOR STACK
SEE ③ ON PAGE 1-3
ADD STATIC DATA TO STACK REQMT
SUM 64K BYTES ?
YES
NO
CHOOSE BETWEEN COMPACT AND LARGE *
CHOOSE BETWEEN COMPACT AND MEDIUM
STOP
* SEE RAMIFICATIONS ON PAGE 1-2

Figure 1-1. Decision Algorithm for Size Control x-295

INTERFACE PROCEDURES AND LIBRARIES 2

2.1 INTRODUCTION

You should read this chapter if you write programs that use iRMX I system calls. You should be familiar with the following concepts:

- System calls
- The process of linking object modules
- Object libraries
- PL/M-86 size control

Familiarity with interface procedures is a prerequisite to understanding several of the programming techniques discussed later in this manual. This chapter defines the concept of an interface procedure and explains how it is used in the iRMX I Operating System.

2.2 DEFINITION OF INTERFACE PROCEDURE

The iRMX I Operating System uses interface procedures to simplify calling one software module from another. To illustrate the usefulness of interface procedures, let's examine what happens without them.

Suppose you are writing an application task that will run in some hypothetical operating system. Figure 2-1 shows your application task calling two system procedures. If the system calls are direct (without an interface procedure serving as an intermediary), the application task must be bound to the system procedures either during compilation or during linking. Such binding causes your application task to depend on the memory location of the system procedures.

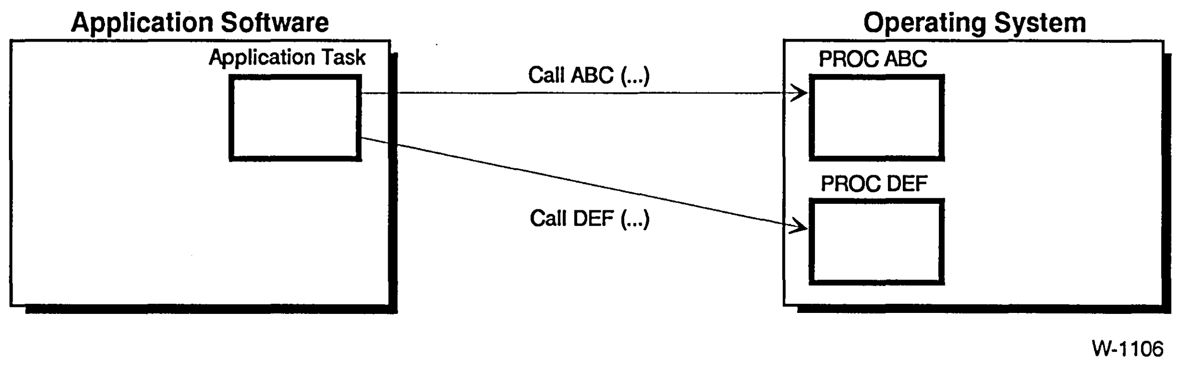

Figure 2-1. Direct Location-Dependent Invocation

Now suppose that someone updates your operating system. If during this updating some of the system procedures are moved to different memory locations, then your application software must be relinked to the new operating system.

Some techniques for calling system procedures do not assume unchanging memory locations. However, most of these techniques are complex (Figure 2-2) and assume that you are familiar with the interrupt architecture of the processor.

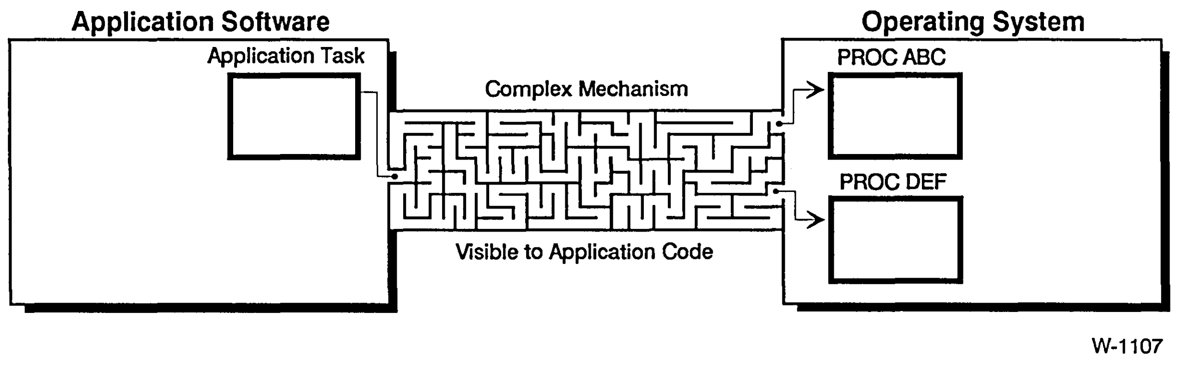

Figure 2-2. Complex Location-Independent Invocation

The iRMX I Operating System uses interface procedures to mask the details of location-independent invocation from the application software (Figure 2-3). Whenever you need to call a system procedure from application code, use a simple procedure call (system call). This system call invokes an interface procedure which, in turn, invokes the actual system procedure.

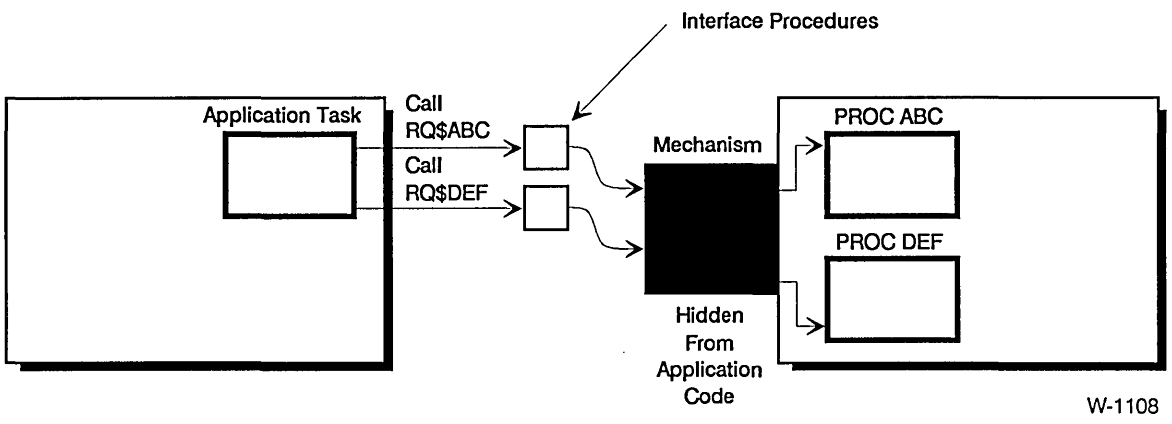

Figure 2-3. Simple Invocation Using an Interface Procedure

2.3 INTERFACE LIBRARIES

The iRMX I Operating System provides a set of object code libraries containing interfaces to system calls. These interface procedures preserve position independence while enabling you to invoke system calls as simple PL/M-86 procedures.

