Quick Contents

Notational Conventions

Most of the references to system calls in the text and graphics use C syntax instead of PL/M (for example, the system call **send_message** instead of **send$message**). If you are working in C, you must use the C header files, *rmx_c.h*, *udi_c.h*, and *rmx_err.h*. If you are working in PL/M, you must use dollar signs ($) and use the *rmxplm.ext* and *error.lit* header files.

This manual uses the following conventions:

- Syntax strings, data types, and data structures are provided for PL/M and C respectively.
- All numbers are decimal unless otherwise stated. Hexadecimal numbers include the `H` radix character (for example, `0FFH`). Binary numbers include the `B` radix character (for example, `11011000B`).
- Bit 0 is the low-order bit. If a bit is set to 1, the associated description is true unless otherwise stated.
- `Data structures and syntax strings appear in this font.`
- **System call names and command names appear in this font.**
- PL/M data types such as BYTE and SELECTOR, and iRMX data types such as STRING and SOCKET are capitalized. All C data types are lower case except those that represent data structures.
- The following OS layer abbreviations are used. The Nucleus layer is unabbreviated.

AL	Application Loader
BIOS	Basic I/O System
EIOS	Extended I/O System
HI	Human Interface
UDI	Universal Development Interface

- Whenever this manual describes I/O operations, it assumes that tasks use BIOS calls (such as **rq_a_read**, **rq_a_write**, and **rq_a_special**). Although not mentioned, tasks can also use the equivalent EIOS calls (such as **rq_s_read**, **rq_s_write**, and **rq_s_special**) or UDI calls (**dq_read** or **dq_write**) to do the same operations.

Contents

5 Debugging Applications

6 Porting Applications

7 Using Compact and Large Memory Models

8 Using the Flat Memory Model

9 Developing Applications for ROM

10 Developing Applications for Multibus II

Index

Tables

Figures

iRMX® Application Development Environment 1

This manual describes techniques for developing applications on the iRMX® Operating System (OS). You can also use this manual as a porting guide for your iRMX applications.

This manual assumes you are familiar with these concepts:

- Programming in the iRMX environment using either C, PL/M, or Assembler
- Using iRMX jobs, tasks, mailboxes, files, and segments
- Using object module linking
- Using object libraries

See also: iRMX objects, *Introducing the iRMX Operating Systems* and *System Concepts*

Examples Provided with the Operating System

The iRMX OS provides code examples to help you learn about the iRMX application development environment. These examples are in various subdirectories of the */rmx386/demo* directory. This manual gives instructions on compiling and running the examples, which are summarized in Table 1-1.

Table 1-1. Code Examples in this Manual

Example Description	Chapter
C language: Multitasking demo, basic concepts, compiling, binding	Ch. 3
Debug Session (PL/M)	Ch. 5
Porting code: PL/M language differences Porting code: assembly language differences Device Driver Porting (8274)	Ch. 6 Ch. 6 Ch. 6
Using Compact and Large Memory Models	Ch. 7
Using Flat Memory Model	Ch. 8
C language: Multibus II, board scanning C language: Multibus II, creating a data transport protocol port C language: Multibus II, send/receive RSVP C language: Multibus II, send/receive a data chain message C language: Multibus II, sending a message in fragments C language: Multibus II, receiving a message in fragments	Ch. 10 Ch. 10 Ch. 10 Ch. 10 Ch. 10 Ch. 10
Assembly language: Macro definitions for common source code Assembly language: Invoking system calls Assembly language: Interrupt handler Assembly language: OS Extension	Ch. 11 Ch. 11 Ch. 11 Ch. 11
PL/M: External declarations, interface libraries, and binding PL/M: Multitasking, basic concepts, compiling, binding PL/M: <Ctrl-C> handler	Ch. 12 Ch. 12 Ch. 12

Application Development Tools

Intel provides tools for developing iRMX applications for your system, including:

- Assemblers
- Compilers
- Application building utilities
- Debuggers
- Non-Intel tool support

See also: *C Library Reference*

Assemblers

Use the ASM386 assembler to produce code for your application. ASM386 supports Intel386™, Intel486™, and Pentium® microprocessors.

See also: Developing Applications in Assembly Language, Chapter 11, *ASM386 Assembly Language Reference*

Intel Compilers

Use these compilers to develop iRMX applications:

- iC-386
- PL/M-386
- Non-Intel C compilers

The iC-386 compiler supports the ANSI standard for the C programming language with some extensions.

The iC-386 and PL/M-386 compilers produce 32-bit code. Depending on the compiler, non-Intel C compilers produce either 16-bit or 32-bit code.

Note

Many non-Intel compilers can produce C or C++ code. The iRMX OS supports only C code produced with such compilers, not C++ code.

See also: *iC-386 Compiler User's Guide*, *C Library Reference*, *PL/M-386 Programmer's Guide*

Intel compilers offer these features:

- Separate compilation of source code files
- Libraries containing external declaration calls and literal files
- Inter-language programming in C, PL/M, or Assembler
- Support for ROM-based applications
- Code optimization for optimizing code performance or size
- In-line functions and macros to access microprocessors and numeric coprocessors
- Run-time libraries to access floating-point support or the OS interfaces

See also: Your compiler's programmer's manual

Optimizing Code

Use these iC-386 compiler controls to optimize your code:

- The `noalign` control produces compact nonaligned data structures. Data structures used for iRMX system calls require the `noalign` control. Non-Intel C compilers provide data packing features to perform the same function.
- The `optimize` control specifies the optimization level the compiler uses to generate object code. Optimized object code is compact and runs faster but takes longer to compile.
- The `nodebug` control requests that the compiler not produce debug information. This optimizes the code the compiler generates.
- The segmentation controls specify the memory model for an application. Segmentation controls include: compact, large, and flat.

See also: C Compiler-specific Information, Chapter 4,
iC-386 Compiler User's Guide

Non-Intel Compilers

This table lists the non-Intel compilers supported in the iRMX OS.

	Supported in MSVC to version 6
32-bit compiler	yes (flat model)
32-bit linker	yes
librarian	yes
make utility	yes
assembler	yes (in-line)

Application Building Utilities

Application building utilities aid in developing iRMX applications. These utilities include:

- The LIB386 librarian utility organizes object modules into libraries.
- The BND386 binding utility binds object modules to produce an executable module or a module for incremental binding.
- The MAP386 map utility creates cross-reference maps of object modules.
- The BLD386 system builder utility builds a working system. You can configure the Interactive Configuration Utility (ICU) to automatically invoke the BLD386 when generating an application system for the iRMX OS.

See also: Overlays, *System Concepts*,
LIB386, BND386, MAP386, *Intel386 Family Utilities*

Debugging Tools

You can use several tools to debug your iRMX application programs, such as:

Soft-Scope debugger
: For most debugging tasks, use the Soft-Scope debugger. It provides all the tools you need for debugging iRMX applications, including source-level and symbolic debugging capabilities.

SDM System Debug Monitor (SDM)
: A debug monitor for debugging systems, disassembling code, executing breakpoints, displaying memory, and downloading programs.

iRMX System Debugger (SDB)
: A symbolic debugging tool for debugging iRMX applications and system programs. This tool extends the SDM's disassembly functions for interpreting iRMX calls, data structures, and stacks.

See also: Debugging an Application, Chapter 4,
Soft-Scope Debugger User's Guide,
System Debugger Reference

Application Development Process

The iRMX development environment provides the programming tools needed to develop 32-bit applications. Figure 1-1 shows the development process for 32-bit applications using Intel tools. Figure 1-2 shows the development process if you are using non-Intel tools.

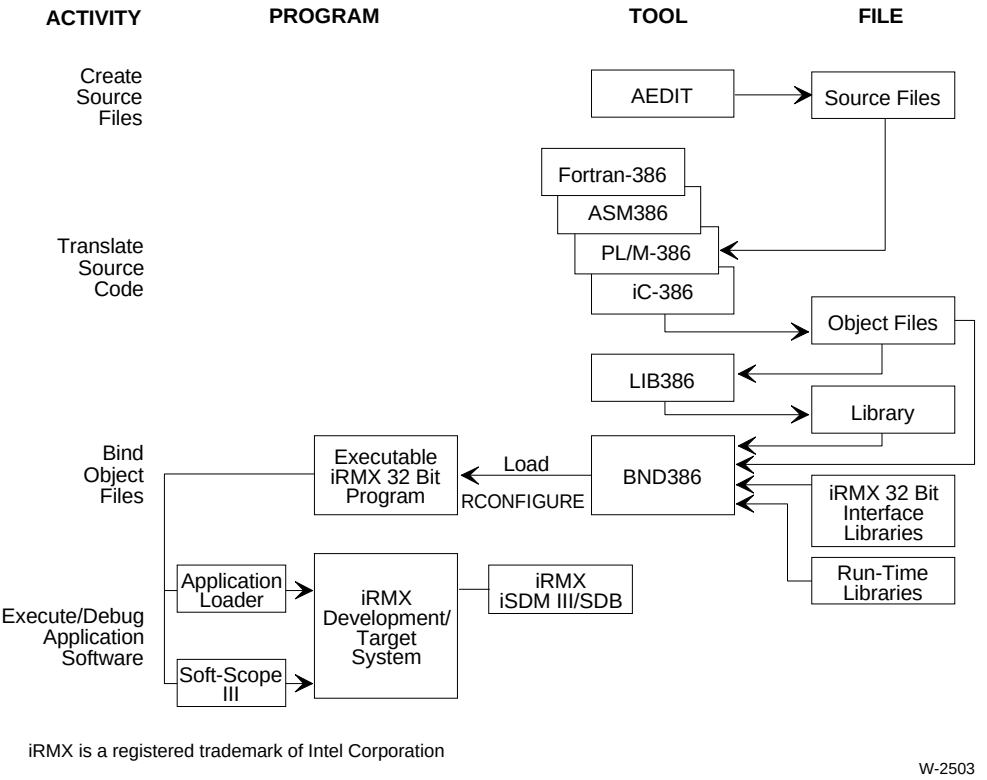


Figure 1-1. The 32-bit Application Development Process (Intel Tools)

You can use the 32-bit compiler and utilities from iRMX using the RUN86 utility. This is user-transparent through aliases provided by the iRMX OS.

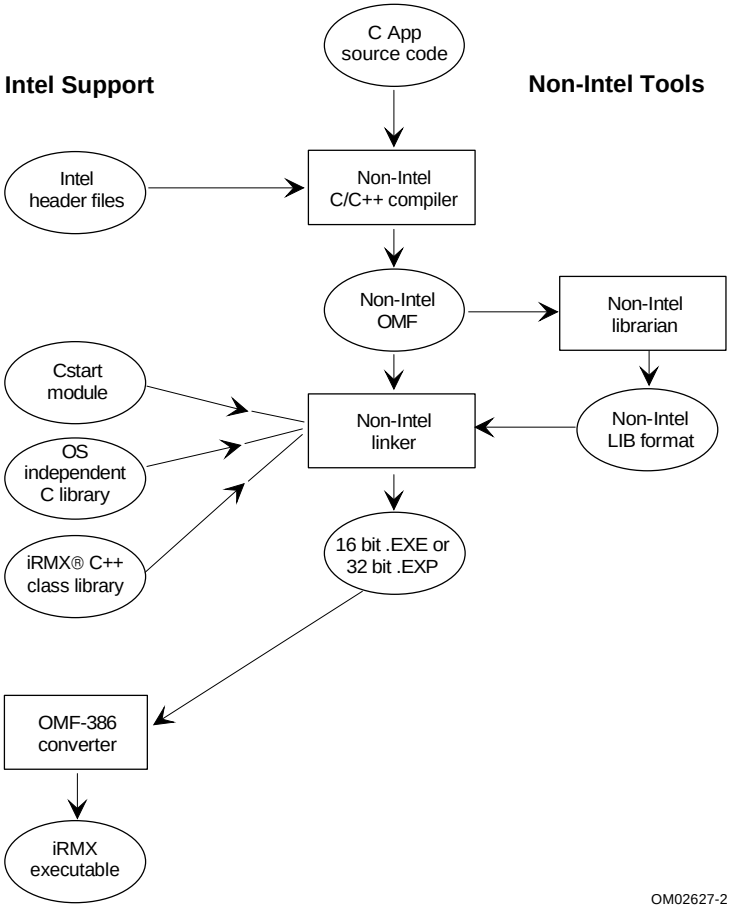


Figure 1-2. The 32-bit Application Development Process (Non-Intel Tools)

□□□

Target Environment Development 2

This chapter describes the Multibus (MB) target file modification and generation on a PC development environment.

Generating Target Files

The Interactive Configuration Utility (ICU) enables you to modify the definition files (*:icu:*.bck*) to create Multibus (MB) target files in a PC-hosted system.

See also: *ICU User's Guide and Quick Reference*

For example, you can generate files on a PC (using DOSRMX or iRMX for PCs), and then copy these files to your target MB system.

Generating a Target File Example

You can use the ICU to generate new target files or modify existing files. In this example, create a new target file by modifying an existing definition file for the SBC 486133SE board. You can create the file on a PC and then copy the file to a Multibus system.

1. Create a working directory called "icutest", attach to this directory, and then copy the definition file to this directory.

```
- crdir icutest <CR>
- af :icutest: <CR>
- copy :icu:486133.bck to $ <CR>
```

2. Invoke the ICU under the DOSRMX or iRMX for PCs OS and select the 486133 definition file.

```
- icu386 486133.bck <CR>
```

3. The ICU outputs this query. Answer with a `y`.

```
Do you want to restore from the file ?  [y]/n: y <CR>
```

4. Answer the next query with a n.

```
Do you want to overwrite input file [y]/n: n <CR>
```

5. At the next query, enter an output name different than that of the definition file.

```
Enter new output file name: icutest.def <CR>
```

6. The ICU acknowledges and processes the command and outputs:

```
The Definition File has been restored to the file:
  ICUTEST.DEF
To see the RESTORE messages, inspect the log file:
  ICUTEST.LOG
```

Note

A message may appear that the definition file has been modified. Ignore this message.

The ICU queries:

```
Continue to the ICU Main Menu? [y]/n: y <CR>
```

The ICU command appears with the list of available ICU commands.

7. For this example, we will change the target directory of the generation files. First, view the main screen to list all changeable options in the definition file.

```
ENTER COMMANDS: c gen <CR>
```

8. The Generation (GEN) screen appears.
9. To change the target directory, type:

```
:raf=/msa32/boot/icutest <CR>
```

10. Press <CR> twice. The GEN screen re-appears with the modified settings.
11. Return to commands screen by quitting the GEN screen.

```
:q <CR>
```

12. Save the file before generating the new files.

```
:s <CR>
```

13. Generate the new definition file at the commands screen.

    ```
    ENTER COMMANDS:  g <CR>
    ```

14. You are queried for a prefix.

    ```
    Enter a letter to be used as prefix:  r <CR>
    ```

15. The ICU generates the files used by the definition file. When the ICU finishes, the `ENTER COMMAND:` prompt appears. Now exit from the ICU.

    ```
    ENTER COMMAND:  e <CR>
    ```

16. On exiting, the ICU creates the definition file, *icutest.def*. It also creates the submit file, *icutest.csd*. This file generates the target environment files. In this example, the target environment is a Multibus system using a SBC 486133SE board.

17. Run the submit file.

    ```
    - submit icutest over icutest.out echo <CR>
    ```

18. Use AEDIT to access *icutest.out* to check for any generation errors. If there are no errors, then copy the target environment file to the target system. If there are errors, invoke the ICU using *icutest.def*.

    ```
    - icu386 icutest.def <CR>
    ```

    Correct the errors, save the changes and regenerate the target environment file.

19. To copy files to a target system, use either iRMX-Net or TCP/IP.

    A. Use iRMX-Net by:

    1) Attaching to the Multibus system:

    ```
    - ad remote_system as rem r <CR>
    ```

    2) Copying *icutest* to the Multibus system.

    ```
    - copy icutest to :rem:msa32/boot <CR>
    ```

    See also: iRMX-Net, *Network User's Guide and Reference*,
    FTP, *TCP/IP and NFS for the iRMX Operating System*

Note

The boot directory, *msa32/boot*, is for definition files on Multibus II systems. For Multibus I systems, substitute */boot32* for *msa32/boot*.

B. If both the development system and the target environment system have TCP/IP running, use FTP to upload the files.

20. Test the files on the new target system.
21. Test and re-generate the files if required.

❑❑❑

3 Designing an Application

This chapter presents concepts for designing and creating an iRMX application. This includes application code demonstrating the concepts. Details about the location and running of the example application code, *demo.c*, are located at the end of the chapter. This code is written in C using the iC-386 compiler. You should be familiar with C syntax and structures to understand the examples.

See also: *Introducing the iRMX Operating Systems*,
System Concepts,
iC-386 Compiler User's Guide,
C Library Reference
C Compiler-specific Information, Chapter 4

Application Categories

Most iRMX applications are written for one of three categories: measurement, process control, or data acquisition. There is no distinct differentiation between categories and an application can overlap one or more categories.

