iRMX™ 86 PROGRAMMING TECHNIQUES

CONTENTS

CONTENTS (continued)

TABLES

FIGURES

***

CHAPTER 1
SELECTING A PL/M-86 SIZE CONTROL

This chapter applies to you only if you have decided to program your iRMX 86 tasks using PL/M-86. In order to understand the following explanation, you should be familiar with

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

PURPOSE OF THIS CHAPTER

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, consequently, greatly affects the amount of memory required to store your application's object code.

The following section explains which size control to use in order to produce the smallest object program while still satisfying the requirements of your system.

MAKING THE SELECTION

When you compile your programs using the PL/M-86 SMALL control, all POINTER values are 16 bits long. This leads to a number of restrictions, including the inability to address the contents of an iRMX 86 segment that has been received from another job. Because of these restrictions, the iRMX 86 Operating System is currently not compatible with PL/M-86 procedures compiled using the SMALL size control.

Since you cannot use the SMALL size control, you must choose between COMPACT, MEDIUM and LARGE. The algorithm for selecting a size control is presented later in this chapter. However, before you examine the algorithm, you should be aware that your choice can place restrictions on your system.

RAMIFICATIONS OF YOUR SELECTION

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

Restrictions Associated With Compact

If you decide to use PL/M-86 COMPACT, you will not be able to use exception handlers. However, you can still process exceptional conditions by dealing with them in your task's code.

Restrictions Associated With Medium

If you decide to use PL/M-86 MEDIUM, you lose the option of having the iRMX 86 Operating System dynamically allocate stacks for tasks that are created dynamically. This means that you must anticipate the stack requirements of each such task, and you must explicitly reserve memory for each stack during the process of configuring the system.

DECISION ALGORITHM

Before you attempt to use 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 of your iRMX 86 job. In order to derive the quantity requested in each of the boxes, follow the directions provided below in the section having the same number as the box.

1. COMPUTE MEMORY REQUIREMENTS FOR STATIC DATA

   Box 1 asks for an estimate of the amount of memory required to store the static data for all the tasks of your iRMX 86 job. Static data consists of all variables other than:

   - parameters in a procedure call
   - variables local to a reentrant PL/M-86 procedure
   - PL/M-86 structures that are 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, find the MODULE INFORMATION area at the end of the listing. Within this area is a quantity labeled VARIABLE AREA SIZE and another labeled CONSTANT AREA SIZE.

   Now you must compute the static data size for each individual compilation by adding the VARIABLE AREA SIZE to the CONSTANT AREA SIZE.

   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 an estimate of the amount of memory required to store the code for all the tasks of your iRMX 86 job. To obtain this estimate, perform the following steps:

- Using the COMPACT size control, compile the code for each task in your job.
- 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 value is the amount of memory required to store the code generated by this individual compilation.
- Sum the code requirements for all the compilations in the job. The result is the code requirement for the entire job.

3. COMPUTE MEMORY REQUIREMENTS FOR STACK

Box 3 asks for an estimate of the amount of memory required to store the stacks of all the tasks in your iRMX 86 job. If you plan to have the iRMX 86 Operating System create your stacks dynamically, your stack requirement (for the purpose of the flowchart) is zero.

If, on the other hand, you plan to create the stacks yourself, you can estimate the memory requirements by performing the following steps. Refer to the MODULE INFORMATION AREA of the compilation listings that you obtained while working with Box 2. Within this area is a value labeled MAXIMUM STACK SIZE. To this number, add the system stack requirement that you can determine by following the procedure in Chapter 8 of this manual. The result is an estimate of the stack requirement for one compilation. To compute the requirements for the entire job, just sum the requirements for all the compilations in the job.