During configuration you must link your application software to the proper object libraries. Table 2-1 shows the correlation between subsystems of the iRMX I Operating System, the PL/M-86 size control, and the interface libraries. To find out which libraries you must link to, find the column that specifies the PL/M-86 size control that you are using. Then find the rows that specify the subsystems of the iRMX I Operating System that you are using. You must link to the libraries named at the intersections of the column and the rows.

Table 2-1. Interface Libraries and iRMX® I Subsystems

	COMPACT	LARGE OR MEDIUM
NUCLEUS	RPIFC.LIB	RPIFL.LIB
BASIC I/O SYSTEM	IPIFC.LIB	IPIFL.LIB
EXTENDED I/O SYSTEM	EPIFC.LIB	EPIFL.LIB
APPLICATION LOADER	LPIFC.LIB	LPIFL.LIB
HUMAN INTERFACE	HPIFC.LIB	HPIFL.LIB
UNIVERSAL DEVELOPMENT INTERFACE	COMPAC.LIB	LARGE.LIB

2.4 iRMX® I EXTERNAL DECLARATION INCLUDE FILES

iRMX I External Declaration INCLUDE Files are designed to stay permanently in one location on individual development systems and provide the PL/M-86 external procedure declarations for all iRMX I and UDI system calls. An INCLUDE file cuts out duplication of statements in your source code modules. The declarations are written once, placed in an INCLUDE file, and then used instead of repeating the actual declaration in each module.

An example of how to use the INCLUDE file NCRTSK.EXT is shown below. It declares the Nucleus System Call CREATE$TASK to be external.

```
$INCLUDE(:RMX86:INC/NCRTSK.EXT)
```

2.4.1 Procedure for Combining Include Files

The external declaration of each system call in iRMX I Release 8 is in a separate file for compatibility with previous releases. Recent releases of the PL/M-86 compiler make these separate files unnecessary. Intel also provides one INCLUDE file per layer in the /rmx86/inc directory.

The external declaration files for each layer are

N = NUCLUS.EXT

I = IOS.EXT

L = LOADER.EXT

H = HI.EXT

U = UDI.EXT

NOTE

If you want to protect certain system calls from general use, the system manager can make the external declarations unavailable, either individually or in the concatenated files.

2.4.2 File Names of the System Call Files in PL/M-86 Programs

Tables 2-2 through 2-7 list all the system calls and their corresponding file names for the following layers:

- Nucleus
- Basic I/O System
- Extended I/O System
- Human Interface
- Application Loader
- Universal Development Interface

Table 2-2. Nucleus System Call File Names

System Call	File Name
RQ$ACCEPT$CONTROL	NACCTL.EXT
RQ$ALTER$COMPOSITE	NALCMP.EXT
RQ$CATALOG$OBJECT	NCTOBJ.EXT
RQ$CREATE$COMPOSITE	NCRCMP.EXT
RQ$CREATE$EXTENSION	NCREXT.EXT
RQ$CREATE$JOB	NCRJOB.EXT
RQ$CREATE$MAILBOX	NCRMBX.EXT
RQ$CREATE$PORT	NCRPRT.EXT
RQ$CREATE$REGION	NCRREQ.EXT
RQ$CREATE$SEGMENT	NCRSEG.EXT
RQ$CREATE$SEMAPHORE	NCRSEM.EXT
RQ$CREATE$TASK	NCRTSK.EXT
RQ$DELETE$COMPOSITE	NDLCMP.EXT
RQ$DELETE$EXTENSION	NDLEXT.EXT
RQ$DELETE$JOB	NDLJOB.EXT
RQ$DELETE$MAILBOX	NDLMBX.EXT
RQ$DELETE$PORT	NDLPRT.EXT
RQ$DELETE$REGION	NDLREG.EXT
RQ$DELETE$SEGMENT	NDLSEG.EXT
RQ$DELETE$SEMAPHORE	NDLSEM.EXT
RQ$DELETE$TASK	NDLTSK.EXT
RQ$DISABLE	NDSABL.EXT
RQ$DISABLE$DELETION	NDSDLN.EXT
RQ$ENABLE$DELETION	NENDLN.EXT
RQ$END$INIT$TASK	NEINIT.EXT
RQ$ENTER$INTERRUPT	NENINT.EXT
RQ$EXIT$INTERRUPT	NEXINT.EXT
RQ$FORCE$DELETE	NFRCDL.EXT
RQ$ENABLE	NENABL.EXT
RQ$GET$EXCEPTION$HANDLER	NGTEXH.EXT
RQ$GET$INTERCONNECT	NGTICT.EXT

(continued)

Table 2-2. Nucleus System Call File Names (continued)

System Call	File Name
RQ$GET$LEVEL	NGTLEV.EXT
RQ$GET$POOL$ATTRIB	NGTPAT.EXT
RQ$GET$PRIORITY	NGTPRI.EXT
RQ$GET$SIZE	NGTSIZ.EXT
RQ$GET$TASK$TOKENS	NGTTOK.EXT
RQ$GET$TYPE	NGTTYP.EXT
RQ$INSPECT$COMPOSITE	NINCMP.EXT
RQ$LOOKUP$OBJECT	NLUOBJ.EXT
RQ$OFFSPRING	NOFFSP.EXT
RQ$RECEIVE$CONTROL	NRCCTL.EXT
RQ$RECEIVE$MESSAGE	NRCMES.EXT
RQ$RECEIVE$SIGNAL	NRCSIG.EXT
RQ$RECEIVE$UNITS	NRCUNI.EXT
RQ$RESET$INTERRUPT	NRSINT.EXT
RQ$RESUME$TASK	NRSTSK.EXT
RQ$SEND$CONTROL	NSNCTL.EXT
RQ$SEND$MESSAGE	NSNMES.EXT
RQ$SEND$SIGNAL	NSNSIG.EXT
RQ$SEND$UNITS	NSNUNI.EXT
RQ$SET$EXCEPTION$HANDLER	NSTEXH.EXT
RQ$SET$INTERCONNECT	NSTICT.EXT
RQ$SET$INTERRUPT	NSTINT.EXT
RQ$SET$OS$EXTENSION	NSTOSX.EXT
RQ$SET$POOL$MIN	NSTPMN.EXT
RQ$SET$PRIORITY	NSTPRI.EXT
RQ$SIGNAL$EXCEPTION	NSGEX.EXT
RQ$SIGNAL$INTERRUPT	NSGINT.EXT
RQ$SLEEP	NSLEEP.EXT
RQ$SUSPEND$TASK	NSUTSK.EXT
RQ$UNCATALOG$OBJECT	NUCOBJ.EXT
RQ$WAIT$INTERRUPT	NWTINT.EXT