Measurement

A point of sale terminal for a gas station is an example of an iRMX application focusing on measurement. As the fuel tank on a car fills, the application tracks the quantity pumped by interacting with a flow meter. When the fuel tank is filled and flow stops, the flow meter signals the application to calculate the cost based on the amount of fuel pumped.

Process Control

An assembly line conveyor belt is an example of an iRMX application focusing on process control. Component parts are removed from the conveyor belt by human operators and placed in certain devices. Electronic eyes monitor the number of component parts passing at given points. If the human operators require more time to remove a part from the belt, an electronic eye recognizes that fewer parts are being removed from the belt. The electronic eye then triggers the application to slow the speed of the belt.

Data Acquisition

A telephone communications network is an example of an iRMX application focusing on data acquisition. The network is partitioned into specific sectors. The application monitors the amount of telephone traffic that occurs in each sector. Subsequent analysis identifies those sectors that have large amounts of telephone traffic. Routing schemes could then be developed to handle the large amount of traffic. Additionally, connection times could be recorded before and after to check the efficiency of the routing schemes.

Design Concepts

All iRMX applications, regardless of category, use some or all of these functions:

- Handling I/O
- Interprocess communication
- Intertask synchronization
- Creating and cataloging objects
- Controlling devices
- Allocating memory
- Processing exceptions
- Prioritizing tasks
- Computing
- Handling interrupts
- File sharing

C Multitasking Demo Program

The demonstration program, *demo.c*, presents programming concepts which use some or all of the functions listed above. Use this program as an aid in developing your own application code. This program is described later in greater detail.

Demo Code Location

The */rmx386/demo/c/intro* directory contains this source code and related files. It is easier to understand the examples if you produce hard copies of the source code or view them from a console screen using an ASCII text editor.

make	file to generate example
demo.c	main program code containing the initial task
task2.c	second task code
crbpool.c	buffer pool code
except.c	exception handler

Demo Example	Generation Environment	Command
iC-386 demo	iRMX	make
Watcom C demo	DOS	make -f makefile.w
Microsoft C demo	DOS	make -f makefile.w
Borland C demo	DOS	make -f makefile.w

The C versions of the demo are generated from the same *demo.c* source. All versions of the demo are functionally equivalent, and all run under the iRMX OS.

See also:C Compiler-specific Information, Chapter 4

Table 3-1 lists the functions and associated system calls used in *demo.c*.

Table 3-1. Demo.c Functions and System Calls

Procedure	Functions Demonstrated	System Calls Used
main()	IORS mailbox creation	rq_create_mailbox
	Getting terminal attributes	rq_a_special
	Receiving an IORS	rq_receive_message
	Deleting an IORS	rq_delete_segment
	Setting terminal attributes	rq_s_special
	Getting iRMX version	dq_get_system_id
	Building the job's object directory	rq_create_mailbox rq_catalog_object rq_create_semaphore rq_catalog_object rq_create_buffer_pool rq_catalog_object rq_get_priority rq_create_task rq_catalog_object
	Getting buffer pool memory	rq_request_buffer
	Using semaphores	rq_send_message rq_receive_units
	Displaying data to the console	rq_s_write_move
write_read	Console I/O	rq_a_write rq_wait_io rq_wait_iors rq_a_read
prompt_and_wait	Console I/O	rq_a_write rq_wait_io rq_wait_iors
	Job termination from console	rq_exit_io_job

Running the Multitasking Demo

Note

Before running any C examples, load the *clib.job* or configure it into the OS with the ICU. You can manually load it by entering this command at the HI prompt:

```
- sysload /rmx386/jobs/clib.job
```

See also: *clib.job, System Configuration and Administration*

The *makefile* file first compiles and binds the source files using iC-386 and BND386 and then creates an executable program named *demo*. Enter these commands to first attach to the directory where the demo files reside and then use the **make** command to run the *makefile*:

```
- af /rmx386/demo/c/intro <CR>
- make <CR>
```

To execute *demo*, enter:

```
- demo <CR>
```

After typing the filename, the program prompts you with this message:

```
iRMX III C Multitasking Demo, VX.Y

Welcome to the C Multitasking Demo!

At the prompt you will be given 60 seconds to hit any key.
If you do not hit a key the demo will continue anyway.
You may hit an "E" if you wish to exit the program.

You now have <xx> seconds left to hit a key.
```

After you press a key, the program clears the screen and prompts you with this message:

```
Please hit a key which will be forwarded to task2 for processing.
```

Assuming you enter the letter X for the first keystroke, the main program, containing the initial task, reads the X from the terminal and passes it on to Task2. Task2 wakes up and prints out this message to the screen:

```
TASK2 PROCESSING X
Please hit a key which will be forwarded to task2 for
processing
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX ....
```

The X continues to appear at the rate of one per second and will repeat indefinitely until you enter another keystroke. Also, notice that the prompt to enter another keystroke is buried in the middle of Task2's processing message and the string of letters that it displays.

Entering the next two keystrokes concludes the program. This output assumes you enter the characters Y and Z:

```
TASK2 PROCESSING Y <CR>
Please hit a key which will be forwarded to task2 for
processing
YYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYY ....
TASK2 PROCESSING Z <CR>

This concludes the C Demo Program.

This would be a good time to examine the program code
to see how these features work.

We will now exit by generating an error.

INTERNAL ERROR IN MODULE demo.c at line #450
STATUS = 0023: E_SUPPORT
```

After you enter the final keystroke, the initial task recognizes that you have entered three characters, signaling the code to end the program. The initial task ends the program before Task2 begins to repeatedly print the third character to the console screen.

Using the Makefile

Each of the demonstration programs has its own unique *makefile* for compiling and binding the programs.

A listing of the *makefile* for generating *demo* follows:

```
#  *-*-*  makefile     *-*-*
#
#  This makefile generates the iC-386 multitasking demo
#  for iRMX III.
#
#       Invocation:     make
#
        ##
        ##  Compile and Bind switches
        ##
```

```
DEBUG           =       nodb
TYPE            =       noty

        ##
        ##      Tool and library definitions
        ##

LANG            =       :lang:
CLIBDIR         =       :sd:intel/lib
RMXLIBDIR       =       :sd:rmx386/lib

RUN86           =       :utils:run86


        ##
        ##      Binder definitions
        ##

BND3            =       :sd:intel/bin/bnd386.exe
BND             =       $(RUN86) $(BND3)
BNDFLAGS        =       $(DEBUG) $(TYPE) rn(code to code32)

        ##
        ##      Compiler definitions
        ##

CC3             =       $(LANG)ic386
CC              =       $(RUN86) $(CC3)
CFLAGS          =       cp dn(0) extend ot(3) si(:include:) nosrclines $(DEBUG)

        ##
        ##      Libraries
        ##

CLIB            =       $(CLIBDIR)/cifc32.lib
CSTART          =       $(CLIBDIR)/cstart32.obj
RMXLIB          =       $(RMXLIBDIR)/rmxifc32.lib
UDILIB          =       $(RMXLIBDIR)/udiifc32.lib

        ##
        ##      Implicit rules
        ##

.SUFFIXES: .obj .c

.c.obj:
        $(CC) $*.c oj($@) pr($*.lst) $(CFLAGS)
```

```
        ##
        ##      Targets and explicit rules
        ##

default: demo

demo: crbpool.obj demo.obj except.obj task2.obj $(BND3) $(CSTART) $(CLIB) \
              $(RMXLIB) $(UDILIB) makefile
        $(BND)                          &
        $(CSTART),                      &  C startup module
        demo.obj,                       &  C Demo modules
        task2.obj,                      &
        except.obj,                     &
        crbpool.obj,                    &
        $(CLIB),                        &  iRMX III Shared C Interface library
        $(UDILIB),                      &  iRMX III UDI Interface library
        $(RMXLIB)                       &  iRMX III System Call Interface library
        $(DEBUG) oj($@) pr($@.mp1)      &
        rn(code32 to code) ss(stack(2400H)) rc(dm(4000h,0FFFFFh))

        ##
        ##      Dependency information
        ##

CMNCINCS    =  :include:i186.h      :include:i286.h       :include:i386.h \
               :include:i86.h       :include:locale.h     :include:reent.h \
               :include:rmxc.h      :include:rmxtypes.h   :include:time.h \
               :include:rmx_err.h   :include:stdio.h      :include:stdlib.h \
               :include:rmx_c.h     :include:yvals.h      :include:_align.h \
               :include:_noalign.h  :include:_restore.h   :include:udi.h \
               :include:udi_c.h     demo.h                makefile

crbpool.obj :      $(CMNCINCS) crbpool.c
demo.obj    :      $(CMNCINCS) demo.c :include:ctype.h :include:rmx_def.h \
                   :include:string.h
except.obj  :      $(CMNCINCS) except.c :include:rmx_def.h
task2.obj   :      $(CMNCINCS) task2.c :include:rmx_def.h :include:string.h
```

Programming Concepts

The specific programming concepts conveyed in *demo.c* are:

- Creating objects using iRMX system calls
- Cataloging objects so tasks can share them
- Processing an Input/Output Result Segment (IORS) data structure to check the status of an I/O operation
- Using response pointers during inter-task communication
- Simultaneous task processing and data sharing
- Using buffer pools to create memory resources for a job
- Processing in-line exceptions resulting from iRMX system calls in application code
- Getting and setting terminal attributes
- Performing screen input/output to read and write data using the physical terminal screen
- Performing simultaneous input/output so tasks perform I/O operations independent of one another
- Accessing the IORS
- Processing interrupt tasks

Creating and Cataloging Objects

Every iRMX object has attributes. These attributes enable you to customize the object's use in an application. You specify these attributes when you create an object.

Listed below are iRMX objects used in the *demo.c* program.

Objects in demo.c	Description of use
Task	An initial task does input and a subtask does output
Semaphore	One semaphore synchronizes the initial task and subtask
Mailbox	Two local mailboxes exchange input/output and a global mailbox transfers between the initial task and subtask
Buffer pool	One buffer pool passes messages and objects between mailboxes

Operations on Objects

The OS has an object-based architecture. There are three main advantages to working in an object-based OS. These advantages are: design consistency, type checking protection, and customization.

Design Consistency. The Nucleus provides objects and functionality found in most normal OSs.

See also: Objects, *Introducing the iRMX Operating Systems*,
Nucleus, *System Concepts*

Type Checking Protection. Because each object has a type attribute, the OS can check for incorrect parameters (object token) in a system call, thereby avoiding system call errors.

Customization. You can define additional object types and system calls (known as operating system extensions). Use these features to customize the OS. However, limit the OS extensions to one application for easy maintenance.

Creating Tasks

If you have tasks that need to share resources (such as data or code), consolidate those tasks in the same job. If you have tasks that perform dissimilar functions, separate those tasks into different jobs. This maximizes modularity and adds protection because of separate memory spaces.

For simple applications that involve only one programmer and that have no maintenance or expansion plans, it is simpler to put all the tasks in one job which lets the tasks:

- Share the same processor
- Use one ready queue
- Are removed from the task queue when waiting for a resource
- Share the same memory space
- Pass data by reference
- Communicate using mailboxes and semaphores

Tasks in different jobs on the same processor can:

- Pass data by reference through global segments
- Use one ready queue
- Have different memory spaces but all in the processor's memory space
- Share the same priority scheme

Dividing an application into jobs provides:

Functional partitioning	Each job is a group of tasks that perform similar functions. This enables easier management and understanding of large projects. Programmers only need to understand how their code interfaces with the code produced by other members of the project team. As long as the interfaces between code modules are controlled, the project itself can respond to significant design changes without adverse schedule impact.
Memory separation	Each job has its own memory pool. This provides protection from segmentation overflow. Tasks from different jobs have a minimal impact on an application if one becomes a runaway task.
Privilege	To isolate an environment in which privileged operations occur, group those tasks with high priorities into one job.

See also: Jobs, *System Concepts*

Task Creation Code Example

In this example, only two tasks exist: the initial task (in the file *demo.c*) and the subtask Task2 (in the file *task2.c*). Regardless of the number of tasks in your particular job, the principles for task creation remain the same.

The following code, from *demo.c*, shows how the initial task creates, assigns a priority to, and catalogs Task2.

```
priority = rq_get_priority(CALLER,&status);
```
Get the priority of the calling task, which is the initial task.

```
task = rq_create_task (-- priority,
```
Create the subtask and give it a lower priority than the initial task.

```
  &task2,
```
Set the start address by pointing to the first instruction of Task2.

```
  _get_ds(),
```
Set the data segment parameter to create its own data segment.

```
  (UINT_16 far *) NULL,
```
Set the stack pointer for automatic stack allocation.

```
  (NATIVE_WORD) 0x2400,
```
Set the stack size to 2400H bytes. Set stack sizes to at least 300H bytes for Nucleus system calls and 700H bytes for C library calls.

```
  (UINT_16) 0,
```
Set task flags to zero, indicating no floating point instructions.

```
  &status);
```
A pointer to where the condition code returns.

```
error_check (__LINE__, __FILE__, status);
```
Each time a system call is made, a subsequent call is made to error_check, which checks the error of status of the previous system call.

```
udistr((char *) &rmx_str,"TASK2");
```
Call iRMX procedure udistr to convert Task2 from a null-terminated C string to a counted iRMX string.

```
rq_catalog_object(CALLER, task, &rmx_str, &status);
```
Catalog the subtask, Task2, in the object directory of the initial task (in demo.c).

Creating and Cataloging Objects Code Example

The following code, from *demo.c*, catalogs and creates a mailbox, a semaphore and a buffer pool.

See also: *System Concepts*, for information about creating these objects.

Note

If debugging with Soft-Scope debugger or the iRMX System Debugger (SDB), catalog objects so TOKEN values correlate with their respective names in the program. Although the $ character is valid in a variable name, it should be omitted from variable names used as input to the debugger.

```
mail_box = rq_create_mailbox (FIFO_QUEUING, &status);
```
Create a mailbox.

```
error_check (__LINE__, __FILE__, status);
udistr((char *) &rmx_str, (const char *) "MBX");
```
Convert MBX from a C string to an iRMX string.

```
rq_catalog_object (CALLER,
```
CALLER is null so the object is cataloged in the initial task's object directory.

```
  mail_box,
```
Catalog the mailbox object.

```
  &rmx_str,
```
Give the object the catalog name of MBX.

```
  &status);
```
A pointer to where the condition code returns.

```
error_check (__LINE__, __FILE__, status);
semaphore = rq_create_semaphore ((UINT_16) 0,
```
Create a semaphore and set the initial number of units to zero.

```
  (UINT_16) 3,
```
Set the maximum number of units to three.

```
  FIFO_QUEUING, &status);
```
Use zero to indicate a FIFO queuing scheme.

```
error_check (__LINE__, __FILE__, status);
udistr((char *) &rmx_str,"SEMAPHORE");
```
Convert the C string to an iRMX string.

```
rq_catalog_object (CALLER, semaphore, &rmx_str, &status);
```
Catalog the semaphore in the initial task's job directory.

```
error_check(__LINE__, __FILE__,status);
pool_tkn = create_buf_pool((UINT_16) 18, (UINT_16) 18, (UINT_16) 0,
  (NATIVE_WORD) POOL_SEG_SIZE, &status);
```
Create a buffer pool through procedure create_buf_pool in external file crbpool.c.

```
error_check (__LINE__, __FILE__, status);
udistr((char *) &rmx_str,"BUFFER");
```
Convert the C string to an iRMX string.

```
rq_catalog_object (CALLER, pool_tkn, &rmx_str, &status);
```
Catalog the buffer pool in the initial task's job directory.

```
error_check(__LINE__, __FILE__,status);
```

Processing Input/Output Result Segments (IORS)

IORS data structures are processed to check the status of an I/O operation. The I/O system creates an IORS when a task requests an I/O operation, such as through the **a_special** system call. The resulting IORS contains information about the request and the device on which the I/O was performed.

The resulting IORS contains information such as error conditions, the type of operation, the device, and pointers to where the data is stored. The status is checked by accessing specific fields in the IORS data structure. For example, fields, such as `status` and `unit_status`, would contain status (including error) codes after the I/O operation.

An IORS is also an integral part of writing a device driver. Since a device driver interacts between the I/O system and the related device, an IORS provides information about the operation performed on the device as well as about the device itself.