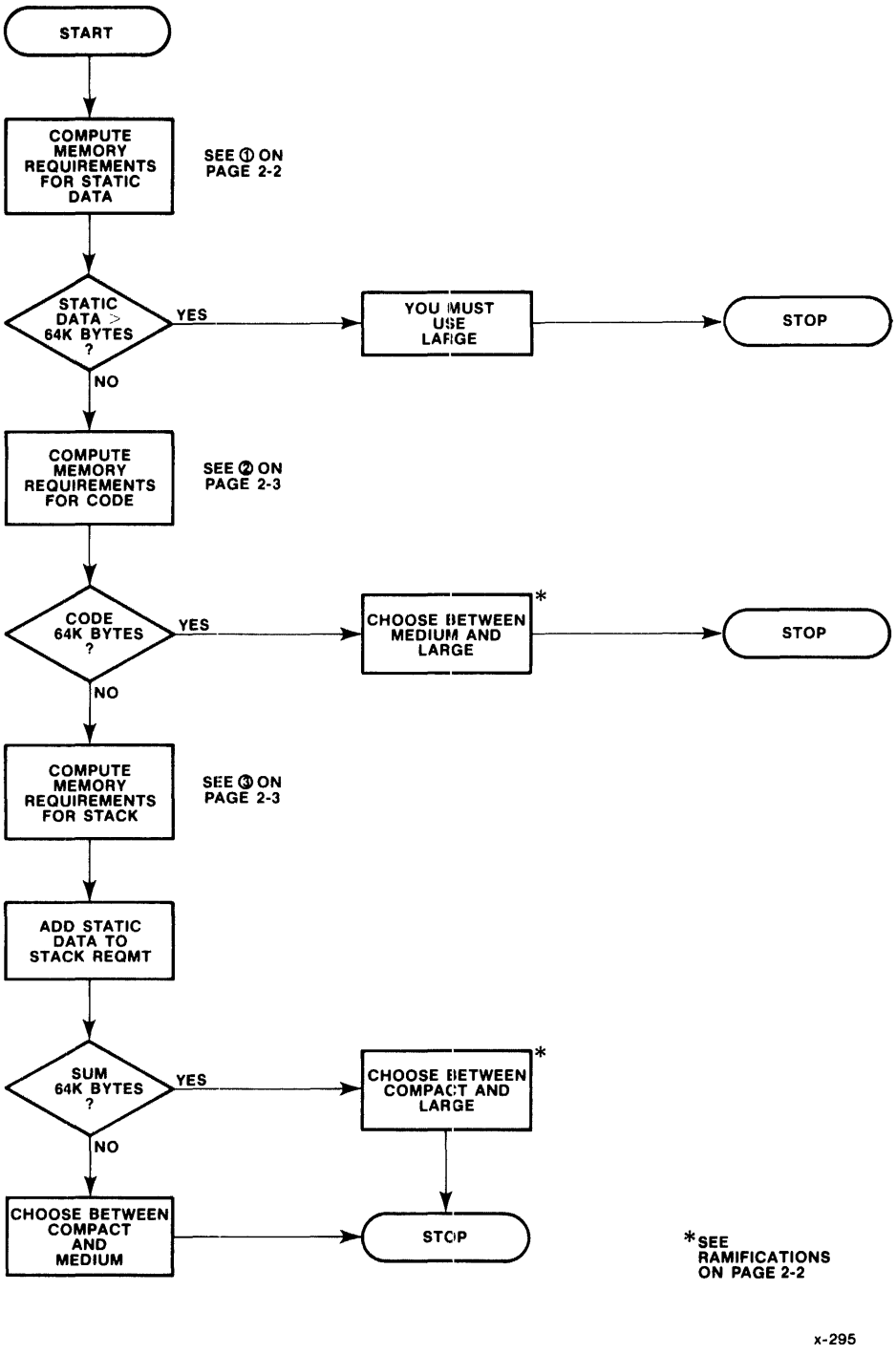

Figure 1-1. Decision Algorithm For Size Control

***

CHAPTER 2
INTERFACE PROCEDURES AND LIBRARIES

This chapter is for anyone who writes programs that use iRMX 86 system calls. In order to understand this chapter, you should be familiar with the following concepts:

- the notion of system call
- the process of linking object modules
- the notion of an object library
- PL/M-86 size control

PURPOSE OF THIS CHAPTER

Familiarity with interface procedures is a prerequisite to understanding several of the programming techniques discussed later in this manual. The primary purpose of this chapter is to define the concept of an interface procedure and explain how it is used in the iRMX 86 Operating System.