Table 2-3. Basic I/O System Call File Names

System Call	File Name
RQ$A$ATTACH$FILE	IAATFL.EXT
RQ$A$CHANGE$ACCESS	IACHAC.EXT
RQ$A$CLOSE	IACLOS.EXT
RQ$A$CREATE$DIRECTORY	IACRDR.EXT
RQ$A$CREATE$FILE	IACRFL.EXT
RQ$A$DELETE$CONNECTION	IADLCN.EXT
RQ$A$DELETE$FILE	IADLFL.EXT
RQ$A$GET$CONNECTION$STATUS	IAGTCS.EXT
RQ$A$GET$DIRECTORY$ENTRY	IAGTDE.EXT
RQ$A$GET$EXTENSION$DATA	IAGTED.EXT
RQ$A$GET$FILE$STATUS	IAGTFS.EXT
RQ$A$GET$PATH$COMPONENT	IAGTPC.EXT
RQ$A$OPEN	IAOPEN.EXT
RQ$A$PHYSICAL$ATTACH$DEVICE	IPATDV.EXT
RQ$A$PHYSICAL$DETACH$DEVICE	IPDTDV.EXT
RQ$A$READ	IAREAD.EXT
RQ$A$RENAME$FILE	IARNFL.EXT
RQ$A$SEEK	IASEEK.EXT
RQ$A$SET$EXTENSION$DATA	IASTED.EXT
RQ$A$SPECIAL	IASPEC.EXT
RQ$A$TRUNCATE	IATRUN.EXT
RQ$A$UPDATE	IUPDAT.EXT
RQ$A$WRITE	IAWRIT.EXT
RQ$CREATE$USER	ICRUSR.EXT
RQ$DELETE$USER	IDLUSR.EXT
RQ$ENCRYPT	IENCRP.EXT
RQ$GET$DEFAULT$PREFIX	IGTPFX.EXT
RQ$GET$DEFAULT$USER	IGTUSR.EXT
RQ$GET$GLOBAL$TIME	IGGTIM.EXT
RQ$GET$TIME	IGTTIM.EXT
RQ$INSPECT$USER	IINUSR.EXT
RQ$SET$DEFAULT$PREFIX	ISTPFX.EXT
RQ$SET$DEFAULT$USER	ISTUSR.EXT
RQ$SET$GLOBAL$TIME	ISGTIM.EXT
RQ$SET$TIME	ISTTIM.EXT
RQ$WAIT$IO	IWTIO.EXT

Table 2-4. Extended I/O System Call File Names

System Call	File Name
RQ$CREATE$IO$JOB	ICRIOJ.EXT
RQ$EXIT$IO$JOB	IEXIOJ.EXT
RQ$GET$USER$IDS	IGTUID.EXT
RQ$LOGICAL$ATTACH$DEVICE	ILATDV.EXT
RQ$LOGICAL$DETACH$DEVICE	ILDTDV.EXT
RQ$HYBRID$DETACH$DEVICE	IHDTDV.EXT
RQ$GET$LOGICAL$DEVICE$STATUS	IGTLDS.EXT
RQ$START$IO$JOB	ISTIOJ.EXT
RQ$VERIFY$USER	IVRUSR.EXT
RQ$S$ATTACH$FILE	ISATFL.EXT
RQ$S$CATALOG$CONNECTION	ISCTCN.EXT
RQ$S$CHANGE$ACCESS	ISCHAC.EXT
RQ$S$CLOSE	ISCLOS.EXT
RQ$S$CREATE$DIRECTORY	ISCRDR.EXT
RQ$S$CREATE$FILE	ISCRFL.EXT
RQ$S$DELETE$CONNECTION	ISDLCN.EXT
RQ$S$DELETE$FILE	ISDLFL.EXT
RQ$S$GET$CONNECTION$STATUS	ISGTCS.EXT
RQ$S$GET$FILE$STATUS	ISGTFS.EXT
RQ$S$LOOK$UP$CONNECTION	ISLUCN.EXT
RQ$S$OPEN	ISOPEN.EXT
RQ$S$READ$MOVE	ISRDMV.EXT
RQ$S$RENAME$FILE	ISRNFL.EXT
RQ$S$SEEK	ISSEEK.EXT
RQ$S$SPECIAL	ISSPEC.EXT
RQ$S$TRUNCATE$FILE	ISTRUN.EXT
RQ$S$UNCATALOG$CONNECTION	ISUNCN.EXT
RQ$S$WRITE$MOVE	ISWRMV.EXT

Table 2-5. Human Interface System Call File Names

System Call	File Name
RQ$C$CREATE$COMMAND$CONNECTION	HCRCCN.EXT
RQ$C$DELETE$COMMAND$CONNECTION	HDLCCN.EXT
RQ$C$FORMAT$EXCEPTION	HFMTEX.EXT
RQ$C$GET$CHAR	HGTCHR.EXT
RQ$C$GET$COMMAND$NAME	HGTCMD.EXT
RQ$C$GET$INPUT$CONNECTION	HGTICN.EXT
RQ$C$GET$INPUT$PATHNAME	HGTIPN.EXT
RQ$C$GET$OUTPUT$CONNECTION	HGTOCN.EXT
RQ$C$GET$OUTPUT$PATHNAME	HGTOPN.EXT
RQ$C$GET$PARAMETER	HGTPAR.EXT
RQ$C$SEND$CO$RESPONSE	HSNCOR.EXT
RQ$C$SEND$COMMAND	HSNCMD.EXT
RQ$C$SEND$EO$RESPONSE	HSNEOR.EXT
RQ$C$SET$PARSE$BUFFER	HSTPBF.EXT

Table 2-6. Application Loader System Call File Names

System Call	File Name
RQ$A$LOAD	LALOAD.EXT
RQ$A$LOAD$IO$JOB	LALIOJ.EXT
RQ$S$LOAD$IO$JOB	LSLIOJ.EXT
RQ$S$OVERLAY	LSOVLY.EXT

Table 2-7. Universal Development Interface System Call File Names

System Call	File Name
DQ$ALLOCATE	UALLOC.EXT
DQ$ATTACH	UATACH.EXT
DQ$CHANGE$ACCESS	UCHAC.EXT
DQ$CHANGE$EXTENSION	UCHEXT.EXT
DQ$CLOSE	UCLOSE.EXT
DQ$CREATE	UCREAT.EXT
DQ$DECODE$EXCEPTION	UDCEX.EXT
DQ$DECODE$TIME	UDCTIM.EXT
DQ$DELETE	UDELET.EXT
DQ$DETACH	UDETAC.EXT
DQ$EXIT	UEXIT.EXT
DQ$FILE$INFO	UFLINF.EXT
DQ$FREE	UFREE.EXT
DQ$GET$ARGUMENT	UGTARG.EXT
DQ$GET$CONNECTION$STATUS	UGTCS.EXT
DQ$GET$EXCEPTION$HANDLER	UGTEXH.EXT
DQ$GET$SIZE	UGTSIZ.EXT
DQ$GET$SYSTEM$ID	UGTSID.EXT
DQ$GET$TIME	UGTTIM.EXT
DQ$OPEN	UOPEN.EXT
DQ$OVERLAY	UOVLY.EXT
DQ$READ	UREAD.EXT
DQ$RENAME	URENAM.EXT
DQ$RESERVE$IO$MEMORY	URSIOM.EXT
DQ$SEEK	USEEK.EXT
DQ$SPECIAL	USPECL.EXT
DQ$SWITCH$BUFFER	USWBF.EXT
DQ$TRAP$CC	UTRAPC.EXT
DQ$TRAP$EXCEPTION	UTRPEX.EXT
DQ$TRUNCATE	UTRUNC.EXT
DQ$WRITE	UWRITE.EXT

2.4.3 Procedure for Using the PASCAL86 INCLUDE File

When you write an application using PASCAL, you will need to include the PASCAL86 INCLUDE file with your PASCAL86 source programs. To accomplish this, enter the statement

```
$INCLUDE(:RMX86:INC/RMXPAS.EXT)
```

which will include the external declaration file for all iRMX I system calls in your PASCAL86 program. However, due to the data type checking done by the PASCAL86 compiler, this file may require minor editing before your program will compile correctly. Three Nucleus and one EIOS system calls are affected:

Nucleus System Calls:	RQ$CREATE$JOB
	RQ$CREATE$TASK
	RQ$SIGNAL$EXCEPTION
EIOS System Call:	RQ$CREATE$IO$JOB

These four system calls enable you to specify either an absolute value (i.e., 0 or 10000H) or a POINTER (i.e., @stacktop) as the STACKPTR parameter. No single PASCAL86 data type can be used to declare both types of values.