See also: DUIB and IORS: Device driver Interfaces, *Driver Programming Concepts* and *System Call Reference*

Processing an IORS Code Example

The initial task, in *demo.c*, performs the I/O operation of getting the attribute of the input device. The iRMX OS creates the IORS and then checks it to verify that the attributes were successfully obtained.

```
rq_a_special (input_conn_t, SPECIAL_GET_TERM_DATA, (void far *)
  &term_atts, read_mbx, &status);
```

This I/O operation gets terminal attributes of the input device. The IORS will be placed in the read_mbx mailbox when it arrives.

```
error_check(__LINE__, __FILE__,status);
```

The initial task then waits until the IORS arrives. This code illustrates how it waits:

```
#ifdef _FLAT_
```

If flat model is used, you must use the following call to access the IORS.

```
rq_wait_iors (input_conn_t,
```

Return the IORS specified in the previous connection.

```
   read_mbx,
```

Set iors_token to receive the terminal attributes.

```
   INFINITE_WAIT,
```

Wait infinitely for the terminal attributes to arrive.

```
   &iors, &status);
```

Point to a buffer where the IORS is placed.

```
    error_check (__LINE__, __FILE__, status);
    error_check (__LINE__, __FILE__, iors.status);
#else
iors_tkn = rq_receive_message (read_mbx,
```

Set iors_token to receive the terminal attributes.

```
  INFINITE_WAIT,
```

Wait infinitely for the terminal attributes to arrive.

```
  (SELECTOR far *) 0, &status);
```

Specify the mailbox which receives a status response.

```
error_check(__LINE__, __FILE__,status);
iors = (A_IORS_STRUCTURE *) buildptr(iors_tkn, (void near *) 0);
```

Build a pointer to the IORS.

```
error_check (__LINE__, __FILE__, a_iors->status);
```
Check the status of the IORS.

```
rq_delete_segment (iors_tkn, &status);
```
Manually delete the IORS because a_special does not recycle it.

Using a Response Pointer During Inter-task Communication

Tasks usually need to communicate with one another. Examples of this are:

- A serving task informing a requesting task that a process is done
- One task informing another that it has received some information
- A requesting task passing information to several serving tasks
- One task passing data to another
- Two or more tasks synchronizing their processing

Mailboxes in *demo* output messages, get user input, and transfer data. A semaphore synchronizes tasks.

The application uses two local mailboxes to pass messages and capture data. Messages to the terminal (output) are also sent to a mailbox. A task checks the mailbox for a message and sends the message to the terminal. User response to the message is captured in a data buffer and placed in another mailbox (in the same task) and returned to the main program.

A mailbox is also used to pass a data buffer among tasks. The initial task places a data buffer in the mailbox and catalogs the mailbox in its object directory. *Demo* imposes a restriction by explicitly cataloging the subtask. The restriction means only *demo* and its subtask (Task2) can access the mailbox. Depending on your application, cataloging a subtask is optional. The subtask accesses the mailbox and processes the data buffer it contains.

The semaphore synchronizes activities between the initial task and the subtask. The initial task creates a semaphore which tracks units sent to it by the subtask. The semaphore is assigned a maximum number of units which serves as a trigger. As the subtask processes each data input from the initial task, it sends one unit to the semaphore. The semaphore accumulates these units. The initial task stops and checks the semaphore to see if it contains its maximum number of units. If it does, the initial task knows that the subtask has completed all of its processing.

Task Synchronization/Data Passing Code Example

The initial task synchronizes its processing with Task2, the subtask. The initial task waits for, receives, and processes keystrokes at the same time that Task2 is writing the previous keystroke to the terminal and waiting for the next one. This synchronization enables input from and output to the terminal to be in separate tasks.

After the initial task obtains user input of a keystroke, it passes the data to Task2 through a mailbox. Task2 prints the keystroke to the screen and acknowledges the input by incrementing the count in the semaphore. It continues printing while waiting for another input from the initial task.

These are the functions and associated system calls used in the file *task2.c* (Task2).

Task Name	Functions Demonstrated	System Calls Used
task2	Getting object directory elements Character and Semaphore I/O	rq_lookup_object rq_receive_message rq_create_mailbox rq_a_write rq_wait_io rq_release_buffer rq_send_units

This code from the initial task, in *demo.c*, shows data passing between tasks and the synchronization of tasks among each other.

```
for (i = 1; i <= 3; i++)
{
```

Start a loop which will execute three times.

```
:
(code)
:
rq_send_message (mail_box,
```

Send a message to the mailbox signaling Task2 to execute.

```
  buff_tkn,
```

Send the data buffer, containing the user keystroke, in the mailbox.

```
  semaphore, &status);
```

Identify the semaphore as the object notified by Task2 when it finishes a process.

```
        :
        (code)
        :
        rq_receive_units (semaphore,
```
Monitor the semaphore to see if it has received three units from Task2.

```
          (UINT_16) 3,
```
Set the trigger number to three units.

```
          INFINITE_WAIT, &status);
```
The semaphore waits infinitely to get three units. Task2 sends one unit to the local variable semaphore, which points to and increments semaphore in the initial task. After the initial task sends the third and final keystroke to Task2, the initial task examines the number of units in the object semaphore and, since it matches the trigger number of three, continues processing.

```
}
```
End the loop.

The initial task and Task2 communicate and synchronize through mailboxes and semaphores. This code listing is for Task2, located in the file *task2.c*:

```
dummy = udistr ((char *) &rmx_str,"MBX");
mail_box = rq_lookup_object (CALLER, &rmx_str, INFINITE_WAIT,
  &status);
```
Look up MBX as the mailbox defined in the object directory of the initial task.

```
error_check (__LINE__, __FILE__,status);
dummy = udistr ((char *) &rmx_str,"BUFFER");
pool_tkn = rq_lookup_object (CALLER, &rmx_str,INFINITE_WAIT,
  &status);
```
Use the buffer defined in the object directory of the initial task.

```
error_check (__LINE__, __FILE__, status);
buff_tkn = NULL_TOKEN;
```
Set this buffer so Task2 does not release it back to the buffer pool.

```
buff2_tkn = rq_receive_message (mail_box, INFINITE_WAIT,
  (SELECTOR far *) &semaphore, &status);
```
Retrieve the buffer containing the keystroke from the mailbox. If the mailbox is empty then wait until it is filled.

```
write_mbx = rq_create_mailbox (FIFO_QUEUING, &status);
```
Create a local mailbox to output messages to the terminal.

```
while (TRUE)
```
Start an infinite loop.

```
{
       if (status == E_OK)
       {
       rq_a_write (output_conn_t, (UINT_8 far *) message,
         (NATIVE_WORD) strlen (message), write_mbx, &status);
```
Output the message that Task2 is processing.

```
       error_check (__LINE__, __FILE__,status);
#ifdef_FLAT_
```
If a flat model is used, use the following call.

```
       rq_wait_iors (output_conn_t, write_mbx, INFINITE_WAIT, &iors,
         &status);
```
Waits for an IORS and copies it to a user-provided buffer.

```
#else
       actual = rq_wait_io (output_conn_t, write_mbx, INFINITE_WAIT,
         &status);
```
Returns the concurrent condition code for the prior call to the calling task.

```
#endif
       error_check (__LINE__, __FILE__,status);
       if (buff_tkn != (selector) NULL)
              {rq_release_buffer (pool_tkn, buff_tkn, (UINT_16) 0,
                &status);
```
Release the buffer back to the buffer pool. However, skip this the first time through since the buffer has not been retrieved from the buff_tkn variable.

```
              error_check (__LINE__, __FILE__,status);
              }
       buff_tkn = buff2_tkn;
```
Transfer the buffer from buff2_tkn to buff_tkn. This enables buff2_tkn to monitor the mailbox and accept a new buffer (keystroke) when it arrives.

```
        rq_send_units (semaphore, (UINT_16) 1, &status);
                                Every time Task2 receives a keystroke from the initial
                                task, Task2 sends a unit to the object semaphore.
                                Task2 knows where to send the unit because the initial
                                task passed the token for semaphore to the mailbox.
                                This token for semaphore is kept in Task2's version of
                                the variable semaphore (semaphore is a local variable).

        error_check (__LINE__, __FILE__,status);
        }
:
(code)
:
        rq_a_write (output_conn_t, (UINT_8 far *) dummy,
          (NATIVE_WORD) 1, write_mbx, &status);
                                Output the buffer (keystroke) to the terminal.

        error_check (__LINE__, __FILE__, status);
#ifdef _FLAT_
        rq_wait_iors (output_conn_t, write_mbx, INFINITE_WAIT,
          &status);
#else
        actual = rq_wait_io (output_conn_t, write_mbx, INFINITE_WAIT,
          &status);
#endif
        error_check (__LINE__, __FILE__, status);
        buff2_tkn = rq_receive_message (mail_box, (UINT_16) 100,
          (selector far *) &semaphore, &status);
                                Check the mailbox to see if a buffer has been sent by
                                the initial task.  If a buffer does not arrive after one
                                second, return to the top of the loop and repeat
                                processing.

}
```

Using Buffer Pools

Buffer pools provide a shared resource of buffers, which are fixed-length segments of memory. Any tasks can use these segments, eliminating the need to repeatedly create or delete memory segments. Use this sequence when creating a buffer pool:

1. Create the buffer pool using the **create_buffer_pool** system call. One of the pool's attributes is having its memory segments defined as contiguous or daisy-chained. Select the contiguous attribute for applications where few data objects are passed or few object transfers are made. Select the daisy-chain attribute if the application transfers a large number of data objects or has a large number of transfers.

2. Once the buffer pool is created, initialize the pool by allocating a set of memory segments (buffers), for the pool. Use the **create_segment** system call to define segments. The size of the segment must accommodate the size of any objects being passed. For example, *demo* uses one byte buffers. This size accommodates the user-input keystroke captured in the buffer.

3. Release the buffer into the buffer pool using the **release_buffer** system call. This call initially populates the buffer pool, as well as recycles buffers when they are no longer needed. The most efficient way to create buffers and release them to the buffer pool is with a loop. Set the loop control variable to the initial number of buffers in the pool.

See also: Buffer ports, *System Concepts*

Note

Create and fill buffer pools at the beginning of your job since creating iRMX memory segments is a slow process relative to other system calls.

Creating Buffer Pools Code Example

The initial task in *demo.c* creates and catalogs a buffer pool. Once the buffer pool has been established, the calling task must request a buffer, assign data to it, and pass the buffer to the subtask (Task2). After receiving the buffer, the serving task must secure the data and release the buffer back to the buffer pool for possible use by other tasks.

The file *crbpool.c* contains a procedure, called by *demo.c*, that creates a buffer pool. This file also creates an initial number of memory segments, and releases them to the buffer pool. A token for the buffer pool is returned to the caller. These are the functions and associated system calls used in *crbpool.c*.

Procedure	Functions Demonstrated	System Calls Used
create_buf_pool	Buffer pool creation Buffer pool initialization	rq_create_buffer_pool rq_create_segment rq_release_buffer

The initial task in *demo.c* calls procedure `create_buf_pool` (defined in *crbpool.c*) as follows:

```
pool_tkn = create_buf_pool
```

This call passes parameters to an external procedure in crbpool.c, which creates the buffer pool and the buffers used in the pool.

```
((UINT_16) 18,
```
Create a maximum of 18 buffers.

```
  (UINT_16) 18,
```
Create a minimum of 18 buffers.

```
  (UINT_16) 0,
```
Set the flags attribute to zero to create contiguous buffers.

```
  (NATIVE_WORD) POOL_SEG_SIZE, &status);
```
Set the size of each buffer to one byte.

```
 error_check (__LINE__, __FILE__, status);

udistr((char *) &rmx_str,"BUFFER");
rq_catalog_object (CALLER, pool_tkn, &rmx_str, &status);
```
Catalog the buffer pool in the object directory of the initial task.

The following is code from procedure `create_buf_pool` in *crbpool.c*:

```
SELECTOR create_buf_pool (
```
Receive the attributes sent from the initial task.

```
UINT_16   max_bufs,
```
Parameter declaring the maximum number of buffers in the buffer pool.

```
UINT_16   init_num_bufs,
```
Parameter declaring the initial number of buffers in the buffer pool.

```
UINT_16   attrs,
```
Parameter declaring attributes for the buffer pool as contiguous buffers.

```
NATIVE_WORD   size,
```
Parameter declaring the size of each buffer as one byte.

```
UINT_16   *status_ptr )
```
Exception pointer.

```
{
SELECTOR buf_pool;
```
Variable declaration for the buffer pool.

```
SELECTOR buf_tok;
```
Variable declaration for the buffer.

```
int   i;
```

Variable declaration for the loop control variable.

```
buf_pool = rq_create_buffer_pool (max_bufs, attrs, status_ptr);
```

Create the buffer pool.

```
error_check (__LINE__, __FILE__,*status_ptr);
```

```
       for (i = 1; i <= init_num_bufs; i++)
```

Set the loop counter variable to the minimum number of buffers so the buffers are created when the loop finishes.

```
       {  buf_tok = rq_create_segment (size, status_ptr);
```

Create the buffer (memory segments).

```
       if (*status_ptr != E_OK)
          return (NULL_TOKEN);
```

Check if the segments are created correctly.

```
       rq_release_buffer (buf_pool, buf_tok, (UINT_16) 2,
          status_ptr);
```

Make the buffer part of the buffer pool.

```
       if (*status_ptr != E_OK)
          return (NULL_TOKEN);  }
return (buf_pool); }
```

Return the token for the complete buffer pool back to the initial task.

Using Buffer Pools Code Example

In order to use buffers from the buffer pool, the initial task and Task2 must request and release buffers. Recall that when the initial task was involved in its loop to send user-supplied keystrokes to Task2, the object being sent was a buffer. This code, from *demo.c*, shows how the main program requests a buffer from the buffer pool and waits for data to come to it.

```
for (i = 1; i <= 3; i++)
```

Set the loop to capture three keystrokes.

```
       {
       buff_tkn = rq_request_buffer(pool_tkn, (UINT_32) 1, &status);
```

Request a token for a free buffer from the buffer pool.

```
       error_check (__LINE__, __FILE__, status);
#ifdef _FLAT_
```

If the flat model is used, you must use a temporary buffer.

```
       *tmp_buff = write_read(message_2, INFINITE_WAIT, &status)
       actual = rq_move_data(_get_ss(), tmp_buff, buff_tkn,
         (void *) 0, (UINT_32) POOL_SEG_SIZE, &status);
       error_check (__LINE__, __FILE__, status);
#else
       buffer = (UINT_8) buildptr(buff_tkn, (void near *) 0);
```

Build a pointer to the buffer.

```
       *buffer = write_read (message_2, INFINITE_WAIT, &status);
```

The program waits indefinitely for the user to enter a keystroke. When a key is pressed, the character goes into a buffer, which is a pointer constructed from buff_tkn.

```
       error_check (__LINE__, __FILE__, status);
#endif
       rq_send_message (mail_box, buff_tkn, semaphore, &status);
```

A semaphore is passed as the exchange to which the response should be sent.

```
error_check (__LINE__, __FILE__, status);
       }
```

After Task2 receives the buffer in a mailbox, it processes it, and then releases the buffer to the pool for recycling. This code is from *task2.c*.

```
{
rq_a_write (output_conn_t, (UINT_8 far *) message, (NATIVE_WORD)
  strlen(message), write_mbx, &status);
```

Output a message to the terminal that Task2 is processing.

```
error_check (__LINE__, __FILE__,status);
#ifdef _FLAT_
```

If the flat model is used, use the following call.

```
       rq_wait_iors (output_conn_t, write_mbx, INFINITE_WAIT,
         &iors, &status);
#else
       actual = rq_wait_io (output_conn_t, write_mbx, INFINITE_WAIT,
         &status);
```

Retrieve the status of the a_write and delete the resulting IORS.

```
#endif
error_check (__LINE__, __FILE__,status);
```

```
if (buff_tkn != (selector) NULL)
       {
       rq_release_buffer (pool_tkn, buff_tkn, (UINT_16) 0, &status);
       error_check (__LINE__, __FILE__,status);
       }
```

The first time through the loop, the variable buff_tkn is NULL, or zero, so Task2 skips the code that releases the buffer back to the buffer pool. The second and third times through, Task2 releases the buffer before capturing the currently received keystroke. The parameter buff_tkn contains the token that indicates which buffer to release (the same buffer requested by the initial task for the previous loop pass).

```
buff_tkn = buff2_tkn;
```

After releasing the buffer, buff_tkn can be set equal to buff2_tkn, the token of the buffer containing newly arrived keystroke. The buff2_tkn token is now free to accept the next user keystroke when it arrives at the mailbox.

```
rq_send_units (semaphore, (UINT_16) 1, &status);
```

Task2 sends a unit to the semaphore. Task2 will send a total of three units to the semaphore.

```
error_check (__LINE__, __FILE__,status);
}
```

Methods of Screen Input/Output

Applications can write from a task buffer to a connected physical file. A connected physical file can be any I/O device. This example obtains physical file connections for the keyboard (input) and console screen (output). When dealing with I/O connections, tokens must be used. This example shows two methods that you can use to perform this type of I/O.