DEFINITION OF INTERFACE PROCEDURE

The iRMX 86 Operating System uses interface procedures to simplify the process of calling one software module from another. In order 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 be dependent upon the memory location of the system procedures.

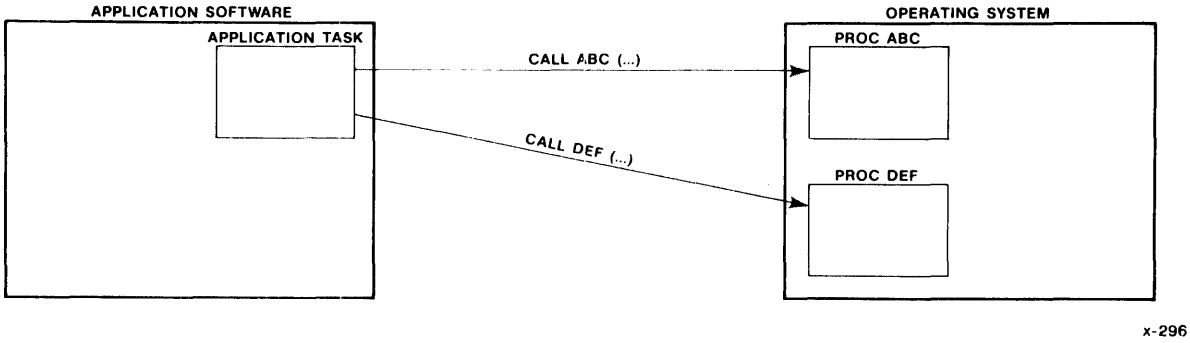

Figure 2-1. Direct Location-Dependent Invocation

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

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

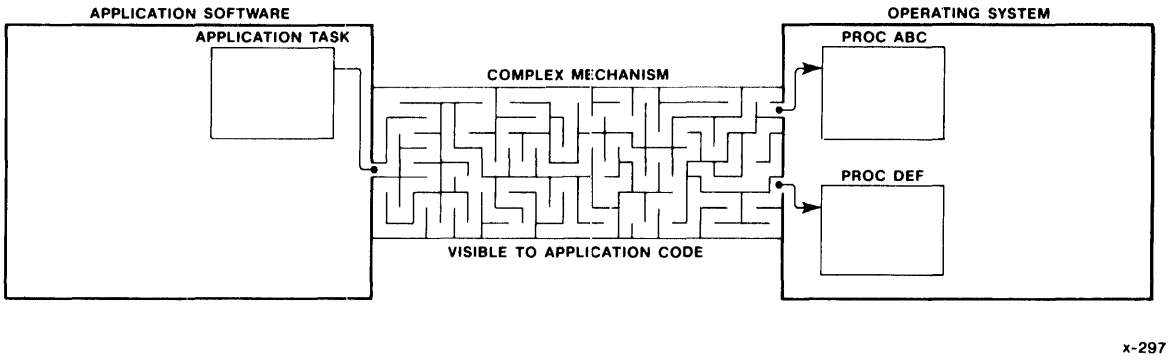

Figure 2-2. Complex Location-Independent Invocation

The iRMX 86 Operating System uses interface procedures to mask the details of location-independent invocation from the application software (Figure 2-3). Whenever application programmers need to call a system procedure from application code, they use a simple procedure call (known as a system call). This system call invokes an interface procedure which, in turn, invokes the actual system procedure.