The file RMXPAS.EXT, as supplied by Intel, contains a declaration for specifying an absolute value. This declaration enables you to use the value zero as the value for the STACKPTR parameter because the zero indicates that the iRMX I Operating System is to provide the location of the application's stack automatically. The value zero is the most commonly used value for the STACKPTR parameter in these system calls.

If you wish to specify your own location for the application's stack, you must edit RMXPAS.EXT as shown below and use comment notation around the form of the declaration no longer needed. The following example changes the declaration of the STACKPTR parameter in the Nucleus call RQ$CREATE$TASK, but the form applies to all four system calls that may need editing. Note that only one STACKPTR declaration can be 'active'.

Change

```
FUNCTION RQCREATETASK(
       PRIORITY     : BYTE;
  VAR  STARTADDRESS : BYTES;
       DATASEG      : WORD;
{ VAR  STACKPTR     : BYTES; }
       STACKPTR     : LONGINT;
       STACKSIZE    : WORD;
       TASKFLAGS    : WORD;
  VAR  EXCEPTR      : BYTES);
```

to read:

```
FUNCTION RQCREATETASK(
       PRIORITY     : BYTE;
  VAR  STARTADDRESS : BYTES;
       DATASEG      : WORD;
  VAR  STACKPTR     : BYTES;
{      STACKPTR     : LONGINT; }
       STACKSIZE    : WORD;
       TASKFLAGS    : WORD;
  VAR  EXCEPTR      : BYTES);
```

Note that you may change any or all four declarations depending on the requirements of your own PASCAL86 program.

2.4.4 Procedure for Using the FORTRAN86 INCLUDE FILE

When you write an application using FORTRAN, you will need to include the FORTRAN86 INCLUDE file with your FORTRAN86 source programs. To accomplish this, enter the statement

```
$INCLUDE(:RMX86:INC/RMXFOR.EXT)
```

which will include the external declaration file for all iRMX I system calls in your FORTRAN86 program. Only those system calls that return values need to be declared so not all system calls are in the file RMXFOR.EXT.

TIMER ROUTINES 3

3.1 INTRODUCTION

You should read this chapter if you write programs that must determine approximate elapsed time. You should be familiar with the following concepts:

- INCLUDE files
- iRMX I interface procedures (see Chapter 2)
- iRMX I tasks
- Initialization tasks
- Using the LINK86 utility

To understand how the timer routines work, you must be fluent in PL/M-86 and know how to use iRMX I regions.

The iRMX I Basic I/O System provides two system calls, GET$TIME and SET$TIME, that supply your application with a timer having units of one second. However, if your application requires no features of the Basic I/O System other than the timer, you can reduce your memory requirements by dropping the Basic I/O System entirely and creating the timer in your application.

This chapter provides the source code needed to build a timer into your application.

3.2 PROCEDURES IMPLEMENTING THE TIMER

Four PL/M-86 procedures implement the timer:

- get_time

  This procedure requires no input parameter and returns a double word (POINTER) value equal to the current contents of the timer in seconds. You can call this procedure any number of times.

- set_time

  This procedure requires a double word (POINTER) input parameter that specifies the value (in seconds) to which you want the timer set. You can call this procedure any number of times.

- init_time

  This procedure creates the timer, initializes it to zero seconds, and starts it running. This procedure requires as input a POINTER to the WORD that will receive the status of the initialization. This status will be zero if the timer is successfully created and nonzero otherwise. Call this procedure only once.

- maintain_time

  Your application does not directly call this procedure. Rather, it runs as an iRMX I task created when your application calls init_time. This task increments the contents of the timer once every second.

3.3 RESTRICTIONS

Keep these two restrictions in mind when using the timer routines:

3.3.1 CALL init_time FIRST

Before calling set_time or get_time, your application must call init_time by calling the init time procedure from your job's initialization task.

3.3.2 ONLY ONE TIMER

These procedures implement only one timer, so you cannot maintain a different timer for each of several purposes. For example, if one job changes the contents of the timer (by using the set_time procedure), it affects all jobs accessing the timer.