See also: **a_write** and **wait_io** system calls, *System Call Reference*

Screen Input/Output Code Example

A very simple type of I/O is used for clearing the screen. This code, from *demo.c*, shows the procedure:

```
void clear_screen
       (void)
       {int  i;
```

Declare the loop control variable.

```
       for (i = 1; i <= 25; i++)
       printf ("\n");
```

This loop clears the console by sending it 25 newlines.

```
       }
```

The second method of I/O first establishes the input and output devices in procedure main in *demo.c*:

```
input_conn_t = _get_rmx_conn (fileno (stdin));
```

Get the token for the read operation connection. The token received is for the standard input, i.e., the keyboard.

```
output_conn_t = _get_rmx_conn (fileno (stdout));
```

Get the token for the write connection. The token received is for standard output, i.e., the console.

In procedure `write_read` (*demo.c*), the program sends output to and waits for input from the I/O devices established above.

```
rq_a_write (output_conn_t,
```

Write a message to the console by sending it the console token.

```
  (UINT_8 far *) msg_3,
```

Sends the message addressed by msg_ptr to the screen.

```
  (NATIVE_WORD) strlen(msg_3),
```

Sends the number of bytes to be written, which is the size of the message addressed by msg_ptr.

```
  write_mbx, &status);
```

The mailbox that receives the IORS.

```
error_check (__LINE__, __FILE__, status);
#ifdef _FLAT_
       rq_wait_iors (output_conn_t, write_mbx, INFINITE_WAIT,
         &iors, &status);
#else
       actual = rq_wait_io (output_conn_t, write_mbx, INFINITE_WAIT,
         &status);
```

Returns the actual number of bytes written in the previous a_write call. The waiting period for wait_io to return data is set to infinite. This tells the procedure that no I/O will occur until data arrives. This call also recycles the IORS and deletes the IORS for all other BIOS calls. The user does not have to specifically delete the IORS.

In-line Exception Processing

Exceptions can be processed three ways: in-line, using the default exception handler, or by assigning your own exception handler. Each one has advantages and disadvantages. In-line handling is the simplest to create but you must also explicitly pass control to your exception handler. Use one of several default handlers to let the system handle the default. The appropriate default handler (selected in the ICU) should be used for your application. Create your own exception handler to have control over handling exceptions. Ensure that the exception is genuine, for example, that the handler does not read an interrupt as an exception.

Writing Your Own Exception Handler

You need to consider several things when you write your own exception handler. For example, 32-bit code requires 32-bit exception handlers, and 16-bit code requires 16-bit exception handlers. The only time this is not true is when the exception handler deletes the offending job, deletes the offending task, or suspends the offending task.

Another consideration is the type of exception you are processing. With this release of the iRMX OS, you can write exception handlers that process hardware traps. This means that your handler can process three groups of errors:

- Hardware traps
- Numeric Processor Extension (NPX) exceptions
- All other programming and environmental conditions

Finally, if you set the system's default exception handler in the ICU on the (NUC) Nucleus screen by setting DSH equal to "User", your exception handler module must have these characteristics:

- The public entry point must be named `rqsysex.`
- It must be 32-bit code.
- It must be compiled as Near using Intel OMF386 tools (iC-386, PL/M-386, or ASM386).

Exception Handler Control Flow

When writing a custom exception handler, follow these guidelines:

- Use the */rmx386/demo/c/intro/nstexh.h* file as a starting template for your exception handler.
- Code the exception handler initialization at the beginning of the application.
- Pass control to the custom exception handler rather than to the system default exception handler.
- Check for the type of exception and handle appropriately. Hardware exceptions can now be returned to your handler. Consequently, you need to check for these exceptions as well as programming and environment exceptions.

See also: **get_exception_handler**, **rqe_get_exception_handler**, **set_exception_handler**, and **rqe_set_exception_handler** system calls, *System Call Reference*

- You can delete the calling task that encounters the exception by using a NULL task token when invoking the **delete_task** system call. The system default exception handler does this automatically.
- Check if a task is interrupt-driven and if it is, use the **reset_interrupt** system call to delete it. If your exception handler deletes tasks using the **delete_task** system call, be sure that it does not attempt to delete an interrupt task. The **delete_task** system call cannot delete an interrupt task. Attempting to do so causes an exception, re-triggering the exception handler to try and delete the task again. This causes an infinite loop.

See also: **delete_task** and **reset_interrupt** system calls, *System Call Reference*

- Depending on your application requirements, your exception handler can have full or partial control.

See also: Exception Handling, *System Concepts*, Default Exception Handler screen, *ICU User's Guide and Quick Reference*

Exception Processing Code Example

Demo calls *except.c*, which contains two procedures that handle exceptional conditions. The first procedure gets the current exception handler and specifies the level of control. The second is an in-line exception handler.

These are the functions and associated system calls used in *except.c*.

Procedure	Functions Demonstrated	System Calls Used
set_exception	Get the exception handler Set the exception mode	rq_get_exception_handler rq_set_exception_handler
error_check	Format the errors that occur during system calls	rq_c_format_exception rq_exit_io_job

The initial task (in *demo.c*) and Task2 (in *task2.c*) call procedure `set_exception`, the exception handler.

See also: **get_exception_handler** and **set_exception_handler** system calls, *System Call Reference*, Managing Exceptional Conditions, *System Concepts*

```
set_exception((int) NO_EXCEPTIONS);
```

Set the exception mode to zero, which tells the OS never to pass control to default exception handler routines. (NO_EXCEPTIONS) is defined as zero in the header file rmx_def.h).

This code in procedure `set_exception`, from *except.c*, creates and invokes the exception handler.

```
rq_get_exception_handler ((EXCEPTIONSTRUCT far *) &except_info,
  &status);
```

Transfer exception handler information to the data structure addressed by except_info.

```
except_info.exception_mode = except_mode;
```

Replace the exception mode with the zero parameter passed from the initial task. This tells the system not to use the default exception handler.

```
rq_set_exception_handler ((EXCEPTIONSTRUCT far *) &except_info,
  &status);
```

Set the exception handler information with the altered data addressed by except_info (which is zero). This system call tells the system under what condition to pass control to the exception handler.

This code in procedure `error_check`, from *except.c*, formats the exception and tells you which error has occurred and where in the application it occurred.

```
rq_c_format_exception ((char *) &local_string, (UINT_16)
  _MAX_STRING, test_status, (BYTE) 1, &status);
```

Identify the type of error for the condition and place it in local_string.

```
local_string.text[local_string.length] = 0;
```

Terminate the string with a null (0) for output purposes.

```
printf ("\nInternal Error in module %s at line # %d\n", module,
  number);
```

Output where the error occurred.

```
printf ("Status = %s\n", &local_string.text);
```

Output what type of error occurred.

Getting and Setting Terminal Attributes

Before accessing the terminal for input or output, you must retrieve the current attributes and change them as necessary. Use the BIOS **a_special** system call and its `spec_func` parameter or use the EIOS **s_special** call and its `function` parameter.

See also: **a_special** and **s_special** system calls, *System Call Reference*

Getting/Setting Terminal Attributes Code Example

The initial task's code (in *demo.c*) uses both the **a_special** and **s_special** calls to access terminal attributes. The two calls use different I/O Result Segments (IORS). This code example in the initial task gets the current terminal attributes by calling **a_special**.

```
rq_a_special (input_conn_t,
```
Select the token on which to perform the function.

```
  SPECIAL_GET_TERM_DATA,
```
Specify the parameters to request the current terminal attributes.

```
  (void far *) &term_atts,
```
Specify the pointer to the array where the attribute data is placed.

```
  read_mbx, &status);
```
Specify the mailbox which receives the IORS.

The initial task then waits until the IORS arrives. This code (*demo.c*) illustrates how it waits:

```
#ifdef _FLAT_
rq_wait_iors(input_conn_t, read_mbx, INFINITE_WAIT, &iors, &status);
error_check (__LINE__, __FILE__, status);
error_check (__LINE__, __FILE__, iors.status);
#else
iors_tkn = rq_receive_message (read_mbx,
```
Set iors_token to receive the terminal attributes.

```
  INFINITE_WAIT,
```
Wait infinitely for the terminal attributes to arrive.

```
  (SELECTOR far *) 0, &status);
```
Specify the mailbox which receives the IORS token.

```
iors = (A_IORS_DATA_STRUCTURE *) buildptr(iors_tkn,
  (void near *) 0);
```

Build a pointer to and check the status of the IORS.

```
error_check (__LINE__, __FILE__, iors->status);
#endif

#ifndef _FLAT_
rq_delete_segment (iors_tkn, &status);
```

Manually delete the IORS because **a_special** does not recycle it.

```
error_check (__LINE__, __FILE__, status);
#endif

term_atts.connection_flags = ((term_atts.connection_flags
  & (~CMASK_LINE_EDIT))|1) | CMASK_ECHO;
```

Modify two terminal attributes to cause no line editing and no keystroke echoing to the screen. This long assignment statement alters the least-significant three bits of the 16-bit connection_flags element of the term_atts data structure. The literals C_MASK_LINE_EDIT and C_MASK_ECHO are equal to 3 and 4, respectively. (The NOT operator is defined in the header file not.h. The literals C_MASK_LINE_EDIT and C_MASK_ECHO are defined in the header file tscrn.h. These header files are in the same directory as demo.)

```
rq_s_special (input_conn_t, SPECIAL_SET_TERM_DATA, (void far *)
  &term_atts, (IORSSTRUCT far *) 0, &status);
```

Write the modified terminal attributes back to the physical terminal connection. When using the s_special call, you can avoid specifically deleting the IORS.

Interrupt Processing

> **Note**
> Interrupt processing involves knowledge of interrupts, interrupt controllers/lines, level of control, the Interrupt Descriptor Table (IDT), and interrupt tasks. These concepts are described in the Managing Interrupts chapter of the *System Concepts* manual.

Applications under the iRMX OS use interrupts to deal with external events. Processing these events asynchronously enables the OS to facilitate real-time processing.

These program examples cover interrupt handling, interrupt tasks, and interrupt latency. These examples use this hardware setup:

- PC Bus system running the iRMX OS
- Data Translation DT2806 Multi-Function I/O Expansion Board jumpered as follows:
  - I/O address 370H: In - W25, W29, W30, W31, and W32; Out - W26, W27, and W28
  - Timer 0 output to IRQ3: In - W24; Out - W2

> **Note**
> Since the application uses IRQ3, make sure no other card, such as a network card, uses this interrupt. Also, since IRQ3 disables COM2, ensure no other devices use COM2.

Interrupt Handlers

Use an interrupt handler to process interrupts when real-time speed and minimal processing are required. You can use an interrupt handler to call an interrupt task, which is slower to respond but enables more flexibility in processing. An interrupt handler executes into the context (stack, data segments) of the task that was interrupted. An interrupt task has its own context and runs with equal or lower priority interrupts disabled.

There are two example applications that demonstrate interrupt handling and interrupt tasks. The interrupt handling example is *inthand.c* and the interrupt task example is *inttask.c*. Both of these examples are located in the */rmx386/demo/c/int* directory.

The *inthand.c* example generates an interrupt and uses an interrupt handler to process the interrupt. The main program of the example sits idle while the interrupt handler processes the interrupt in the background. Every time an interrupt occurs, the interrupt handler increments a count. Finally, the main program prints the number of interrupts processed by the interrupt handler while it was sleeping.

The *inttask.c* example processes interrupts using an interrupt task. Every time an interrupt occurs, the interrupt task prints the message that it has processed that interrupt. The main program sits idle until the interrupt task is finished.

A single *makefile* compiles and binds these examples. To run the examples, attach to the directory, run *make*, and then run the executable.

- **af /rmx386/demo/c/int <CR>**
- **make <CR>**

To run *inthand.c*, type:

- **inthand <CR>**

To run *inttask.c*, type:

- **inttask *interrupts* <CR>**

where *interrupts* is the number of interrupts to process. The default value is 10 (minimum) and the maximum value is 100.

Interrupt Servicing

This section illustrates how interrupts are serviced. Tables 3-2, 3-3, and 3-4 outline a scenario where an interrupt handler is assigned to a level, an interrupt arrives at that level and is serviced, and the assignment of an interrupt handler is canceled. The tables show these cases:

- In Table 3-2, the interrupt handler deals with the interrupt (handler is assigned to master level 4).
- In Table 3-3, the interrupt handler invokes an interrupt task, either immediately or after filling a single buffer of data (handler is assigned to master level 4).
- In Table 3-4, an interrupt handler and an interrupt task use multiple buffers to service interrupts (handler is assigned to slave level 35).

The Interrupt Levels Necessarily Disabled column of each table indicates that the events of the example cause certain levels to be enabled or disabled. Other events outside the scope of the example might cause other levels to be disabled as well.

See also: Interrupts, *System Concepts*

Table 3-2. Servicing Interrupts with an Interrupt Handler

Step	Events	Explanation	Interrupt Levels Necessarily Disabled
1	--	No interrupt handler assigned to level M4.	M4
2	**rq_set_interrupt** (LEVEL_4,0,...);	A task assigns an interrupt handler to level M4.	None
3	Level 4 device interrupts	An interrupt arrives at level M4.	All
4	.	The interrupt is serviced by the interrupt handler.	All
5	**rq_exit_interrupt** (LEVEL_4,...);	Interrupt hardware reset by the interrupt handler.	All
6	Interrupt handler returns.	Interrupts are re-enabled.	None
7	**rq_reset_interrupt** (LEVEL_4,...);	A task cancels the assignment of an interrupt handler to level M4.	M4

Table 3-3. Servicing Interrupts with an Interrupt Task

Step	Events	Explanation	Interrupt Levels Necessarily Disabled
1	--	No interrupt handler assigned to level M4.	M4
2	**rq_set_interrupt** (LEVEL_4, 1, ...);	A task assigns an interrupt handler to level M4 and assigns itself to be the interrupt task for that level. It specifies that one **signal_interrupt** request can be outstanding.	M4-M7, 50-77
3	**rq_wait_interrupt** or **rqe_timed_-interrupt** (LEVEL_4,...);	The interrupt task begins to wait for an interrupt.	None
4	Level 4 device interrupts	An interrupt arrives at level M4. The interrupt handler gains control and optionally, does some servicing. The handler may service several interrupts by performing steps 4 through 6 of Table 3-2.	All
5	**rq_signal_interrupt** (LEVEL_4,...);t	The interrupt handler invokes the interrupt task.	M4-M7, 50-77
6	. .	The interrupt is serviced by the interrupt task.	M4-M7, 50-57
7	**rq_wait_interrupt** or **rqe_timed_-interrupt**. (LEVEL_4,...);	The interrupt task finishes and begins to wait for another level M4 interrupt. Control passes back to the interrupt handler and then back to an application task.	None
8	**rq_reset_interrupt** (LEVEL_4,...);	A task cancels the assignment of a handler to M4.	M4

Table 3-4. Servicing Interrupts with an Interrupt Handler, an Interrupt Task, and Multiple Buffering

Step	Events	Explanation	Interrupt Levels Necessarily Disabled
1	--	No interrupt handler assigned to level 35	35
2	**rq_set_interrupt** (LEVEL_35, 2, ...);	A task assigns an interrupt handler to level 35 and assigns itself to be the interrupt task for that level. It specifies two **signal_interrupt** requests can be outstanding (double buffering).	M4-M7 36-77
3	**rq_wait_interrupt** or **rqe_timed_interrupt** (LEVEL_35,...);	The interrupt task begins to wait for an interrupt.	None
4	Level 35 device interrupts	An interrupt arrives at level 35. The interrupt handler gains control and does some servicing.	All
5	. . .	The handler services all interrupts, as described in steps 4 through 6 of Table 3-2, until the first buffer is full.	All
6	**rq_signal_interrupt** (LEVEL_35,...);	The interrupt handler invokes the interrupt task.	M4-M7, 36-77
7	. . .	The interrupt task processes the full buffer. Meanwhile, the interrupt handler services interrupts, as described in steps 4 through 6 of Table 3-3, until the next buffer is full.	M4-M7, 36-77
8	**rq_wait_interrupt** or **rqe_timed_interrupt** (LEVEL_35,...);	The interrupt task finishes and waits for another signal from the interrupt handler. Control passes back to the interrupt handler and then back to an application task.	None
9	**rq_reset_interrupt** (LEVEL_35,....);	A task cancels the assignment of an interrupt handler to level 35.	35

Interrupt Latency

The *intlat.c* example, in the */rmx386/demo/c/intlat* directory, measures interrupt latency. Interrupt latency is the delay between when a device issues an interrupt request and when the microprocessor responds to the request.

See also: Interrupts, *System Concepts*

The *intlat.c* example uses this software setup:

- An Esubmit file, *measure.csd*, executes *intlat* a specified number of times, saving each of the executions' data in a unique data file. See the comment header of *measure.csd* for more information on this feature.
- A file *makefile*, which compiles and binds *intlat.c*.

See also: *readme.txt, measure.csd, /rmx386/demo/c/intlat* directory, *Driver Programming Concepts*

A single *makefile* compiles and binds the example. To run the example, first attach to the directory, and then run makefile to generate the proper files.

```
- af /rmx386/demo/c/intlat <CR>
- make <CR>
```

Now run *measure.csd*:

```
- esubmit measure(executions,timings_per_execution) <CR>
```

where *executions* is a number from 1 to 999 (3E7H), and *timings_per_execution* is a number from 1 to 8192 (2000H).