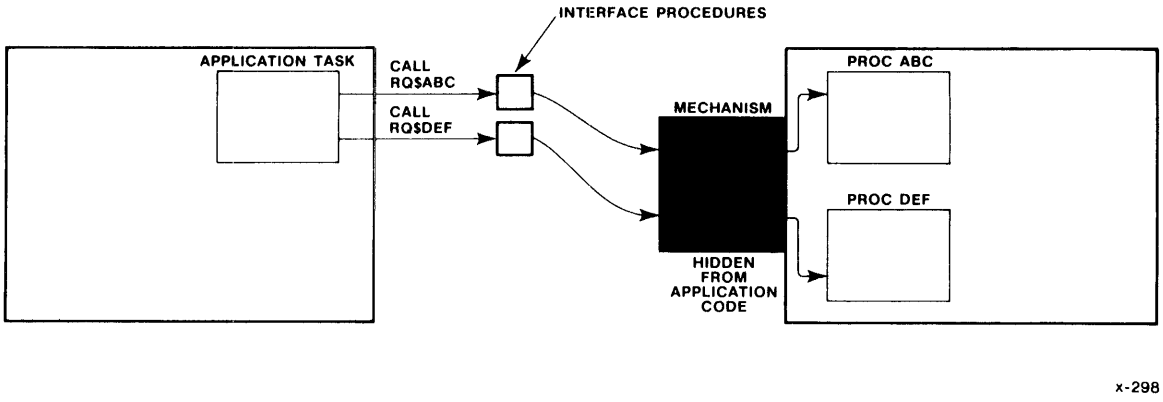

Figure 2-3. Simple Invocation Using An Interface Procedure

INTERFACE LIBRARIES

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

During the process of configuring an application system you must link your application software to the proper object libraries. Table 2-1 shows the correlation between subsystems of the iRMX 86 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, and the rows that specify the subsystems of the iRMX 86 Operating System that you are using. You must link to the libraries that are named at the intersections of the column and the rows.

Table 2-1. Interface Libraries and iRMX™ 86 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
THE UNIVERSAL DEVELOPMENT INTERFACE	COMPAC.LIB	LARGE.LIB

***

CHAPTER 3
TIMER ROUTINES

This chapter is for anyone who writes programs that must determine approximate elapsed time. In order to make use of this chapter, you should be familiar with the following concepts:

- INCLUDE files
- iRMX 86 interface procedures
- iRMX 86 tasks
- initialization tasks
- using the LINK86 utility

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

PURPOSE OF THIS CHAPTER

The iRMX 86 Basic I/O System provides GET$TIME and SET$TIME system calls. These two calls 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 altogether and implementing the timer in your application.

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

PROCEDURES IMPLEMENTING THE TIMER

Four PL/M-86 procedures are used to implement the timer. In brief, the procedures are:

- 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. This procedure can be called 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. This procedure can be called 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 which is to receive the status of the initialization. This status will be zero if the timer is successfully created and nonzero otherwise. This procedure should be called only once.

- maintain_time

  This procedure is not called directly by your application. Rather, it runs as an iRMX 86 task that is created when your application calls init_time. The purpose of this task is to increment the contents of the timer once every second.

RESTRICTIONS

There are two important restrictions that you should keep in mind when using the timer routines:

CALL init_time FIRST

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

ONLY ONE TIMER

These procedures implement only one timer. They do not allow you to 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), all jobs accessing the timer will be affected.

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. However, before compiling these procedures, you must create files containing the external procedure declarations for the iRMX 86 interface procedures. The names of these files are specified in the $INCLUDE statements below.

```
$title('INDEPENDENT TIMER PROCEDURES')
/*************************************************************************
*                                                                        *
*   This module consists of four procedures which implement a timer      *
*   having 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 86 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 0FFH',
            DWORD                   LITERALLY 'POINTER',
            TOKEN                   LITERALLY 'WORD',
            REGION                  LITERALLY 'TOKEN',
            E$OK                    LITERALLY '00000H',
            PRIORITY_QUEUE          LITERALLY '1',
            TASK                    LITERALLY 'TOKEN';
```

```
/*****************************************************************
*                                                                *
*   The following INCLUDE statements cause the external procedure *
*   declarations for some of the iRMX 86 system calls to be included *
*   in the source code.                                          *
*                                                                *
*****************************************************************/

$INCLUDE(:f1:icrtas.ext)  /* rq$create$task interface proc.*/
$INCLUDE(:f1:icrreg.ext)  /* rq$create$region    "       "  */
$INCLUDE(:f1:isleep.ext)  /* rq$sleep            "       "  */
$INCLUDE(:f1:idereg.ext)  /* rq$delete$region    "       "  */
$INCLUDE(:f1:iregio.ext)  /* rq$send$control     "       "  */
                     /* and  rq$receive$control "       "  */



$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
                                               seg.*/

            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 86 task.  It repeatedly           *
*     performs 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 in order to       *
*     obtain the contents of time_in_sec.  This procedure can be         *
*     called any number of times.                                        *
*************************************************************************/

get_time: PROCEDURE DWORD REENTRANT 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;
```

```
$subtitle('Initialize Time')
/*************************************************************************
*   init_time                                                            *
*                                                                        *
*     This procedure zeros the timer, creates a task to                  *
*     maintain the timer, and a region to ensure exclusive               *
*     access to the timer.  This procedure must be called                *
*     before the first time that get_time or set_time is                 *
*     called.  Also, this procedure should be called only                *
*     once.  The easiest way to make sure this happens is to             *
*     call init_time from your initialization task.                      *
*                                                                        *
*     The timer task will run in the job from which this                 *
*     procedure is called.                                               *
*                                                                        *
*     If your application experiences a lot of interrupts,               *
*     the timer may run slow.  You can rectify this                      *
*     problem by raising the priority of the timer                       *
*     task.  To do this, change the 128 in the                           *
*     rq$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 w/ 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;
```