3.4 SOURCE CODE

You can compile the following PL/M-86 source code as a single module. This will yield an object module that you can link to your application code.

```
$title('INDEPENDENT TIMER PROCEDURES')
/***************************************************************************
*                                                                         *
* This module consists of four procedures that implement a timer          *
* with one-second granularity.  The outside world has access to only      *
* three of these procedures-                                              *
*                                                                         *
*  init_time                                                              *
*  set_time                                                               *
*  get_time                                                               *
*                                                                         *
* The fourth procedure, maintain time, is invoked by init time and        *
* is run as an iRMX I task to measure time and increment the time         *
* counter.                                                                *
***************************************************************************/
timer:  DO;
/***************************************************************************
*   The following LITERALLY statements are used to improve the            *
*   readability of the code.                                              *
***************************************************************************/
DECLARE
FOREVER                 LITERALLY 'WHILE OFFH',
TOKEN                   LITERALLY 'SELECTOR',
REGION                  LITERALLY 'TOKEN',
E$OK                    LITERALLY '0000OH',
PRIORITY_QUEUE          LITERALLY '1',
TASK                    LITERALLY 'TOKEN';
```

```
/*************************************************************************
*                                                                        *
*   The following INCLUDE statements cause the external procedure       *
*   declarations for some of the iRMX I system calls to be included    *
*   in the source code.                                                  *
*                                                                        *
************************************************************************/
$INCLUDE(:RMX:INC/nuclus.ext)
$subtitle('Local Data')
/*************************************************************************
*   The following variables can be accessed by all of the procedures    *
*   in this module.                                                      *
************************************************************************/
DECLARE
time_region       REGION,          /* Guards access to time in sec */
time_in_ sec      DWORD,           /* Contains time in seconds */
                                   /* Overlay         */
time_in_sec_o     STRUCTURE(       /* used to obtain */
                  low   WORD,      /* high and low   */
                  high  WORD)      /* order words    */
                  AT (@time_in_sec),
data_seg_p        POINTER,         /* Used to obtain loc of data segment */
data_seg_p_o      STRUCTURE(       /* Overlay used to */
                   offset WORD,    /* obtain loc of   */
                   base   WORD)    /* data segment    */
                   AT (@data_seg_p);
```

```
$subtitle('Time maintenance task')
/**************************************************************************
*   maintain_time                                                         *
*                                                                         *
*     This procedure is run as an iRMX I task.  It repeatedly             *
*     executes the following algorithm-                                   *
*                                                                         *
*           Sleep 1 second.                                               *
*           Gain exclusive access to time_in_sec.                         *
*           Add 1 to time_in_sec.                                         *
*           Surrender exclusive access to time_in_sec.                    *
*                                                                         *
*     If the last three steps in the preceding algorithm require          *
*     more than one Nucleus time unit, the time in sec counter            *
*     will run slow.                                                      *
*                                                                         *
*     This procedure must not be called by any procedure other than       *
*     init_time.                                                          *
**************************************************************************/
maintain_time: PROCEDURE REENTRANT;
DECLARE    status  WORD;
timer_loop:
 DO FOREVER;

CALL rq$sleep( 100, @status );  /* Sleep for one
                                   second. */
CALL rq$receive$control         /* Gain exclusive */
        (time_region, @status); /* access.        */
time_in_sec_o.low =             /* Add 1 second */
    time_in_sec_o.low +1;       /* to low order */
                                /* half of timer.*/
IF (time_in_sec_o.low = 0)      /* Handle overflow.*/
THEN time_in_sec_o.high =
     time_in_sec_o.high + 1;
CALL rq$send$control(@status);  /* Surrender access*/
 END timer_loop;
END maintain_time;
```

```
$subtitle('Get_Time')
/*************************************************************************
*   get_time                                                              *
*                                                                         *
*     This procedure is called by the application code to               *
*     obtain the contents of time_in_sec.  This procedure can be         *
*     called any number of times.                                        *
*************************************************************************/
get_time: PROCEDURE DWORD RE-ENTRANT PUBLIC;
DECLARE   time   DWORD,
  status WORD;
CALL rq$receive$control            /* Gain exclusive */
  ( time_region, @status);         /* access.        */
time = time_in_sec;
CALL rq$send$control(@status);     /* Surrender access.*/
RETURN( time );
END get_time;


$subtitle('Set_Time')
/*************************************************************************
*   set_time                                                              *
*                                                                         *
*     Application code can use this procedure to place a specific       *
*     double word value into time_in_sec.  This procedure can be         *
*     called any number of times.                                        *
*************************************************************************/
set_time: PROCEDURE( time ) REENTRANT PUBLIC;
DECLARE time         DWORD,
            status  WORD;
CALL rq$receive$control         /* Gain exclusive access.*/
(time_region, @status);
time_in_sec = time;             /* Set new time. */
CALL rq$send$control(@status);  /* Surrender access. */
END set_time;
```

```
$subtitle('Initialize_Time')
/************************************************************************
*   init_time                                                           *
*                                                                       *
*     This procedure sets the timer to zero and creates a task to       *
*     maintain the timer and a region to ensure exclusive               *
*     access to the timer.  This procedure must be called               *
*     before the first time that get_time or set_time is                *
*     called.  Also, this procedure should be called only once.         *
*     To ensure this, call init_time from your initialization task.     *
*                                                                       *
*     The timer task will run in the job from which this                *
*     procedure is called.                                              *
*                                                                       *
*     If your application experiences many interrupts,                  *
*     the timer may run slow.  You can rectify this                     *
*     problem by raising the priority of the timer                      *
*     task.  To do this, change the 128 in the                          *
*     rq$create$task system call to a smaller number.                   *
*     This change may slow the processing of your                       *
*     interrupts.                                                       *
************************************************************************/
init_time: PROCEDURE(ret_status_p) REENTRANT PUBLIC;
DECLARE ret_status_p       POINTER,
    ret_status          BASED ret_status_p WORD,
    timer_task t        TASK,
    local_status        WORD;
time_in_sec = 0;
time_region = rq$create$region       /* Create a region. */
         (PRIORITY_QUEUE, ret_status_p);
IF (ret status <> E$OK) THEN
RETURN;                              /* Return with error. */
data_seg_p = @data_seg_p;            /* Get contents of
                                        DS register. */
timer_task t = rq$create$task        /* Create timer task. */
        (128,                        /*  priority         */
         @maintain_time,             /*  start addr       */
         data_seg_p_o.base,          /*  data seg base    */
         0,                          /*  stack ptr        */
         512,                        /*  stack size       */
         0,                          /*  task flags       */
         ret_status_p);
IF (ret_status <> E$OK) THEN
  CALL rq$delete$region              /* Since could not */
       (time_region, @local_status); /* create task,    */
                                     /* must delete     */
                                     /* region.         */
END init_time;
END timer;
```

ASSEMBLY LANGUAGE SYSTEM CALLS 4

4.1 INTRODUCTION

You should read this chapter if you will be using iRMX I system calls from programs written in ASM86 assembly language. You should be familiar with the following concepts:

- iRMX I system calls
- iRMX I interface procedures (see Chapter 2)
- PL/M-86 size controls (see Chapter 1)

You should also be familiar with PL/M-86 and fluent in ASM86 assembly language.

This chapter first outlines the process for using an iRMX I system call from an assembly language program. It then directs you to other Intel manuals that provide either background information or details about interlanguage procedure calls.

4.2 CALLING THE SYSTEM

Programs communicate with the iRMX I Operating System by calling interface procedures designed for use with programs written in PL/M-86. So the problem of using system calls from assembly language programs becomes the problem of making your assembly language programs obey the procedure-calling protocol used by PL/M-86. For example, if your ASM86 program uses the SEND$MESSAGE system call, then you must call the rq$send$message interface procedure from your assembly language code.

NOTE

You can read about the techniques for calling PL/M-86 procedures from assembly language in the manual *ASM86 Macro Assembler Operating Instructions for 8086-Based Development Systems*.

4.3 SELECTING A SIZE CONTROL

Before writing assembly language routines that call PL/M-86 interface procedures, you must select a size control (COMPACT, MEDIUM, or LARGE) because conventions for making calls depend on the model of segmentation.

If you write your entire application in assembly language, you can arbitrarily select a size control and use the libraries for the selected control. However, you can obtain a size and performance advantage by using the COMPACT interface procedures, since their procedure calls are all NEAR. The LARGE interface, which has procedures that require FAR procedure calls, is only helpful if your application code is larger than 64K bytes.

On the other hand, if you write some of your application code in PL/M-86, your assembly language code should use the same interface procedures as those used by your PL/M-86 code.

COMMUNICATION BETWEEN iRMX® I JOBS 5

5.1 INTRODUCTION

You should read this chapter if you want to pass information from one iRMX I job to another. You should be familiar with the following concepts:

- iRMX I jobs, including object directories
- iRMX I tasks
- iRMX I segments
- The root job of an iRMX I-based system
- iRMX I mailboxes
- iRMX I physical files or named filesfiles:named
- iRMX I stream filesfiles:stream
- iRMX I type managers and composite objects

In multiprogramming systems, where each of several applications is implemented as a distinct iRMX I job, there is an occasional need to pass information from one job to another. This chapter describes several techniques that you can use to do this.