The results are placed in the log directory in the file named *intlat.xxx*.

Note

The *intlat* executable can be run alone but it requires certain parameters. To view the parameters, enter:

```
- intlat -HELP <CR>
```

❑❑❑

C Compiler-specific Information 4

This chapter provides information on:

- The iC-386 compiler
- Non-Intel tools you can use
  - The iRMX-supplied elements and how to use them
  - Debugging your object code
- Adding a first-level job created with non-Intel tools

Using the iC-386 Compiler to Develop iRMX Applications

Support files supplied with the iC-386 compiler facilitate iRMX application development. Using these files enables you to use iRMX system calls like C procedures calls.

Using the C Language Header Files

The iRMX directory structure includes Intel-supplied header files in the */intel/include* directory. These files have an extension of *.h*. Header files provide data structure definitions used by iRMX system calls and useful literal definitions used in iC-386 code. Use `#include` statements to include the header files.

These header files provided with the OS allow you to write programs with or without underscores in system call names, structure data types, and condition code mnemonics.

See also: Header Files, *System Call Reference*, for a list of header files to include in your programs.

Binding Your Code to Interface Libraries

After you have written your programs and inserted include statements for the necessary header files, compile the code and bind it to the appropriate iRMX interface library.

Interface libraries supplied with the OS provide a standard interface to the system calls. The interface libraries contain procedures that correspond to iRMX system calls. The interface procedure performs operations needed to invoke the actual system call, such as to call gates.

See also: Interface Libraries, *System Call Reference*,
Using the 80386 Binder, *Intel386 Family Utilities*,
Detailed bind sequence descriptions, *iC-386 Compiler User's Guide*

Note

When using header files or other external files, make sure you specify the correct path to the file, especially when using a *makefile*.

Condition and Error Codes

The header files *rmxerr.h* and *rmx_err.h* in the */intel/include* directory define iRMX condition codes that may occur during system operations. The condition codes are divided into three categories:

- Programmer errors
- Environmental conditions
- Hardware traps

A programmer error is a condition, such as a syntax error, that can be changed in the application code. An environmental condition is an OS problem over which you have no control. A hardware trap is when the microprocessor generates a hardware interrupt request based on the occurrence of certain internal microprocessor events.

The header files list the condition codes by OS layer and by ascending numeric values. Each entry includes the condition code mnemonic, the numeric value, and a brief description.

See also: Condition codes in individual call descriptions, Master list of condition codes, and Header files, *System Call Reference*

Using Non-Intel Tools to Develop iRMX Applications

> **Note**
> C++ is not supported. Many of these tools allow you to develop C or C++ applications. The iRMX OS supports only C applications developed with these tools. There is no iRMX support for C++ applications.

The iRMX OS environment allows you to develop C applications using Microsoft MSVC 32-bit versions to version 6.

For assembly code, you can use Microsoft MASM which produces 32-bit code accepted by the Microsoft linkers.

The iRMX OS provides these elements:

- A set of common C header files, compatible with all supported compilers.
- A custom cstart module for each supported compiler, in each supported memory model.
- An iRMX Shared C Library that provides an iRMX/C interface and is compatible with all supported compilers. It is compatible with existing iC-386 applications without recompiling or relinking.

Using Microsoft C /C++ Development Tools

Microsoft C/C++ tools are tailored to the Windows environment so you cannot use the default compiler switches, libraries, and header files. Override the defaults with options, libraries, and header files appropriate for the iRMX environment as listed here.

This section describes only the switches known to be necessary or to cause problems. Some switches not discussed here may be useful in your application, however, these have not been evaluated.

> **Note**
> The compiler and linker invocations in this section illustrate the use of required switches, but this is not how the example programs invoke these tools. Examine *makefile.m* in the *\rmx386\demo\c\intro* directory to see the invocation used in the examples.

Microsoft Visual C++ Compiler Invocation

iRMX applications require certain project settings in Microsoft Developer Studio. To view and verify settings required by iRMX software, select Microsoft Developer Studio's Project>Settings menu option. The table below lists the required settings; leave all other settings at their default values.

Note

The \iRMXIII\Project directory includes a flat model sample program that is compiled using the Microsoft Developer Studio (MSVC 6.0). Each subdirectory under the Projects directory is a separate workspace for MSVC

Table 4-1. Build Settings for Microsoft Developer Studio

Tab	Category	Field	Value
General		Microsoft Foundation Classes	Not using MFC
Debug	This tab requires no special settings.		
Custom Build	This tab requires no special settings.		
EC++	Common options	Display only field; shows values derived from other fields on this tab.	
	General	Debug info	C7 compatible
		Optimizations	Disable Maximize Speed
	Code Generation	Calling convention	__cdecl *
	Precompiled headers	Not using precompiled headers	Enabled (checked)
	Preprocessor	Preprocessor definitions	WIN32 _DEBUG _WIN32
		Ignore standard include paths	Enabled (checked)
Link	Project Options	Display only field; shows values derived from other fields on this tab.	
		Important: You must add value: /heap:0x100000,0x2000	
	General	Object/library modules	cstrtf3m.obj ciff3m.lib netiff3m.lib rmxiff3m.lib
		Generate debug info	Enabled (checked)
		Additions library path	c:\irmxIII\lib
		Ignore all default libraries	Enabled (checked)

	Customize	Output filename	Must have an .RTA extension
		Use program database	Disabled (not checked)
	Debug	Debug info	
		Debug info	Enabled (checked)
		Microsoft format	Enabled (checked)
	Input	Object/library modules	cstrtf3m.obj ciff3m.lib netiff3m.lib rmxiff3m.lib
		Additions library path	c:\irmxIII\lib
		Ignore all default libraries	Enabled (checked)
	Output		
	Stack allocations	Reserve	0x4000
		Commit	0x2000
	Version information	Major	21076
		Minor	20052
Resources	This tab requires no special settings.		
Browse Info	This tab requires no special settings.		

Using Header Files

The iRMX OS provides a set of common C header (`#include`) files that work with all supported compilers. The header files support all compiler-specific C data types and compiler-specific aliases. One file, *yvals.h*, contains all compiler-specific declarations, macros, and built-ins. It determines which compiler you are using and automatically makes the necessary adjustments.

These are a few of the header files designed to use with non-Intel development tools, with definitions and suggestions:

<*_align.h*>	Starts 2-byte/4-byte alignment (16-bit/32-bit compilers). This header file (with <*_noalign.h*>) is required to support multiple compilers.
<*_noalign.h*>	Ends multiple-byte alignment (see *_align.h* above); provides compiler-independent byte alignment. You can include this header file before structures to be affected, and then change back to *_align.h*.
<*_restore.h*>	Returns structure alignment to the compiler default (as specified on the command line).
<*rmxtypes.h*>	Defines iRMX kernel data types (UINT_8, etc.) to make them available to C programmers.

<yvals.h>	Contains standard C values, macros, built-in functions, and support definitions for all supported compilers.

See also: Header files, *C Library Reference*, for C functions,
iRMX header files, *System Call Reference*, for iRMX OS definitions

Existing iC-386 Applications

You must use iC-386 version V4.7 or later with the common header files, because the headers use global align/noalign pragmas instead of individual alignment pragmas for each structure. The global pragmas do not work correctly with earlier versions of iC-386, and unexpected results may occur. The individual alignment pragmas for each structure declaration have been removed from the header files since they are non-standard.

See also: Structure Data Alignment, in this chapter

Built-in functions

The *yvals.h* header file provides compiler-independent versions of the common built-in functions. ANSI C built-in functions are provided for new code, and the iC-386 built-in function names are provided for all compilers to simplify porting an existing iC-386 application to other compilers.

Listed below are the generic built-in functions provided for all compilers. An application that uses these built-in functions instead of the compiler-specific built-ins will remain portable across all supported compilers. Refer to the i*C-386 Compiler User's Guide* for more information on the use of these functions.

Function Name	Action
buildptr	Construct a pointer from a selector and offset
causeinterrupt	Generate a software interrupt
inbyte	Input a byte from an I/O port
inword	Input a word from an I/O port
outbyte	Output a byte to an I/O port
outword	Output a word to an I/O port
byte_rol	Rotate a byte left
byte_ror	Rotate a byte right
hword_rol	Rotate a 16-bit word left
hword_ror	Rotate a 16-bit word right
blockinbyte	Input a sequence of bytes from an I/O port
blockinword	Input a sequence of 16-bit words from an I/O port
blockoutbyte	Output a sequence of bytes to an I/O port
blockoutword	Output a sequence of 16-bit words to an I/O port

selector	16-bit selector data type
disable	Disable interrupts
enable	Enable interrupts

Calling Conventions

The iRMX system calls and Shared C Library functions require different calling conventions. These conventions are supported by each compiler in different ways. To achieve uniform function declarations, all functions and system call prototypes are declared in the header files with one of the following modifier macros:

_Cdecl	Declares the VPL (Variable Parameter List) calling convention, used by some Shared C Library functions.
_Pascal	Declares functions that use the FPL (Fixed Parameter List) calling convention, including most Shared C Library functions. It also indicates that the function preserves the (E)DI and (E)SI registers. The compiler does not need to save these registers.
_Fparam	Used for FPL functions that do not preserve (E)DI and (E)SI. This includes all iRMX system calls. The compiler will produce code surrounding the call to save and restore these registers, if necessary.

These macros are resolved in *yvals.h*, where they are mapped into the correct keyword for each supported compiler. Not all compilers support all of the calling conventions. For example, the Intel iC-386 compiler does not fully support the _Pascal convention (it does not preserve EDI/ESI). To resolve this, _Pascal is mapped to _Fparam in the iC-386 section of *yvals.h*.

Note

The Microsoft 32-bit compiler does not support the _Pascal calling convention so _Pascal is mapped to _Cdecl for flat model applications.

Structure Data Alignment

There are two types of data alignment required in the header files:

- The iRMX Shared C Library accepts and returns structures that are 32-bit aligned. This means that members of the structure are arranged so that they do not cross a 32-bit boundary. The compiler adds bytes of 0 between elements as necessary. The structures are aligned the same for both 16- and 32-bit applications.

- The iRMX system calls accept and return structures that are byte-aligned (also known as non-aligned).

To support both types of alignment on all supported compilers, the header files change the setting of the compiler's global alignment switch during compilation. Your application should therefore make no assumptions about structure alignment. Instead, the application should include one of these header files before structure declarations that require alignment or non-alignment:

<*_align.h*>	Enables structure alignment
<*_noalign.h*>	Disables structure alignment
<*_restore.h*>	Restores compiler default alignment (as specified on the command line)

Note

Do not use the `#pragma noalign` declaration in any application that includes the new common header files, including iC-386 applications.

Alignment with iC-386

The iC-386 compiler does not provide a way to return to the default alignment, nor does it provide a way to determine the default alignment at compile time. This is not consistent with the common header files, which no longer use individual #pragmas around every structure. To avoid this problem, set this macro on the command line for compiler invocation:

```
__NOALIGN__
```

The <*_restore.h*> header file examines this macro when attempting to restore the default alignment for iC-386. If `__NOALIGN__` is defined, <*_restore.h*> sets the alignment to `noalign`. If the macro is not defined, <*_restore.h*> sets the alignment to `align` since this is the iC-386 default.

To use the macro, define it in conjunction with the iC-386 `NOALIGN` pragma and/or command line switches. For example:

```
ic386 hello.c noalign define(__NOALIGN__)    /* command line example */
#pragma noalign                              /* program example */
#define __NOALIGN__
```

Supported Memory Models

The iRMX OS and the C header files support these memory models:

- 16-bit large model
- 16-bit compact model
- 32-bit compact model
- 32-bit flat model

If you attempt to compile a program in any other memory model, the header files return an error message. This prevents you from using an incorrect model that would not run correctly but would compile and link without errors. The error message is:

```
#error: Invalid memory model
```

This feature is not available on iC-386, since the compiler does not always set the flags that determine the memory model (for example, subsystems do not cause the compiler to set any of memory model flags).

Using Cstart Startup Code

The provided cstart modules initialize processes and call main( ). Link to the proper cstart module for your compiler and memory model. The files are in the *\intel\lib* directory.

Cstart Module	**Compiler**
cstartli.obj	Intel 16-bit large
cstartci.obj	Intel 16-bit compact
cstart32.obj	Intel 32-bit compact
cstrtf3m.obj	Microsoft 32-bit flat

Cstart provides the starting address for the program. The generic cstart algorithm is:

1. Set up stack and DS register.
2. Initialize any compiler-specific data.
3. Call any compiler-specific initialization routines.
4. Call get_arguments to obtain the command line arguments.
5. Call main().
6. Call any compiler-specific cleanup routines.
7. Call exit(0).

Note

Upon returning from main( ), the program calls exit( ) with a status of zero (E_OK). Status from main( ) is ignored. Since most programs do not return a value from main( ), it is left undefined. Calling exit( ) with an E_OK status also prevents random error messages from appearing on the terminal at program termination.

Stack Size

The default stack size provided in the cstart modules is 4 Kbytes. You can override this size in the link step.

Stack usage for a 16-bit application is actually greater than for an equivalent 32-bit application, because the OS converts the 16-bit parameters to 32-bits by expanding them and pushing an entire copy of the parameter frame on the stack before entry to an OS primitive.

Using Interface Libraries

There are a variety of interface libraries supplied with the OS for the interface to C library functions and iRMX system calls. For different Intel and non-Intel tools you must bind (link) to different libraries.

See also: Interface Libraries, *System Call Reference*, for a complete list of library files

Debugging with the Soft-Scope Debugger

The Soft-Scope debugger is provided with the iRMX OS. You must convert your final object module to OMF-386 format before you can debug it with Soft-Scope. Use the standard Soft-Scope procedures for debugging. If you are using Microsoft C or Watcom C compilers, you can also do remote debugging with Soft-Scope for Windows. The debugging tools supplied with non-Intel compilers are not suitable for on-target iRMX application debugging.

See also: *Soft-Scope Debugger User Guide*

Summary of Debug Switches

Use the command-line switches shown below to produce debug symbols for the Soft-Scope debugger. To eliminate debug symbols from your final code, do not use these switches when compiling, linking, and invoking STL.

Tool		Debug Switch
Microsoft C 32-bit	Compiler	/Z7 /Od
	Linker	/DEBUG /DEBUGTYPE:CV /PDB:None

❑❑❑

5 Debugging Applications

This chapter contains a sample PL/M program demonstrating task communication. A description of the program is included. The program compiles without errors, however, it does not run due to an error. The error exists to show the debug process. A debugged version of this program is also provided.

This chapter outlines a step-by-step process using the SDM monitor (SDM) and System Debugger (SDB) commands to locate the error, fix it, and then test the corrected code. Additional debugging techniques and commands are also provided in addition to instruction on running the example.

Example Application Program

This program includes three tasks:

- An initialization task, called Init, that creates a mailbox and the two other tasks
- A task called Alphonse that exchanges messages using mailboxes
- A task called Gaston which exchanges messages using mailboxes like Alphonse

The debug (error) version of the source code is listed in this chapter. These files are located in:

/rmx386/demo/plm/sdb/alphonse.plm
/rmx386/demo/plm/sdb/gaston.plm
/rmx386/demo/plm/sdb/init.plm

The version of this program which does not contain an error is in:

/rmx386/demo/plm/intro/alphonse.plm
/rmx386/demo/plm/intro/gaston.plm
/rmx386/demo/plm/intro/init.plm

Note

To run the errorless program in the */rmx386/demo/plm/intro* directory, first attach to the directory, then compile the program by entering **`make`**. Finally, run the program by entering **`tskcom32`**.

This *makefile* creates the PL/M multitasking demo and the *tskcom32* program described below.

See also: Designing an Application, Chapter 3, for more information on the PL/M multitasking demo.

This is how the corrected program (*tskcom32*) works:

1. The application code runs as a Human Interface (HI) program. Enter the name of the program at the HI prompt.
2. The task called Init runs first. This task creates a master mailbox and catalogs it in the root directory under the name Master. It creates the tasks Alphonse and Gaston then suspends itself.
3. When Gaston receives control, it:
   - Gets the token for the mailbox created by Init. Gaston looks up the name Master in the root job's object directory.
   - Creates a segment in which it will place a message and a response mailbox to which Alphonse will send a reply.
   - Loops and places a message in the segment after displaying it on the screen, sends the segment to the master mailbox, then waits at the response mailbox for a reply.
4. When Alphonse receives control, it:
   - Gets the token for the mailbox created by Init by looking up the name in the root job's object directory.
   - Loops and waits at the mailbox for a message and checks to see if the token it received is a segment. If so, Alphonse places its own message in the segment (after displaying it on the screen), then sends the segment to the response mailbox. If it is not a segment, Alphonse exits the loop and deletes itself.