***

CHAPTER 4
ASSEMBLY LANGUAGE SYSTEM CALLS

This chapter is for anyone who wants to use iRMX 86 system calls from programs written in ASM86 assembly language. In order to be able to use system calls from assembly language, you should be familiar with the following concepts:

- iRMX 86 system calls
- iRMX 86 interface procedures
- PL/M-86 size controls

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

PURPOSE OF THIS CHAPTER

The purpose of this chapter is twofold. First, it briefly outlines the process involved in using an iRMX 86 system call from an assembly language program. Second, it directs you to other Intel manuals that provide either background information or details concerning interlanguage procedure calls.

CALLING THE SYSTEM

If you read Chapter 2 of this manual, you found that your programs communicate with the iRMX 86 System by calling interface procedures that are 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$MESSAGE system call, then you must call rq$send$message interface procedure from your assembly language code.

NOTE

The techniques for calling PL/M-86 procedures from assembly language are completely described in the manual ASM86 MACRO ASSEMBLER OPERATING INSTRUCTIONS for 8086-BASED DEVELOPMENT SYSTEMS.

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 upon the model of segmentation.

If all of your application is written 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 advantageous if your application code is larger than 64K bytes.

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

***

CHAPTER 5
COMMUNICATION BETWEEN iRMX™ 86 JOBS

This chapter applies to anyone who wants to pass information from one iRMX 86 job to another. In order to understand this chapter, you must be familiar with the following concepts:

- iRMX 86 jobs, including object directories
- iRMX 86 tasks
- iRMX 86 segments
- the root job of an iRMX 86-based system
- iRMX 86 mailboxes
- iRMX 86 physical files or named files
- iRMX 86 stream files
- iRMX 86 type managers and composite objects

PURPOSE OF THIS CHAPTER

In multiprogramming systems, where each of several applications is implemented as a distinct iRMX 86 job, there is an occasional need to pass information from one job to another. This chapter describes several techniques that you can use to accomplish this.

The techniques are divided into two collections. The first collection deals with passing large amounts of information from one job to another, while the second collection deals with passing iRMX 86 objects.

PASSING LARGE AMOUNTS OF INFORMATION BETWEEN JOBS

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

1) You can create an iRMX 86 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 86 subsystems.
- The iRMX 86 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 means of 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 86 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.

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.
- 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.

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, and 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 86 segment or in a fixed memory location.

Although you can pass any object from one job to another, there is a restriction pertaining 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 perform I/O. Instead, the second job (Job B) must create another connection to the same file. This restriction is discussed in the iRMX 86 BASIC I/O SYSTEM REFERENCE MANUAL and in the iRMX 86 EXTENDED I/O SYSTEM REFERENCE MANUAL.

PASSING OBJECTS THROUGH OBJECT DIRECTORIES

For the purpose of this discussion, 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 perform 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 are within the same job, this would be a straightforward use of a semaphore. But 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 trick is to use the object directory of the root job.

In the following explanation, each of the two tasks must perform 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.