The techniques are divided into two groups. The first group deals with passing large amounts of information from one job to another. The second group deals with passing iRMX I objects.

5.2 PASSING LARGE AMOUNTS OF DATA BETWEEN JOBS

There are three methods for sending large amounts of information from one job to another:

1. You can create an iRMX I segment and place the information in the segment. Then, using one of the techniques discussed below for passing objects between jobs, you can deliver the segment.

   The **advantages** of this technique are

   - Since this technique requires only the Nucleus, you can use it in systems that do not use other iRMX I subsystems.
   - The iRMX I Operating System does not copy the information from one place to another.

   The **disadvantages** of this technique are

   - The segment will occupy memory until it is deleted, either explicitly (by the DELETE$SEGMENT system call), or implicitly (when the job that created the segment is deleted). Until the segment is deleted, a substantial amount of memory is unavailable for use elsewhere in the system.
   - The application code may have to copy the information into the segment.

2. You can use an iRMX I stream file.

   The **advantages** of this technique are

   - The data need not be broken into records.
   - This technique can easily be changed to Technique 3 (below).

   The **disadvantage** of this technique is that you must configure one or both I/O systems into your application system.

3. You can use either the Extended or the Basic I/O System to write the information onto a mass storage device, from which the job needing the information can read it.

   The **advantages** of this technique are

   - Many jobs can read the information.
   - This technique can easily be changed to Technique 2 (above).
   - The information need not be divided into records.

   The **disadvantages** of this technique are

   - You must incorporate one or both I/O systems into your application system.
   - Device I/O is slower than reading and writing to a stream file.

5.3 PASSING OBJECTS BETWEEN JOBS

Jobs can also communicate with each other by sending objects across job boundaries. You can use any of several techniques to accomplish this, but you should avoid using one seemingly straightforward technique. In the following discussions you will see how to pass objects by using object directories, mailboxes, and parameter objects. You will also see why you should not pass object tokens by embedding them in an iRMX I segment or in a fixed memory location.

Although you can pass any object from one job to another, there is a restriction applying to connection objects. When a file connection created in one job (Job A) is passed to a second job (Job B), the second job (Job B) cannot successfully use the object to do I/O. Instead, the second job (Job B) must create another connection to the same file. The *iRMX® Basic I/O System User's Guide* and the *iRMX® Extended I/O System User's Guide* describe this restriction.

5.3.1 Passing Objects Through Object Directories

Consider a hypothetical system in which tasks in separate jobs must communicate with each other. Specifically, suppose that Task B in Job B must not begin or resume running until Task A in Job A grants permission.

One way to do this synchronization is to use a semaphore. Task B can repeatedly wait at the semaphore until it receives a unit, and Task A can send a unit to the semaphore whenever it wishes to grant permission for Task B to run. If Tasks A and B were within the same job, this would be a straightforward use of a semaphore. However, the two tasks are in different jobs, and this causes some complications.

Specifically, how do Tasks A and B access the same semaphore? For instance, Task A can create the semaphore and access it, but how can Task A provide Task B with a token for the semaphore? The solution is to use the object directory of the root job.

In the following explanation, each of the two tasks must do half of a protocol. The process of creating and cataloging the semaphore is one half, and the process of looking up the semaphore is the other.

For this protocol to succeed, the programmers of the two tasks must agree on a name for the semaphore, and they must agree which task does which half of the protocol. In this example, the semaphore is named permit_sem. Because Task B must wait until Task A grants permission, Task A will create and catalog the semaphore, and Task B will look it up.

Task A does the creating and cataloging as follows:

1. Task A creates a semaphore with no units by calling the CREATE$SEMAPHORE system call. This provides Task A with a token for the semaphore.

2. Task A calls the GET$TASK$TOKENS system call to obtain a token for the root job.
3. Task A calls the CATALOG$OBJECT system call to place a token for the semaphore in the object directory of the root job under the name permit_sem.
4. Task A continues processing, eventually becomes ready to grant permission, and sends a unit to permit_sem.

Task B does the look-up protocol as follows:

1. Task B calls the GET$TASK$TOKENS system call to obtain a token for the root job.
2. Task B calls the LOOKUP$OBJECT system call to obtain a token for the object named permit_sem. If the name has not yet been cataloged, Task B waits until it is.
3. Task B calls the RECEIVE$UNITS system call to request a unit from the semaphore. If the unit is not available, Task A has not yet granted permission and Task B waits. When a unit is available, Task A has granted permission and Task B becomes ready.

   You should be aware of several aspects of this technique:

   - In the example, the object directory technique was used to pass a semaphore. You can use the same technique to pass any type of iRMX I object.
   - The semaphore was passed via the object directory of the root job. The root job's object directory is unique because it is the only object directory to which all jobs in the system can gain access. This accessibility allows one job to "broadcast" an object to any job that knows the name under which the object is catalogued.
   - The object directory of the root job must be large enough to accommodate the names of all the objects passed in this manner. If it is not, it will become full and the iRMX I Operating System will return an exception code when attempts are made to catalogue additional objects.
   - If you use this technique to pass many objects, you could have problems ensuring unique names. To avoid this, use an object directory instead of the root job directory for different sets of jobs. To do this, have one of the jobs catalogue itself in the root job's object directory under a previously set name. The other jobs can then look up the catalogued job and use its object directory rather than that of the root job.
   - In the example, the object-passing protocol was divided into two halves: the create-and-catalogue half and the look-up half. The protocol works correctly regardless of which half runs first.

5.3.2 Passing Objects Through Mailboxes

You can also send objects from one job to another using a mailbox. This is a two-step process in that the two jobs using the mailbox must first use the object directory technique to obtain mutual access to the mailbox, and then they use the mailbox to pass additional objects.

5.3.3 Passing Parameter Objects

One of the parameters of the CREATE$JOB system call is a parameter object. This parameter allows a task in the parent job to pass an object to the newly created job. Once the tasks in the new job begin running, they can obtain a token for the parameter object by calling GET$TASK$TOKENS. The following example illustrates this technique:

Suppose that Task 1 in Job 1 is responsible for spawning a new job (Job 2). Suppose also that Task 1 maintains an array needed by Job 2. Task 1 can pass the array to Job 2 by putting the array into an iRMX I segment and designating the segment as the parameter object in the CREATE$JOB system call. Then the tasks of Job 2 can call the GET$TASK$TOKENS system call to obtain a token for the segment.

In the example, the parameter object is a segment. However, you can use this technique to pass any kind of iRMX I object.

5.3.4 Avoid Passing Objects Through Segments or Fixed Memory Locations

In the current version of the iRMX I Operating System, tokens remain unchanged when objects are passed from job to job. However, Intel reserves the right to modify this rule. If you pass objects from one job to another and you want your software to be able to run on future releases of the iRMX I Operating System, follow these guidelines:

- Never pass a token from one job to another by placing the token in an iRMX I segment and then passing the segment.
- Never pass a token from one job to another by placing the token in any memory location that both jobs access.