The two tasks, Alphonse and Gaston, synchronize by using the two mailboxes. Gaston sends a message to the first mailbox and waits at the second one before continuing. Alphonse waits at the first mailbox. When it receives a message, it sends a reply to the second mailbox and waits at the first for another message. This cycle continues for six messages.

After sending its sixth message, Gaston exits the loop. Instead of sending a segment to the master mailbox, Gaston displays a final message to the screen then sends the task token (the token for the Init task) to the mailbox. When Alphonse receives this token and finds it is not a segment, Alphonse exits its loop and deletes itself.

To finish the processing, Gaston causes the Init task to resume processing since Init suspended itself earlier. When Init takes over, it deletes both offspring tasks and returns control to the Human Interface level.

Include Files

The *init.plm* file uses both Nucleus and EIOS calls so it includes the external files for both these layers. The *alphonse.plm* and *gaston.plm* files use Nucleus and HI system calls so they include the external files for those two layers.

Each task must contain its own set of include files because each is a separately compiled module. If the tasks were all contained in the same program module, only one set of `$include` statements would be needed.

Compiling and Running the Code

The example code contains an error to invoke SDB. A *makefile* compiles and binds the example code (*init.plm*, *alphonse.plm*, and *gaston.plm*).

The PL/M compiler commands in *makefile* do not include controls for selecting the model of segmentation (small, compact, medium, or large) because the `$compact` control was already included in the source files.

The compiler produces three files of object code. If the PL/M compiler command did not specify names for the object code files, the files would be given the names *init.obj*, *alphonse.obj*, and *gaston.obj* by default.

After compiling, you must bind the object files with the iRMX interface libraries. The section from *makefile* shows the bind command lines:

```
sdbIII:alphonse.obj gaston.obj init.obj
       $(BND)
       init.obj,                  &
       alphonse.obj,              &
       gaston.obj,                &
       $(PLMLIB),                 &
       $(RMXLIB),                 &
       pr($@.mp1)                 &
       oj($@)                     &
       renameseg(code32 to code)  &
       segsize(stack(+2400))      &
       rc(dm(5000,0fffffH))
```

Bind the three object files, *init.ob3*, *gaston.ob3*, and *alphonse.ob3*, together with the two libraries *plm386.lib* and *rmxifc32.lib*. The `$(PLMLIB)` alias is for the */intel/lib/plm386.lib* library. This library is the standard PL/M library distributed with the compiler. The `$(RMXLIB)` alias is for the */rmx386/lib/rmxifc32.lib* library. This is the 32-bit compact version of the iRMX interface library.

The `object` control specifies the name of the executable file generated by BND386. In this case, the file is called *sdbiii*.

The `SEGSIZE(STACK(+2400))` control reserves 2400 bytes of stack in addition to the amount required by the program. This amount represents the amount required by iRMX applications that include the Human Interface.

See also: Resource and Stack Size Guidelines, Appendix A

The `rc(dm(5000,0fffffH))` control directs BIND386 to produce an STL (single-task loadable) module and to assign a minimum of 5000H bytes of dynamic memory to the module.

Debugging the Program

The sample program does not include error checking even though it contains an error. This is to demonstrate more features of the System Debugger (SDB). This section describes two approaches for using SDB to find the error and correct it.

The addresses and token values in these examples have been assigned by the system in this debugging session. Most of these values will change from session to session. In a debugging session, it is helpful to record the various addresses and tokens.

Invoking SDM freezes both the application code and the operating system code. However, you can disassemble and execute the application instructions by using SDM and SDB commands.

See also: *System Debugger Reference*

To compile the program, first attach to the directory, then invoke the makefile by entering:

```
- af /rmx386/demo/plm/sdb <CR>
- make <CR>
```

This command produces an executable file called *sdbiii*. To run *sdbiii*, type this at the Human Interface prompt.

```
- sdbiii <CR>
```

Debugging Approach #1

When the sample program runs, the system displays this message:

```
Interrupt 13 at c4f0:00000399 General Protection ECODE =00000000
..
```

The values `c4f0:00000399` are where the Code Segment and Instruction Pointer Registers (CS:EIP) were pointing when the program halted. (The CS value of `c4f0` varies with each invocation of the application.) The prompt (`..`) indicates that SDM is active. However, since the program has been executed, you must re-enter SDM to re-execute the code. Use the CLI-restart feature to return to the Command Line Interpreter (CLI). This command works only if the existing CS:EIP is GDT-based protected mode code.

To restart the CLI, enter:

```
..g 284:1c <CR>
```

The system responds with the Human Interface prompt (-). Next, enter:

```
- debug sdbiii <CR>
```

The system responds with:

```
SEGMENT MAP FOR JOB: 84A8

NAME    BASE    NAME    BASE    NAME    BASE    NAME    BASE

LDT(2)  C998    LDT(3)  C9A0    LDT(4)  C9A8

Break At c998:00000000
..
```

Use SDM's **g** (go) command to set a breakpoint at the instruction where the program halted (remember the CS:EIP value is given in the interrupt message displayed when the program halts). The code segment (CS) value will change each time you re-enter SDM, but the instruction pointer (EIP) will remain the same. Enter:

```
..g,399 <CR>
Break At c998:00000399
```

To find out where you are in the code, use SDM's **d** (display) command to display a disassembled block of code. Enter:

```
..10 dx, <CR>
```

The system displays this code:

```
c998:00000399H F366A5            rep    movsw
c998:0000039CH 1E                push   ds
c998:0000039DH 07                pop    es
c998:0000039EH B800000000        mov    eax,0
c998:000003A3H 8BD0              mov    edx,eax
c998:000003A5H 52                push   edx
c998:000003A6H 50                push   eax
c998:000003A7H 6800000000        push   0
c998:000003ACH 668B057A000000    mov    ax,word ptr 07a
c998:000003B3H BF00000000        mov    edi,0h ——
```

The instruction at address `c998:00000399` is a move string word instruction. The only move word instruction in the sample program is the PL/M `MOVW` call when Gaston enters the loop after creating the segment.

```
response$mbox = RQ$CREATE$MAILBOX (     /* Create response mailbox */
       fifo,
       @status);

seg$token = RQ$CREATE$SEGMENT(          /* Create message segment */
       seg$size,
       @status);

DO WHILE count < final$count;
       message.count = 23;

       CALL MOVW (@main$message, @message.text, SIZE(main$message));

       CALL RQ$C$SEND$CO$RESPONSE (     /* Send message to screen */
              NIL,
              0,
              @message.count,
              @status);
```

If displaying the instruction does not provide enough information about why the program halted, look at the surrounding code by displaying forward or backward from the CS:EIP. Because you specified a comma in the previous DX command, you can display forward another 10 instructions from the current CS:EIP by entering only a comma (`,`). However, since the instruction where the exception occurred is traceable to the sample code, you know where the program fails. Refer to Debugging Approach #2 for displaying backward from the CS:EIP.

To examine what happens when the system tries to move the message, return to the protected-mode prompt (by entering a <CR>) and examine register contents before and after `MOVSW` is executed. Enter this command:

```
..x <CR>
```

The system displays this:

```
EAX=07e4ca88  CS=c998  EIP=00000399  EFL=00013297  LDTR=02a0
EBX=00000072  SS=ca70  ESP=000007fc  EBP=000007fc  TR  =0278
ECX=00000017  DS=c9a0  ESI=0000007c  FS =ca88      MSW =fffb
EDX=0000ca88  ES=ca88  EDI=00000001  GS =0034
GDTR .BASE=00110000 .LIMIT=0f9ff
IDTR .BASE=0011fa00 .LIMIT=007ff
```

To execute the MOVSW instruction, enter:

```
..n, <CR>
```

The system displays:

```
c998:00000399H F366A5                    rep movsw   -
```

Enter a comma:

```
, <CR>
```

The system responds with:

```
Interrupt 13 at c998:00000399 General Protection ECODE =00000000
..
```

To see how executing this instruction changed register contents, enter:

```
..x <CR>
```

The system displays:

```
EAX=07e4ca88  CS=c998  EIP=00000399  EFL=00003297  LDTR=02a0
EBX=00000072  SS=ca70  ESP=000007fc  EBP=000007fc  TR  =0278
ECX=00000008  DS=c9a0  ESI=0000009a  FS =ca88      MSW =fffb
EDX=0000ca88  ES=ca88  EDI=0000001f  GS =0034
GDTR .BASE=00110000 .LIMIT=0f9ff
IDTR .BASE=0011fa00 .LIMIT=007ff
```

In the assembly language MOVSW instruction, DS:ESI represents the source from which the data is moving; ES:EDI is the destination and ECX is the count.

See also: MOVSW, *ASM386 Macro Assembler Operating Instructions/ASM386 Assembly Language Reference*

To check the limit of the ES register, enter:

```
..ddt(es) <CR>
```

The system displays:

```
GDT(6481T) DSEG32 BASE=002ecce0 LIMIT=0001f P=1 DPL=0 ED=0 W=1 A=1 G=0
..
```

The `LIMIT` parameter shows that the segment limit is 1FH (31 decimal). Since the system counts from zero, the segment size is 32 decimal, which is the value assigned to `seg$size` in Gaston. The EDI register tries to move the word into memory at ES:1FH and 20H when the error occurred. The system was trying to write past the segment limit of 1FH into 20H when the program halted. This suggests the PL/M `MOVW` instruction should be changed to a `MOVB` instruction. At this point, you could exit SDM, change the PL/M code, then recompile and run it.

However, you can use SDM's **x** (examine/modify) command to change a register value and the **g** command to execute the program. Making changes with the **x** and **s** (substitute) commands enables you to test code without having to recompile and bind it.

The ECX register contains the count of bytes or words moved. If you decrease the count in the ECX register from 17 to 15 before you execute the `MOVSW` instruction, you should be able to move all the data. Exit and re-enter SDM and set a breakpoint at the `MOVSW` instruction by entering:

```
..g 284:1c <CR>
-debug sdbiii <CR>
..g,399 <CR>
```

Set the ECX register to 15. Enter:

```
..x ecx=f <CR>
```

Now, execute the rest of the program by entering:

```
..g <CR>
```

The system responds with:

```
After you, Alphonse

After you, Gaston
Interrupt 13 at cec8:00000399 General Protection ECODE =00000000
..
```

Since the change was valid for one pass through the code, the first pass through the Gaston loop worked. The next pass failed.

To return to the CLI, enter:

```
..g 284:1c <CR>
```

This partially successful run shows that if you reduce the number of words moved, the program works. Therefore, to make a permanent fix, you should change the PL/M `MOVW` call to `MOVB` in the sample code, then recompile and bind it.

Debugging Approach #2

You can also make changes in the disassembled code. Suppose you have run the program for the first time, and the system displayed this message:

```
Interrupt 13 at 6368:00000399 General Protection ECODE =00000000
..
```

Restart the system using the CLI-restart feature as you did in Debugging Approach #1, then re-enter SDM by entering:

```
-debug sdbiii <CR>
```

Set a breakpoint at the instruction that was executing when the program failed and display a block of disassembled code by entering:

```
..g,399 <CR>
..5 dx <CR>
```

The system displays:

```
8340:00000399H F366A5                    rep movsw
8340:0000039CH 1E                        push   ds
8340:0000039DH 07                        pop    es
8340:0000039EH B800000000                mov    eax,0
8340:000003A3H 8BD0                      mov    edx,eax
```

To look at the instructions preceding MOVSW, enter:

```
..14 dx cs:eip - 25 <CR>
```

The system displays this code:

```
8340:00000374H 7A00                jle    $+02 ;a=00000376
8340:00000376H 0000                add    byte ptr [eax],al
8340:00000378H 64C6050000000017    mov    byte ptr fs:0,17
8340:00000380H BE7C000000          mov    esi,7c
8340:00000385H 668B057A000000      mov    ax,word ptr 7a
8340:00000391H B917000000          mov    ecx,17
8340:00000396H 8ECO                mov    es,ax
8340:00000398H FC                  cld
8340:00000399H F366A5              rep movsw
8340:0000039CH 1E                  push   ds
8340:0000039DH 07                  pop    es
8340:0000039EH B800000000          mov    eax,0
8340:000003A3H 8BD0                mov    edx,eax
```

MOVSW is a repetitive move from DS:ESI to ES:EDI. Looking at the preceding instructions, you see the instruction at address 8340:00000391 moves 017H into ECX. Remember that ECX is the count of bytes or words moved. To display the ES register contents, use this command line:

```
ddt(es) <CR>
```

The screen displays:

```
GDT(6481T) DSEG32 BASE=002ecce0 LIMIT=0001f P=1 DPL=0 ED=0 W=1 A=1
..
```

As in the last example, you can check the limit. Since the segment size is 32 (decimal) and the system is trying to write 17H words, the system fails when it tries to write past the segment limit. To reduce this count you must move the data. Re-enter SDM and, using the SDM **s** command, change the code at `8340:00000391` by entering the following instructions outlined in bold:

Screen Input/Output	Comments
..g 284:1c <CR>	
-debug sdbiii <CR>	
..s cs:391 <CR>	Enter SDM to substitute memory at EIP=00000391.
e110:00000391 b9 - **,**	Enter comma to step the count.
e110:00000392 17 - **f <CR>**	Enter the new count.
..g <CR>	Re-start code execution.

The system responds with six iterations of this:

```
After you, Alphonse

After you, Gaston
.
.
.
```

After six iterations of the previous screen, the monitor displays:

```
If you insist, Alphonse
-
```

Viewing System Objects

Consider that a problem you are experiencing could be deadlock. By looking at system objects at various stages of execution, you can observe how synchronization (or lack of it) is occurring. To do this you use SDM commands

You can view any object in a job using the **vo** (view job object) command (specifying the job's token) to provide the broad picture of the system state, then the **vt** (view token, or display iRMX object) command to focus on individual elements. Suppose, you want to view the state of the objects before entering the loop in which Gaston and Alphonse exchange messages. Assume you have stepped through the code, verifying system calls until you located the CS:EIP for the Nucleus **create_segment** system call in Gaston. Re-enter SDM and set a breakpoint at this CS:EIP by entering:

```
-debug sdbiii <CR>
..g,352 <CR>
```

To get the job token, enter:

```
..vj <CR>
```

The system displays this screen output. The values in the output may differ from yours. Comments have been added to the output.

Job Token (iRMX Job Tree)	Comments
0258	Root Job
11b8	Human Interface
4f38	Command Line Interpreter
b7e0	Application Job
3f70	EIOS
3968	iRMX NET
3238	BIOS

The token for the application job in this output is `b7e0`. To view objects for this job, enter:

```
..vo b7e0 <CR>
```

The system displays:

```
Child Jobs:
Tasks:       c250     c170    c108
Mailboxes:   c238 t   c098
Semaphores:
Regions:
Segments:    c2a0     c3c0    c418    c100    c8a8    c850
             c700     c740    c1f0    c120
Extensions:
Composites:  bc10     c7a0
Buffer Pools:
..
```

At this stage of program execution, two mailboxes exist. The `t` following mailbox `c238` means one or more tasks are waiting at this mailbox (Alphonse was created first and is waiting for a message from Gaston). Examine mailbox `c238` by entering:

```
..vt c238 <CR>
```

The system responds with:

```
Object type = 3  Mailbox

Mailbox type        Object   Task queue head     c170
Queue discipline    FIFO     Object queue head   0000
Containing job      b7e0     Object cache depth  08

Task queue  c170
```

Use SDB's **u** (display system calls in a task's stack) command to view the waiting task's stack. To unwind the stack, enter:

```
..vu c170 <CR>
```

The system displays:

```
gate #0430

Return cs:eip - c850:0000020f
c1f0:000007e4 00000040 8075c700 0000003e 0000c700 0000ffff 0000c238
c1f0:000007fc 00000000

(Nucleus)  receive message

             |.....excep$p.....|...response$p....|..time..|..mbox..|
```

You can continue to examine objects or set a breakpoint at the return CS:EIP.

Set the CS:EIP by entering:

```
..g, 20f <CR>
```

This causes SDM to display:

```
Interrupt 13 at c850:00000399 General Protection ECODE =00000000
```

This message indicates that the program halts in Gaston and that `c850:00000399` is the last instruction executing.

Alternative Debugging Techniques

This chapter has shown two ways to find an error and two ways to make temporary fixes from the SDM/SDB. The message displayed when the program halts contains the `CS:EIP` of the last instruction executing. If setting the `CS:EIP` at this instruction and displaying the surrounding code does not help you locate this point in your application code, there is another method.

Use combinations of the **vj**, **vo**, **vt**, **vu**, and **vs** commands to locate the running task. Then set the breakpoint at the `CS:EIP` of the last executing instruction and display code, objects, and registers to determine how the system is executing that instruction.

❑❑❑

Porting Applications 6

This chapter discusses porting existing 16-bit iRMX II code to the 32-bit iRMX III, DOSRMX, or iRMX for PCs OS. The topics covered are:

- Three different approaches to porting iRMX code
- The compiler switches used to port code
- Language differences for PL/M, C, and ASM
- An example of porting a device driver
- Porting code to PC-bus systems

Before porting code, learn the data types recognized by iRMX OSs. Mismatching data types when porting code cause program errors.