In order 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 performs which half of the protocol. In this example, the semaphore is named permit_sem. And, 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 performs 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 performs 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 then 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.

There are several aspects of this technique that you should be aware of:

- In the example, the object directory technique was used to pass a semaphore. The same technique can be used to pass any type of iRMX 86 object.

- The semaphore was passed via the object directory of the root job. The root job's object directory is unique in that 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 cataloged.

- 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 86 Operating System will return an exception code when attempts are made to catalog additional objects.

- If you use this technique to pass many objects, you could have problems ensuring unique names. If name management becomes a problem, different sets of jobs can adopt the convention of using an object directory other than that of the root job. To accomplish this, one of the jobs catalogs itself in the root job's object directory under an agreed-upon name. The other jobs can then look up the cataloged 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-catalog half, and the look-up half. The protocol works correctly regardless of which half starts to run first.

PASSING OBJECTS THROUGH MAILBOXES

Another means of sending objects from one job to another is to use 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.

PASSING PARAMETER OBJECTS

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

Suppose that Task 1 in Job 1 is responsible for spawning a new job (Job 2). Suppose also that Task 1 maintains an array that is needed by Job 2. Task 1 can pass the array to Job 2 by putting the array into an iRMX 86 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 foregoing example, the parameter object is a segment. However, you can use this technique to pass any kind of iRMX 86 object.

AVOID PASSING OBJECTS THROUGH SEGMENTS OR FIXED MEMORY LOCATIONS

In the current version of the iRMX 86 Operating System, tokens remain unchanged when objects are passed from job to job. However, Intel reserves the right to modify this rule. In other words, 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 86 System, obey the following guidelines:

- Never pass a token from one job to another by placing the token in an iRMX 86 segment and then passing the segment.
- Never pass a token from one job to another by placing the token in any memory location that the two jobs both access.

COMPARISON OF OBJECT-PASSING TECHNIQUES

There are several guidelines to consider 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.

***

CHAPTER 6
SIMPLIFYING CONFIGURATION DURING DEVELOPMENT

For your convenience, the configuration information found in this chapter has been added to the iRMX 86 CONFIGURATION GUIDE. For any information that you might need concerning the following topics, refer to the iRMX 86 CONFIGURATION GUIDE.

- Data segments
- Configuration
- Freezing locations of entry points
- The Interactive Configuration Utility (ICU)
- The LOC86 command
- Padding memory segments

***

CHAPTER 7
DEADLOCK AND DYNAMIC MEMORY ALLOCATION

This chapter is for anyone who writes tasks which dynamically allocate memory, send messages, create objects, or delete objects. In order to understand this chapter, you should be familiar with the following concepts:

- memory management in the iRMX 86 Operating System
- using either iRMX 86 semaphores or regions to obtain mutual exclusion

PURPOSE OF THIS CHAPTER

Memory deadlock is not difficult to diagnose or correct, but it is difficult to detect. Because memory deadlock generally occurs under unusual circumstances, it can lie dormant throughout development and testing, only to bite you when your back is turned. The purpose of this chapter is to provide you with some special techniques that can prevent memory deadlock.

HOW MEMORY ALLOCATION CAUSES DEADLOCK

The following example illustrates the concept of memory deadlock and shows the danger that iRMX 86 tasks can face when they cause memory to be allocated dynamically.

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

- Task A has lower priority than Task B.
- Each task wants two iRMX 86 segments of a given size, and each asks for the segments by calling the CREATE$SEGMENT system call 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 is awakened. Since Task B is of 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 kind of memory allocation deadlock problem is particularly insidious because it quite likely would not occur during debugging. The reason for this is that the order of events is critical in this deadlock situation.

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. Consequently, the most economical method for dealing with memory deadlock is to prevent it.

SYSTEM CALLS THAT CAN LEAD TO DEADLOCK

A task cannot cause memory deadlock unless it causes memory to be allocated dynamically. And the only means for a task to allocate memory is 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.

PREVENTING MEMORY DEADLOCK

Using any one of the following techniques, you can 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 that a job cannot borrow memory from the pool of its parent, you can use an iRMX 86 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 upon whether the system can satisfy all of 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 it may be used by another task.