5.3.5 Comparison of Object-Passing Techniques

Consider these guidelines when deciding how to pass an object between jobs:

- If you are passing only one object from a parent job to a child job, use the parameter object when the parent creates the child.
- If you are passing only one object but not from parent to child, use the object directory technique. It is simpler than using a mailbox.
- If you need to pass more than one object at a time, you can use any of the following techniques:
  - Assign an order to the objects and send them to a mailbox where the receiving job can pick them up in order.
  - Give each of the objects a name and use an object directory.
  - Write a simple type manager that packs and unpacks a set of objects. Then pass the set of objects as one composite object.

DEADLOCK AND DYNAMIC MEMORY ALLOCATION 6

6.1 INTRODUCTION

You should read this chapter if you write tasks that dynamically allocate memory, send messages, create objects, or delete objects. You should be familiar with the following concepts:

- Memory management in the iRMX I Operating System
- Using either iRMX I semaphores or regions to obtain mutual exclusion

Memory deadlock is not hard to diagnose or correct but it is difficult to detect. Because memory deadlock often occurs under unusual circumstances, it can lie dormant throughout development and testing, only to bite you when your back is turned. This chapter provides some special techniques to help prevent memory deadlock.

6.2 HOW MEMORY ALLOCATION CAUSES DEADLOCK

The following example illustrates the concept of memory deadlock and shows the danger that iRMX I tasks can face when they allocate memory dynamically.

Suppose that the following circumstances exist for Tasks A and B, which belong to the same job:

- Task A has lower priority than Task B.
- Each task wants two iRMX I segments of a given size, and each asks for the segments by calling CREATE$SEGMENT repeatedly until both segments are acquired.
- The job's memory pool contains only enough memory to satisfy two of the requests.
- Task B is asleep and Task A is running.

Now suppose that the following events occur in the order listed:

1. Task A gets its first segment.

2. An interrupt occurs and Task B wakes up. Since Task B has higher priority than Task A, Task B becomes the running task.
3. Task B gets its first segment.

The two tasks are now deadlocked. Task B remains running and continues to ask for its second segment. Not only are both of the tasks unable to progress, but Task B is consuming a great deal, perhaps all, of the processor time. At best, the system is seriously degraded.

This deadlock problem does not usually occur during debugging because the order of events is critical. Note that the key event in the deadlock example is the awakening of Task B just after Task A invokes the first CREATE$SEGMENT system call, but just before Task A invokes the second CREATE$SEGMENT call. Because this critical sequence of events occurs only rarely, a "thoroughly debugged" system might, after a period of flawless performance, suddenly fail.

Such intermittent failures are costly to deal with once your product is in the field. Thus, the most economical method for dealing with memory deadlock is to prevent it.

6.3 SYSTEM CALLS THAT CAN LEAD TO DEADLOCK

A task cannot cause memory deadlock unless it dynamically allocates memory. Tasks allocate memory only by using system calls. If your task uses any of the following system calls, you must take care to prevent deadlock:

- Any system call that creates an object
- Any system call belonging to a subsystem other than the Nucleus
- SEND$MESSAGE
- DELETE$JOB
- DELETE$EXTENSION

If a task uses none of the preceding system calls, it cannot deadlock as a result of memory allocation.

6.4 PREVENTING MEMORY DEADLOCK

Using one of the following techniques will eliminate memory deadlock from your system:

- When a task receives an E$MEM condition code, the task should not endlessly repeat the system call that led to the code. Rather, it should repeat the call only a predetermined number of times. If the task still receives the E$MEM condition, it should delete all its unused objects and try again. If the E$MEM code is still received, the task should sleep for a while and then reissue the system call.

- If you have designed your system so a job cannot borrow memory from the pool of its parent, you can use an iRMX I semaphore or region to govern access to the memory pool. Then, when a task requires memory, it must first gain exclusive access to the job's memory pool. Only after obtaining this access may the task issue any of the system calls listed above.

  The task's behavior should then depend on whether the system can satisfy all the task's memory requirements:

  -- If the system cannot satisfy all requirements, the task should delete any objects that were created and surrender the exclusive access. Then the task should again request exclusive access to the pool.

  -- If, on the other hand, all requests are satisfied, the task should surrender exclusive access and begin using the objects.

  This technique prevents deadlock by returning unused memory to the memory pool, where another task may use it.

- If you have designed your system so a job cannot borrow memory from the pool of its parent, prevent the tasks within the job from directly competing for the memory in the job's pool. You can do this by allowing no more than one task in each job to use the system calls listed earlier.

GUIDELINES FOR STACK SIZES 7

7.1 INTRODUCTION

This chapter is for three kinds of readers:

- Those who write tasks that create iRMX I jobs or tasks.
- Those who write interrupt handlers.
- Those who write tasks to be loaded by the Application Loader or tasks to be invoked by the Human Interface.

You should be familiar with the iRMX I Debugger, and you should know which system calls the various subsystems of the iRMX I Operating System provide. You also should know the difference between maskable and nonmaskable interrupts.

This chapter will help you compute how much stack you must specify in the system call that creates a job or task. If you are writing an interrupt handler, this chapter informs you of stack size limitations to which you must adhere. If you are writing a task to be loaded by the Application Loader or invoked by the Human Interface, this chapter shows you how much stack to reserve during the linking and locating process.

7.2 STACK SIZE LIMITATION FOR INTERRUPT HANDLERS

Many tasks running in the iRMX I Operating System are subject to two kinds of interrupts: maskable and nonmaskable. When these interrupts occur, the associated interrupt handlers use the stack of the interrupted task. As a result, you must know how much of your task's stack to reserve for these interrupt handlers.

The iRMX I Operating System assumes that all interrupt handlers, including those that you write, require no more than 128 (decimal) bytes of stack, even if a task is interrupted by both a maskable and a nonmaskable interrupt. If when writing an interrupt handler you fail to adhere to this limitation, you risk stack overflow in your system.

To stay within the 128 (decimal) byte limitation, you must restrict the number of local variables that the interrupt handler stores on the stack. For interrupt handlers serving maskable interrupts, you may use as many as 20 (decimal) bytes of stack for local variables. For handlers serving the nonmaskable interrupt, you may use no more than 10 (decimal) bytes. The balance of the 128 bytes is consumed by the SIGNAL$INTERRUPT system call and by storing the registers on the stack.

For more information about interrupt handlers, refer to the *iRMX® I Nucleus User's Guide*.

7.3 STACK GUIDELINES FOR CREATING TASKS AND JOBS

Whenever you invoke a system call to create a task, you must specify the size of the task's stack. Since every new job has an initial task created simultaneously with the job, you must also specify a stack size whenever you create a job.

When you specify a task's stack size, you should do so carefully. If you specify a number that is too small, your task might overflow its stack and write over information following the stack. This condition can cause your system to fail. If you specify a number that is too large, the excess memory will be wasted. Ideally, you should specify a stack size that is only slightly larger than what is actually required.

This chapter provides two techniques for estimating the size of your task's stack. One technique is arithmetic, and the other is empirical. For best results, you should start with the arithmetic technique and then use the empirical technique for tuning your original estimate.