See also: Data Types, *System Call Reference*

Porting Code from 16-Bits to 32-Bits

Migrating from 16-bit iRMX II-based applications to 32-bit iRMX III-based applications increases performance if large data manipulations or numerics are involved. It also makes code easier to maintain. Use one of these porting strategies to port your code:

- Use the existing 16-bit object files without any changes.
- Port only the code that gains in performance due to the change to 32 bits.
- Port the entire application to 32 bits.

In the following situations, however, you should not port to 32 bits:

- If the platform on which the application will run uses an Intel 80286 microprocessor and there is no performance reason or other need to move to an Intel386 or higher microprocessor. The iRMX III OS requires an Intel386 or higher microprocessor.
- If all computations only involve integers smaller than 64 Kbytes (65,536 bytes) and there is no present or foreseeable need to use contiguous memory areas larger than 64 Kbytes.

- Because the Intel386 microprocessor object module format (OMF386) does not support memory overlays, iRMX III cannot support overlay loading in 32-bit applications. iRMX II applications that use overlays can still execute in 16-bit compatibility mode.
- Applications written in 16-bit require more code and data space (an average of 30%) when ported to 32 bits. Additional space is required for the OS itself. If there are severe constraints on memory in the system, you should not port to 32 bits.
- In certain cases, the application may be written using a 16-bit compiler for which no 32-bit compiler is available.

Using Existing 16-Bit Code

Most 16-bit iRMX II executable code does not need to be recompiled for 32-bit iRMX systems. These 16-bit applications run together with 32-bit applications without change. For example, the iRMX II **dir** command can be used on an iRMX III system without changes.

iRMX II applications (either run-time loadable or configured as first-level jobs) will run under iRMX III without modification as long as they do not include 16-bit interrupt-handlers, device-drivers, and OS extensions. Such applications execute in 16-bit compatibility mode.

16-bit C (compiled with iC-286 V4.1 or later) and 16-bit PL/M programs are also fully binary compatible with iRMX III provided no 16-bit device drivers, interrupt handlers or OS extensions are used. However, C applications may be more stack-intensive than PL/M applications. They may run out of stack space under iRMX III unless they are allocated additional stack size using the SEGSIZE control in BND286.

Advantages of 32-Bit Application Code

This list describes situations in which it is an advantage to port from 16 bits to 32-bit code.

- Applications containing intensive computations with unsigned integers larger than 64 Kbytes (65,536) or signed integers larger than 32 Kbytes (32,768) will run faster.
- Intel386, Intel486, and Pentium microprocessors offer several bit and bit-string manipulation instructions. Applications that do bit-field manipulation in software could improve their performance. Applications that previously used bytes to store binary flags could be rewritten much more compactly.

- Applications where the processor might access memory across a 32-bit bus, like Multibus II, will access it faster.
- When there is a 32-bit interface between the microprocessor, the numeric processor, and memory; floating-point applications will see a moderate performance boost because operands are transferred in 32-bit blocks to and from the processor.
- When manipulating large data arrays, you can use fewer segments because you are not constrained to the 64 Kbyte size limitation. Data is now accessed in a single, large (up to 4 Gbytes) segment, which saves the overhead of multiple segment manipulation. Reading and writing this segment from and to mass storage is also faster because a single I/O call is used instead of multiple 64 Kbyte-constrained I/O calls.

Porting Entire Applications to 32-Bits

You must recompile and rebind all the code when porting your entire application system. Although it requires greater effort, this method provides the best overall performance.

This list describes important considerations when re-generating 16-bit code into 32-bit code.

- The logical pathname (*:rmx:*) points to the */rmx386* directory instead of */rmx286*. The directory *:rmx:inc* contains files with EXTERNAL declarations for the iRMX and UDI calls in the PL/M source.
- You must bind the 32-bit iRMX III code with the 32-bit iRMX and UDI interface libraries (*rmxifc32.lib*, *udiifc32.lib*, in this example).
- When binding compact model object files, a RENAMESEG control must be used to rename the code segment (output by PL/M-386) from CODE to CODE32. The code segments of the *rmxifc* and *udiifc* libraries are already named CODE32. In the compact model, only one code segment is allowed and BND386 can only combine segments that have the same name.
- Use 32-bit word sizes if the 16-bit application being ported has:
  - Any arithmetic operation involving DWORDs (in PL/M-286) or long/double declarations in C-286.
  - String searching/copying operations (CMPB/ CMPW/MOVB/MOVW in PL/M) are limited to 64 Kbyte segments with a 16-bit OS. All physical memory can be covered by one 32-bit operation.

– Certain variable declarations at the start of each source module and procedure/function, especially at the size of arrays. Any arrays of close to 64 Kbyte size, or 32 Kbyte 16-bit WORD size, may benefit from being extended.

– 80286 code which performs bit manipulation routines. Performance may be increased by re-coding with 32-bit microprocessor-based functions. These functions may have to call assembler routines to access these bit manipulation functions.

Porting 16-Bit PL/M Code to 32 Bits

Once you decide how much application code needs to be ported, you must choose between two porting processes. The only difference between the two methods is the invocation switches on the compiler:

WORD16 switch	This is typically the easiest method to use when porting code. This switch causes all WORD values to remain 16-bits and all DWORD values to remain 32-bits. First, edit your source file to change the data types of variables that can be larger. For example, variables containing the offset of indirect near calls and those that indicate the size of data transfers should be changed to a DWORD value. Then compile your source code using the WORD16 switch.
No switches	Compile the code you select for porting using the PL/M-386 compiler and no switches. This forces a default value of 16 bits for each HWORD value, 32 bits for each WORD value and a 64-bit value for each DWORD value. Because 64-bit arithmetic is much slower than 32-bit arithmetic, you should carefully review the existing DWORD variables. Those variables that need to be only 32-bit values should be changed to WORD variables.

When converting 16-bit PL/M code to 32 bits, you must:

- Change the WORD data type to WORD_16
- Change the DWORD data type to WORD_32
- Use the WORD16 compiler switch

Differences Between PL/M-386 and Previous PL/M Code

This section describes differences between code that was compiled using versions of the PL/M compiler other than PL/M-386. If you are using binary compatibility and not recompiling your code, you do not need to make changes. Some of these differences are changes to the iRMX OS, others are changes to the compiler. Each difference is explained along with any changes you need to make are:

- OFFSET is a reserved word in PL/M-386. If you are porting code to 32 bits and your code contains variables named OFFSET, change these variable names. For example, change:

  ```
  DECLARE OFFSET      WORD;
  ```

  To:

  ```
  DECLARE OFF_SET     WORD_32;
  ```

- The limits of the PL/M built-in string functions, such as CMPB, FINDB, SKIPB, SETB, MOVB, CMPW, SETW, and so on, have increased from 0FFFFH to 0FFFFFFFFH. This enables searches of buffers that are greater than 64 Kbytes in length. You can force the buffer length to remain 64 Kbytes by means of truncation. That is, you place the result of the CMPB and FINDB functions into WORD_16 variables and truncate the upper 16 bits. Be sure your code does not attempt to search past the end of your forced 64 Kbyte segment.

- Change all WORD_16 variables that contain the offset of a POINTER to WORD_32 variables. For example, change:

  ```
  DECLARE
  PTR$OVERLAY  LITERALLY 'STRUCTURE(offset WORD, base TOKEN)';
  ```

  To:

  ```
  DECLARE
  PTR$OVERLAY  LITERALLY 'STRUCTURE(off_set WORD_32,
                                    base TOKEN)';
  ```

- Change all variables that reference data transfer counts from WORD_16 values to WORD_32 values. For example, change:

```
DECLARE
   save$count          WORD,
      .
      .
      .
     save$count = iors.count;
```

To:

```
DECLARE
   save$count          WORD_32,
      .
      .
      .
     save$count = iors.count;
        /* iors.count is now a 32-bit value /*
```

Porting 16-Bit C Code to 32 Bits

These sections describe the main concerns when creating or modifying 16-bit code which will be ported to 32 bits. The two main concerns are:

- Including the *rmx_c.h* file and using its types
- Using the NATIVE_WORD type for variables which will expand from 16 bits to 32 bits when porting your application

Using the rmx_c.h Header file

The */intel/include/rmx_c.h* file provides definitions for system calls, structures and other items needed for iRMX application development. Including this file and using its definitions throughout your application enables much easier conversion of that code from 16-bit to 32-bit source.

See also: Header Files, *System Calls*

Using the NATIVE_WORD Type Definition

Type definitions of variables which expand from 16 bits to 32 bits when porting to 32-bit code should use the NATIVE_WORD type definition. Examples of these variables are:

- I/O counts
- Memory pool sizes
- Stack sizes
- Segment sizes
- Application-specific variables which must expand to 32 bits

This example uses NATIVE_WORD and includes a pointer overlay:

```
typedef struct exception_struct {
        NATIVE_WORD       offset;
        SELECTOR          base;
        BYTE              exception mode;
};
```

The I/O count in this iRMX system call uses NATIVE_WORD:

```
rq$a$write (output$conn$t, (BYTE *) message,
           (NATIVE_WORD) strlen(message), write$mbx,
           &status;
```

Porting 16-Bit ASM Code to 32 Bits

If you use ASM386, you must use registers differently. These sections describe the differences.

- Properly clear all registers used as index or scratch locations to check for zero. If they are not properly cleared, bits left in the extended (upper 16 bits) of the register may interfere with the intended operation. To properly clear registers change:

```
mov       ax, word ptr ds:8
or        ax, ax
jz        ...
```

  To:

```
movzx     eax, word ptr ds:8
or        eax, eax
jz        ...
```

- Use two `shl` (shift left) statements before a jump in the index to a case statement. To properly increment an index, change:

```
xor       bh, bh
mov       bl, cdate.interrupt_type
and       bl, ts_more_ints
shl       bl, 1                ; Make bx a pointer to a
                               ; 16-bit word to index
                               ; into case_table
jmp       cs:case_table[bx]
```

  To:

```
xor       ebx, ebx
mov       bl, cdata.interrupt_type
and       bl, ts_more_ints
shl       ebx, 2               ; Make bx a pointer to a
                               ; 32bit word to index
                               ; into case_table]
jmp       cs:case_table[ebx]
```

- PL/M-like procedures that return pointers now place the POINTER in DX:EAX instead of ES:BX. For example, change:

```
mov       es, ptr_base
mov       bx, ptr_offset
ret
```

  To:

```
mov       dx, ptr_base
mov       eax, ptr_offset
re
```

- Change interrupt handlers written in assembly language to run in the 32-bit environment. This example shows an interrupt handler for the 16-bit system:

```
int_handler     proc      near
     public     cominthandler

pusha                ; save the processor state
push       ds
push       es

push       cx        ; make room for status
mov        bp, sp    ; ss:bp is status$p
```

```
push       ss
push       bp
call       rqgetlevel
push       ax        ; returned level

push       ss        ; ax = rq$get$level(status$p)
push       bp
call       rqsignalinterrupt

pop        cx
pop        es
pop        ds
popa

iret                 ; return from interrupt
int_handler   endp

code       ends
end
```

This is an interrupt handler ported to a 32-bit system. Note the IF-ELSE statement that is added to this example. This IF block enables using the same code on 16-bit and 32-bit systems, depending on which assembler is used and how it is invoked.

```
%IF (%r_32) THEN (%'    ; macro definitions which
    %define (ax) (eax)  ; allow code to go both ways
    %define (bx) (ebx)
    %define (cx) (ecx)
    %define (dx) (edx)
    %define (si) (esi)
    %define (di) (edi)
    %define (bp) (ebp)
    %define (sp) (esp)
    %define (mov16) (movzx)
    %define (pusha) (pushad)
    %define (popa) (popad)
    %define (pushf) (pushfd)
    %define (popf) (popfd)
    %define (iret) (iretd)
    %define (dw) (dd)
    %define (dd) (dp)

) ELSE (%'
```

```
            %define (ax) (ax)
            %define (bx) (bx)
            %define (cx) (cx)
            %define (dx) (dx)
            %define (si) (si)
            %define (di) (di)
            %define (bp) (bp)
            %define (sp) (sp)
            %define (mov16) (mov)
            %define (pusha) (pusha)
            %define (popa) (popa)
            %define (pushf) (pushf)
            %define (popf) (popf)
            %define (iret) (iret)
            %define (dw) (dw)
            %define (dd) (dd)

        )FI%'
               int_handler     proc      near
                    public     cominthandler

               %pusha             ; save the processor state
               push    ds
               push    es
%IF  (%r_32) THEN(push    fs
               push    gs)  FI

               push    %cx         ; make room for status
               mov     %bp, %sp    ; ss:bp is status$p

               push    ss          ; ax = rq$get$level(status$p)
               push    %bp
               call    rqgetlevel

               push    %ax ; CALL rq$signal$interrupt(ax, status$p)
               push    ss
               push    %bp
               call    rqsignalinterrupt

               pop     %cx         ; pop status
                                   ; restore processor state
```

```
%IF  (%r_32) THEN(pop     gs
                  pop     fs)  FI
                  pop     es
                  pop     ds
                  %popa
                  %iret               ; return from interrupt
                  int_handler   endp

                  code    ends

                  end
```

To assemble this example, select one of these statements:

```
ASM286 inthand.asm object(inthand.ob2) pr(inthand.ls2) %SET(r_32,0)

ASM386 inthand.asm object(inthand.ob3) pr(inthand.ls3) %SET(r_32,1)
```

Example: Porting a Device Driver

This section contains a portion of an example device driver (8274 Terminal Driver) ported to the iRMX OS. Though changes to the driver are minimal, you must also port the include files and libraries. In this code, the PL/M compiler's and Assembler's SET controls, a PL/M identifier, permits IF-ELSE branches while compiling the code.

```
PLM386 :F1:x8274.P28 SET(r_32)word16  ; for 32 bits
PLM286 :F1:x8274.P28 RESET (r_32)     ; for 16 bits
PLM86  :F1:x8274.P28 SET(tsc) RESET(r_32)
```

Two identifiers are used: `tsc` and `r_32`. The `r_32` identifier is used to port the code to the iRMX OS. IF-ELSE decision blocks were added so the same code can be compiled into a driver for both the 32-bit and 16-bit versions of the OS. The LIB statements for the 8274 Driver are:

```
LIB386 :F1:xcmdrv.lib nobu  ; for 32-bit systems
       delete x8274
       add :F1:x8274.obj
       compress
       quit
       exit
LIB286 :F1:xcmdrv.lib nb    ; for 16-bit systems
       delete x8274
       add :F1:x8274.obj
       compress
       quit
       exit
LIB86
       delete :F1:xcmdrv.lib(x8274)
       add :F1:x8274.obj to :F1:xcmdrv.lib
       exit
```

Figure 6-1 is a device driver example which uses the r_32 porting identifier.

```
$title('x8274: 8274 terminal device driver')
/*
 *  Allow iRMX I/II common source
 */
$IF tsc
    $OPTIMIZE(3)
    $COMPACT(tsc -CONST IN CODE- HAS x8274)
    $large ( other_libs
    $EXPORTS RQ$GetTaskTokens;
    $EXPORTS RQ$LookupObject;
    $EXPORTS RQ$CreateSegment;
    $EXPORTS RQ$DeleteSegment)
$ELSE
    $COMPACT
    $ROM
    $OPTIMIZE(3)
$ENDIF

$subtitle('Module Header')
/*
 *
 * TITLE:        x8274
 *
 * ABSTRACT:     This module is the interface between the iRMX286
 *               Terminal Support, and the 8274 MPSC.
 *

x8274:
DO;

$include(:f1:xcomon.lit)
$include(:f1:xnutyp.lit)
$include(:f1:xiotyp.lit)
$include(:f1:xexcep.lit)
$include(:f1:xtsdtn.lit)

$include(:f1:xtssow.ext)
$include(:f1:xgdlay.ext)
$include(:f1:xncall.ext)
```

Figure 6-1. Device Driver Example Using r_32 Conditional Statements

```
$subtitle('Data structures and literals')
/*
 *    8274 register values
 */
DECLARE
    WR0                LITERALLY '00H',
    WR1                LITERALLY '01H',
    .
    .
    .
/*
 *    8274 Device information Structure
 */
DECLARE
    i8274$CONTROLLER$INFO   LITERALLY  'STRUCTURE(
                                                    i8274$INFO$1,
                                                    i8274$INFO$2,
                                                    i8274$INFO$3,
                                                    i8274$INFO$4,
                                                    i8274$INFO$5,
                                                    i8274$INFO$6,
                                                    i8274$INFO$7)';
DECLARE
$IF r_32
    i8274$INFO$1    LITERALLY    'filler(22)        WORD',
$ELSE
    i8274$INFO$1    LITERALLY    'filler(13)        WORD',
$ENDIF
    i8274$INFO$2    LITERALLY    'ch_a_data_port         WORD,
                                  ch_a_status_port       WORD,
                                  ch_b_data_port         WORD,
                                  ch_b_status_port       WORD',
    i8274$INFO$3    LITERALLY    'ch_a_in_rate_port      WORD,
                                  ch_a_in_rate_cmd_port  WORD,
                                  ch_a_in_rate_counter   BYTE,
                                  ch_a_in_rate_freq      DWORD',
       .
       .
       .
```