- If you have designed your system so that a job cannot borrow memory from the pool of its parent, prevent the tasks within the job from directly completing 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.

***

CHAPTER 8
GUIDELINES FOR STACK SIZES

This chapter is for three kinds of readers:

- Those who write tasks that create iRMX 86 jobs or tasks.
- Those who write interrupt handlers.
- Those who write tasks that are to be loaded by the Application Loader or tasks to be invoked by the Human Interface.

In order to understand all of this chapter, you must be familiar with the iRMX 86 Debugger, and you must know which system calls are provided by the various subsystems of the iRMX 86 Operating System. You also must know the difference between maskable and nonmaskable interrupts. Finally, if you are writing an interrupt handler, you must know what an interrupt handler is.

PURPOSE OF THIS CHAPTER

This chapter has three purposes. If your are writing a task that creates a job or another task, the purpose of this chapter is to help you compute the amount of stack that you must specify in the system call that performs the creation. If you are writing an interrupt handler, the purpose of this chapter is to inform you of stack size limitations to which you must adhere. If you are writing a task that is to be loaded by the Application Loader or invoked by the Human Interface, the purpose of this chapter is to show you how much stack to reserve during the linking and locating process.

STACK SIZE LIMITATION FOR INTERRUPT HANDLERS

Many tasks running in the iRMX 86 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. Consequently, you must know how much of your task's stack to reserve for these interrupt handlers.

The iRMX 86 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 expose your system to the risk of stack overflow.

In order 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 86 NUCLEUS REFERENCE MANUAL.

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. And, since every new job has an initial task that is created simultaneously with the job, you must also designate 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 situation can cause your system to fail. On the other hand, if you specify a number that is too large, the excess memory will be wasted. So ideally, you should specify a stack size that is only slightly larger than what is actually required.

This chapter provides you with 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.

STACK GUIDELINES FOR TASKS TO BE LOADED OR INVOKED

If you are creating a task that is 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 in this manual will help you estimate the size of your task's stack.

ARITHMETIC TECHNIQUE

This technique provides you with 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 the fact that there are at most three factors affecting a task's stack. These factors are:

- interrupts
- iRMX 86 system calls
- requirements of the task's code
  (For example, the stack used to pass parameters to procedures or to hold local variables in reentrant procedures.)

You can estimate the size of a task's stack by summing the amount of memory needed to accommodate these factors. The following sections explain how to compute the stack requirements for the first three factors.

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. Consequently, 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 86 Operating system are designed to 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.

STACK REQUIREMENTS FOR SYSTEM CALLS

When your task invokes an iRMX 86 system call, the processing associated with the call uses some of your task's stack. The amount of stack required depends upon which system calls you use.

Table 8-1 tells you 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 8-1 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 8-1. Stack Requirements For System Calls

System Calls	Bytes (Decimal)
S$SEND$COMMAND CGETINPUT$PATHNAME C$GET$OUTPUT$PATHNAME CGETINPUT$CONNECTION C$GET$OUTPUT$CONNECTION ALL OTHER HUMAN INTERFACE SYSTEM CALLS	800 600
S$LOAD$IO$JOB A$LOADIOJOB A$LOAD S$OVERLAY	440 400
EXTENDED I/O SYSTEM CALLS	400
BASIC I/O SYSTEM CALLS	300
CREATE$JOB DELETE$EXTENSION DELETE$JOB DELETE$TASK FORCE$DELETE RESET$INTERRUPT ALL OTHER NUCLEUS CALLS	225 125

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

You can use the sum of these three numbers as a reasonable estimate of your task's stack requirements. If you desire more accuracy, use the sum as a starting point for the empirical fine tuning described later in this chapter.

EMPIRICAL TECHNIQUE

This technique starts with an overly large stack and uses the iRMX 86 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 86 Debugger. In order to use the Debugger, you must include it when you configure your application system. Information on how to do this is provided in the iRMX 86 CONFIGURATION GUIDE.

The Inspect Task command of the 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. Then you can 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 encounter the most demanding combination of interrupts and system calls. This could cause you to underestimate your task's stack requirement and could, consequently, 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 so as 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

Primary references are underscored.

***