7.4 STACK GUIDELINES FOR TASKS TO BE LOADED OR INVOKED

If you are creating a task to be loaded by the Application Loader or invoked by the Human Interface, you must specify the size of the task's stack during the linking or locating process. The arithmetic and empirical techniques that follow will help you estimate the stack size.

7.5 ARITHMETIC TECHNIQUE

This technique gives a reasonable overestimate of your task's stack size. After you use this technique to obtain a first approximation, you may be able to save several hundred bytes of memory by using the empirical technique described later in this chapter.

The arithmetic technique is based on three elements that affect a task's stack:

- Interrupts
- iRMX I system calls
- Requirements of the task's code (for example, the stack used to pass parameters to procedures or to hold local variables in re-entrant procedures)

You can estimate the size of a task's stack by adding the amount of memory needed to accommodate these elements. The following sections explain how to compute these values.

7.5.1 Stack Requirements for Interrupts

Whenever an interrupt occurs while your task is running, the interrupt handler uses your task's stack while servicing the interrupt. Thus, you must ensure that your task's stack is large enough to accommodate the needs of two interrupt handlers: one for maskable interrupts and one for nonmaskable interrupts. All interrupt handlers used with the iRMX I Operating System are designed to ensure that even if two interrupts occur (one maskable, one not), no more than 128 (decimal) bytes of stack are required by the interrupt handlers.

7.5.2 Stack Requirements for System Calls

When your task invokes an iRMX I system call, the processing associated with the call uses some of your task's stack. The amount of stack required depends on which system calls you use.

Table 7-1 shows how many bytes of stack your task must have to support various system calls. To find out how much stack you must allocate for system calls, compile a list of all the system calls that your task uses. Scan Table 7-1 to find which of your system calls requires the most stack. By allocating enough stack to satisfy the requirements of the most demanding system call, you can satisfy the requirements of all system calls used by your task.

Table 7-1. Stack Requirements for System Calls

System Calls	Bytes (Decimal)
S$SEND$COMMAND C$GET$INPUT$PATHNAME C$GET$OUTPUT$PATHNAME C$GET$INPUT$CONNECTION C$GET$OUTPUT$CONNECTION	800
All other Human Interface System Calls	600
S$LOAD$IO$JOB	440
A$LOAD$IO$JOB A$LOAD S$OVERLAY EXTENDED I/O SYSTEM CALLS	400 400
BASIC I/O SYSTEM CALLS CREATE$JOB DELETE$EXTENSION DELETE$JOB DELETE$TASK FORCE$DELETE RESET$INTERRUPT	300 225
All Other Nucleus Calls	125

7.5.3 Computing the Size of the Entire Stack

To compute the size of the entire stack, add the following three numbers:

- The number of bytes required for interrupts (128 decimal bytes)
- The number of bytes required for system calls
- The amount of stack required by the task's code segment

This sum is a reasonable estimate of your task's stack requirements. For more accuracy, use the sum as a starting point for the empirical fine tuning described below.

7.6 EMPIRICAL TECHNIQUE

This technique starts with a large stack and uses the iRMX I Debugger to determine how much of the stack is unused. Once you have found out how much stack is unused, you can modify your task- and job-creation system calls to create smaller stacks.

The cornerstone of this technique is the iRMX I Dynamic Debugger. To use the Debugger, you must include it when you configure your application system. Refer to the *iRMX® I Interactive Configuration Utility Reference Manual* for detailed information.

The INSPECT TASK command of the Dynamic Debugger provides a display that includes the number of bytes of stack that have not been used since the task was created. If you let your task run a sufficient length of time, you can use the INSPECT TASK command to find out how much excess memory is allocated to your task's stack. Examine the stack area and check how much of it still contains the stack's initial value (0C7H). You can then adjust the stack-size parameter of the system call to reserve less stack.

The only judgment you must exercise when using this technique is deciding how long to let your task run before obtaining your final measurement. If you do not let the task run long enough, it might not meet the most demanding combination of interrupts and system calls. This could cause you to underestimate your task's stack requirement and thus lead to a stack overflow in your final system.

Underestimation of stack size is a risk inherent in this technique. For example, your task might be written to use its peak demand for stack only once every two months. Yet you probably don't want to let your system run for two months just to save several hundred bytes of memory. You can avoid such excessive trial runs by padding the results of shorter runs. For instance, you might run your task for 24 hours and then add 200 (decimal) bytes to the maximum stack size. This padding reduces the probability of overflowing your task's stack in your final system.

INDEX

intel®

REQUEST FOR READER'S COMMENTS

Intel's Technical Publications Departments attempt to provide publications that meet the needs of a Intel product users. This form lets you participate directly in the publication process. Your commen 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 th publication. If you have any comments on the product that this publication describes, please conta 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 suggestio for improvement.

3. Is this the right type of publication for your needs? Is it at the right level? What other types o 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 (COUNTRY) ____ ZIP CODE ____

Please check here if you require a written reply ☐

E'D LIKE YOUR COMMENTS . . .

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

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

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 79 HILLSBORO, OR

POSTAGE WILL BE PAID BY ADDRESSEE

Intel Corporation
OMSO Technical Publications, MS: HF3-72
5200 N.E. Elam Young Parkway
Hillsboro, OR 97124-9978

INTERNATIONAL SALES OFFICES

INTEL CORPORATION
3065 Bowers Avenue
Santa Clara, California 95051

BELGIUM
Intel Corporation SA
Rue des Cottages 65
B-1180 Brussels

DENMARK
Intel Denmark A/S
Glentevej 61-3rd Floor
dk-2400 Copenhagen

ENGLAND
Intel Corporation (U.K.) LTD.
Piper's Way
Swindon, Wiltshire SN3 1RJ

FINLAND
Intel Finland OY
Ruosilante 2
00390 Helsinki

FRANCE
Intel Paris
1 Rue Edison-BP 303
78054 St.-Quentin-en-Yvelines Cedex

ISRAEL
Intel Semiconductors LTD.
Atidim Industrial Park
Neve Sharet
P.O. Box 43202
Tel-Aviv 61430

ITALY
Intel Corporation S.P.A.
Milandfiori, Palazzo E/4
20090 Assago (Milano)

JAPAN
Intel Japan K.K.
Flower-Hill Shin-machi
1-23-9, Shinmachi
Setagaya-ku, Tokyo 15

NETHERLANDS
Intel Semiconductor (Netherland B.V.)
Alexanderpoort Building
Marten Meesweg 93
3068 Rotterdam

NORWAY
Intel Norway A/S
P.O. Box 92
Hvamveien 4
N-2013, Skjetten

SPAIN
Intel Iberia
Calle Zurbaran 28-IZQDA
28010 Madrid

SWEDEN
Intel Sweden A.B.
Dalvaegen 24
S-171 36 Solna

SWITZERLAND
Intel Semiconductor A.G.
Talackerstrasse 17
8125 Glattbrugg
CH-8065 Zurich

WEST GERMANY
Intel Semiconductor G.N.B.H.
Seidlestrasse 27
D-8000 Munchen

intel

INTEL CORPORATION
3065 Bowers Avenue
Santa Clara, California 95051
(408) 987-8080