Figure 6-1. Device Driver Example Using r_32 Conditional Statements (continued)

```
$IF r_32
  DECLARE
    SIZE$OF$OFFSET LITERALLY 'DWORD'; /* Support for larger segments*/
$ELSE
    /*  Note that either type of segmentation is supported */
  DECLARE
    SIZE$OF$OFFSET LITERALLY 'WORD';
$ENDIF

DECLARE
   BOOLEAN         LITERALLY 'BYTE',
   TRUE            LITERALLY '0FFH',
   FALSE           LITERALLY '000H',
   FOREVER         LITERALLY 'WHILE TRUE',

   PTR$OVERLAY     LITERALLY 'STRUCTURE(off_set SIZE$OF$OFFSET,
                                        base TOKEN)',
   P$OVERLAY       LITERALLY 'STRUCTURE(off_set SIZE$OF$OFFSET,
                                        base WORD)',

   STRING          LITERALLY 'STRUCTURE(length BYTE, char(1) BYTE)',
   NO$TIME$LIMIT   LITERALLY '0FFFFH',
   .
   .
   .
```

Figure 6-1. Device Driver Example Using r_32 Conditional Statements (continued)

Figure 6-2 is a literal file which uses the r_32 porting identifier.

xtstdn.lit

```
/*
 *     xtsdtn.lit
 *
 *     Terminal Support cdata, udata, and bddata structures as
 *     available to the user for the purpose of writing a terminal
 *     driver which is compatible with the Terminal Support Code.
 *     This file has the same structure as xtsdat.lit but only
 *     defines that portion of the structure which is visible to the
 *     user.
 *
 *     Defines RECV$INFO$STRUCT for MBII drivers
 *
 *     Defines a substructure TS$BDDATA4 which is the same as
 *     TS$BDDATA3 minus driver$user$only. This enables drivers to
 *     overlay a different structure over TS$UDATA (TS$UDATA1 +
 *     TS$UDATA2 + TS$BDDATA1 + TS$BDDATA2 + TS$BDDATA4 + a driver
 *     specific structure)
 *
 *     Adds 32 bit conditional support.
 */

DECLARE
       TS$CDATA     LITERALLY 'STRUCTURE(
                    ios$data$segment                SEGMENT,
                    status                          WORD_16,
                    interrupt$type                  BYTE,
                    interrupting$unit               BYTE,
                    dinfo$p                         POINTER,
                    driver$cdata$p                  POINTER,
$IF r_32
                    reserved(46)                    BYTE,
$ELSE
                    reserved(34)                    BYTE,
$ENDIF
                    udata(1)                        BYTE)';
```

Figure 6-2. Literal File Using r_32 Conditional Statements

```
 *     CDATA STRUCTURE duplicated here for use with UDATA members
 *     for single structure overlay
 */

DECLARE
       TS$CDATA$INC  LITERALLY
                     'ios$data$segment                SEGMENT,
                        status                        WORD_16,
                        interrupt$type                BYTE,
                        interrupting$unit             BYTE,
                        dinfo$p                       POINTER,
                        driver$cdata$p                POINTER,
$IF r_32
                     reserved1(46)             BYTE';
$ELSE
                     reserved1(34)             BYTE';
$ENDIF

DECLARE
       TS$UDATA      LITERALLY 'STRUCTURE(
                     TS$UDATA1,
                     TS$UDATA2,
                     TS$BDDATA1,
                     TS$BDDATA2,
                     TS$BDDATA3)';

DECLARE
       TS$UDATA1     LITERALLY
                     'uinfo$p                         POINTER,
                        term$flags             WORD_16,
$IF r_32
                     in$rate                          WORD_32,
                     out$rate                         WORD_32,
$ELSE
                     in$rate                          WORD_16,
                     out$rate                         WORD_16,
```

Figure 6-2. Literal File Using r_32 Conditional Statements (continued)

```
$ENDIF
                    scroll$number              WORD_16,
                    x$y$size                         WORD_16,
                    x$y$offset                       WORD_16',
       TS$UDATA2    LITERALLY
                    'raw$size                        WORD_16,
                       raw$data$p              POINTER,
                       raw$in                        WORD_16,
                       raw$out                       WORD_16,
                       output$scroll$count           WORD_16,
                       unit$number                   BYTE,
$IF r_32
                    reserved(1099)                   BYTE',
$ELSE
                    reserved(890)              BYTE',
$ENDIF

       TS$BDDATA1   LITERALLY
                    'buffered$device                 BYTE,
                       buff$input$state              WORD_16,
                       buff$output$state             WORD_16,
                       select(2)                     BYTE,
                       line$ram$p              POINTER,
                       function$id                   BYTE,
$IF r_32
                    in$count                         WORD_16,
$ELSE
                    in$count                         BYTE,
$ENDIF
                    out$count                        WORD_16',

       TS$BDDATA2   LITERALLY
                    'units$available                 WORD_16,
                       output$buffer$size            WORD_16,
                       user$buffer$p                 POINTER,
                       echo$count              BYTE,
                       echo$buffer$p                 POINTER,
                       received$special              WORD_16,
```

Figure 6-2. Literal File Using r_32 Conditional Statements (continued)

```
                        special$modes                   WORD_16,
                        high$water$mark                 WORD_16',

       TS$BDDATA3    LITERALLY
                     'low$water$mark                   WORD_16,
                        fc$on$char              WORD_16,
                        fc$off$char                     WORD_16,
                        link$parameter                  WORD_16,
                        spc$hi$water$mark               WORD_16,
                     special$char(4)                    BYTE,
$IF r_32
                     bd$reserved(41)                    BYTE,
                     driver$use$only(48)                BYTE';
$ELSE
                     bd$reserved(25)                    BYTE,
                     driver$use$only(32)                BYTE';
$ENDIF

/* Note! TS$BDDATA4 must be same as TS$BDDATA3 minus
   driver$use$only */
DECLARE
       TS$BDDATA4    LITERALLY
                     'low$water$mark                   WORD_16,
                        fc$on$char              WORD_16,
                        fc$off$char                     WORD_16,
                        link$parameter                  WORD_16,
                        spc$hi$water$mark               WORD_16,
                        special$char(4)                 BYTE,
$IF r_32
                     bd$reserved(41)                    BYTE';
$ELSE
                     bd$reserved(25)                    BYTE';
$ENDIF
```

Figure 6-2. Literal File Using r_32 Conditional Statements (continued)

```
DECLARE
$IF r_32
       TS$UDATA$SIZE       LITERALLY           '1280',
       TS$CDATA$SIZE       LITERALLY           '40H';
$ELSE
       TS$UDATA$SIZE       LITERALLY           '1024',
       TS$CDATA$SIZE       LITERALLY           '30H',
       TS$UDATA$FACTOR             LITERALLY           '10';
$ENDIF

DECLARE
       INPUT$ONLINE                LITERALLY           '0001H',
       INPUT$CMD$PENDING           LITERALLY           '0002H',
       INPUT$FULL                  LITERALLY           '0008H',
       RAW$BUFF$FULL       LITERALLY           '0010H';

DECLARE
       OUTPUT$SEMAPHORE            LITERALLY           '001H',
       OUTPUT$STOPPED              LITERALLY           '002H',
       OUTPUT$SCROLL       LITERALLY           '004H',
       OUTPUT$CONTROL              LITERALLY           '008H';

DECLARE
       FLOW$CONTROL                LITERALLY           '001H',
       SPECIAL$CHAR$MODE           LITERALLY           '002H';

DECLARE
       NON$BUF$DEV$RAW$SIZE        LITERALLY           '100H';

/* Structure for passing MBII messages to term$check */
```

Figure 6-2. Literal File Using r_32 Conditional Statements (continued)

```
DECLARE
       RECV$INFO$STRUCT LITERALLY
       'STRUCTURE(
             data$p              POINTER,
             flags               WORD_16,
             status              WORD_16,
             trans$id            WORD_16,
             data$length         WORD_32,
             forwarding$port     TOKEN,
             remote$socket WORD_32,
             control$msg(20)     BYTE,
             reserved(4)         BYTE)';
/* Structure for passing Mailbox messages to term$check */

DECLARE
       MBOX$RECV$INFO$STRUCT LITERALLY
       'STRUCTURE(
             object$t     TOKEN,
             resp$mbox$t  TOKEN)';
```

Figure 6-2. Literal File Using r_32 Conditional Statements (continued)

Migrating Code to a PC-Bus Platform

This section discusses the differences between the way a PC-bus system and other systems handle numeric processors. Be aware of these differences when porting code to a PC-bus system from a different system.

Using a Numeric Processor Extension (NPX)

You can increase the performance of math-intensive tasks by using a Numeric Processor Extension (NPX) or math coprocessor to perform the math functions. In systems that use a math coprocessor, the processor and the microprocessor are synchronized by a busy signal from the numeric processor. In a PC-bus system, this numeric error signal is routed through the programmable interrupt controllers (PICs). The numeric error signal is connected to the slave PIC interrupt 5, which is connected to the master PIC interrupt 2.

The OS, through task prioritization, automatically disables certain interrupt levels when a task runs. The levels disabled depend on the priorities of the current and previous tasks. If a task can create a physical interrupt, make sure that the task's priority does not mask the interrupt level that it uses. Failure to coordinate the task's priority with the physical interrupts it uses can cause a system deadlock situation.

See also: Disabled interrupt levels, *System Concepts*

Note

If a task's code includes instructions that execute on a NPX, the task should not have a priority high enough to disable the interrupt level of the NPX. The highest task priority for tasks using NPX instructions is 45. Code written on a PC-bus system can be ported to a Multibus system without change. Code written on a Multibus system can be moved to a PC-bus system if the tasks that execute on a NPX have a priority of 46 or numerically higher.

Segmentation Considerations

The 32-bit interface libraries for the iRMX OS support only the compact segmentation model. This requires 32-bit application code to reside in the same code segment as the interface libraries. The best way to implement this is to structure your application as one or more compact subsystems. When porting an existing 16-bit large memory model application to a 32-bit compact memory model application, consider this:

- Compact model code runs faster than large model code. It takes 26 clocks for each segment register load. Near calls used in a compact segmentation model require no segment register loads; far calls in a large segmentation model require at least 4 register loads per call. Register loading impacts application performance quickly, especially if nested calls are made. A simple large model, 16-bit test program making recursive calls to just four system calls had a 6 percent performance boost when changed to compact.
- When moving from large to compact, insure that a valid DS value is available to jobs and tasks created by the **create_job**, **rqe_create_job**, **create_io_job**, **rqe_create_io_job**, **load_io_job**, **rqe_load_io_job**, and **create_task** system calls.

See also: Using Compact and Large Memory Models, Chapter 7,
Using the Flat Memory Model, Chapter 8

The second option follows. The EXPORTS directive causes the compiler to provide a FAR interface for the procedure task_1. This interface includes setting up DS upon procedure entry.

```
$COMPACT(my_code -CONST IN CODE- HAS my_proc;
$        EXPORTS task$1)
       ...
       ...
my_proc:
DO;
       ...
       ...
   task$1: PROCEDURE PUBLIC;
       ...
       ...
   END task$1;
END my_proc;
```

□□□

Using Compact and Large Memory Models 7

This chapter provides information on using the compact and large memory models to build iRMX applications. These guidelines apply only if you use a compiler that supports segmentation, like the Intel compilers. Understanding the following concepts will help you better understand the information presented in this chapter:

- Segmentation models
- Subsystems
- iRMX jobs, tasks, and segments

See also: Segmentation models and subsystems, *iC-386 Compiler User's Guide*, *PL/M-386 Programmer's Guide*

Choosing a Memory Model

When compiling your application source code, use compiler controls to specify the memory model for the application.

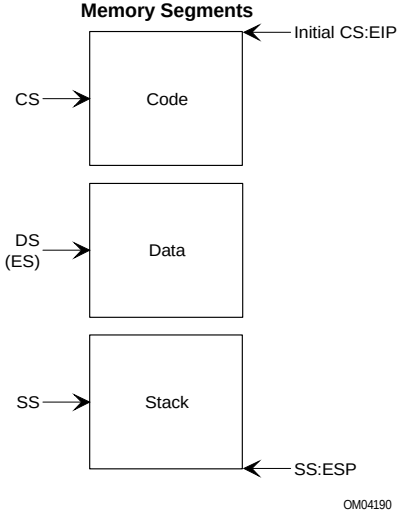


Figure 7-1. Basic Large/Compact Model Program

32-Bit Applications

For 32-bit applications, use the compact model by specifying the `compact` compiler control. If you need the efficiency and protection of multiple segments, divide your code into subsystems.

The compiler places code sections from all linked modules in the same code segment, which are addressed by the CS register. Data sections are placed into a single data segment and addressed by the DS register. Stack sections are placed into a stack segment and addressed by the SS register.

For 32-bit programming, only the compact model is allowed and there is no segment size limitation.

16-Bit Applications

For 16-bit applications, follow these guidelines when choosing a segmentation model:

- Use the compact model if your code and data can each fit into a 64 Kbyte segment.
- Use the large model if you cannot use the compact model. There are fewer size and iRMX restrictions with large, but this model results in the largest number of segment register switches.

Compile and bind your application under the compact model to determine if it fits into the compact model. If it is too large for the compact model, BND386 returns an error message. If an error message occurs, use the large segmentation model or compact subsystem.

Note

When using the Soft-Scope debugger on 16-bit, multiple stack applications, you must set the **segsize(stack(*x*))** parameter to be greater than or equal to 1024 bytes when binding the application. This is because the iRMX OS assumes stack segments which are at least 1024 bytes in length.

Code and data sections from each object module have their own code and data segments. The total size of code and data can be more than 64 Kbytes. Stack sections have a single stack segment and are addressed by the SS register. Code and data segments are paired. During program execution, both the CS and DS registers are updated whenever a public or external procedure is activated.

Porting Applications

When porting iRMX source code from a 16-bit application to 32-bit application, you must change the segmentation model if the code is not already compact. Use the compact segmentation model because the iRMX OS supports only this model for 32-bit applications.

If you use exception handlers with the compact model, use the `exports` subsystem control to export the exception handler procedures. This enables other segments to access the handler with a far call.

See also: Porting Applications, Chapter 6

If you are porting from a large/compact application to a flat application, you must use unique system calls and data types.

See also: Porting Large/Compact to Flat, Chapter 8

Using ROM and RAM Compiler Controls

If your application will be loaded into RAM, you can use the ROM or RAM controls to adjust segment sizes so that your application fits into the compact model. Specifying the ROM or RAM compiler controls determines whether the constants defined in your programs are placed in the code or the data areas. This provides additional control on the size of those segments.

For example, if your application's data is slightly larger than 64 Kbytes, specifying the ROM control (which places the constants in the code segment) might allow the remaining data to fit in a 64 Kbyte segment. This could make your code eligible for the compact model.

See also: Developing Applications for ROM, Chapter 9

Subsystems

Subsystems are very efficient for applications with multiple program modules that need to share data and communicate efficiently. You must use the compact or large models when using subsystems. A subsystem is a collection of program modules that have the same segmentation model and share the same code and data segments. For large applications, set up your application to use multiple compact subsystems.

See also: Subsystems, *iC-386 Compiler User's Guide*
or the *PL/M-386 Programmer's Guide*

Subsystem Advantages

Subsystems are efficient for these reasons:

- Code and data can be partitioned for easier maintenance.
- Segment registers are changed only when an application calls procedures or accesses data in another subsystem.
- Calls made only within a subsystem are near calls.
- Pointers referenced only within a subsystem are near pointers.
- Data is protected from being overwritten by other subsystems.
- Subsystems are useful for building loadable device drivers.

See also: Making a Driver Loadable, *Driver Programming Concepts*

Closed Subsystems

Closed subsystems have these attributes:

- The subsystem is named.
- A module list is needed.
- The `exports` control lists the functions and variables of a subsystem accessible by outside subsystems.
- Only the listed modules are combined in a closed subsystem.
- You can add or delete modules from the subsystem by changing the list of modules and regenerating the system.

The code and data segment names for a closed subsystem have the subsystem name as a prefix. For example, a 32-bit closed subsystem named `subsystem1` uses `subsystem1_code32` for the code segment and `subsystem1_data` for the data segment. The stack segment is named `stack`. In a closed subsystem, the execution stack is shared with other subsystems.

See also: Prefixes, *System Concepts*

Open Subsystems

Open subsystems have these attributes:

- The subsystem is unnamed.
- A module list is not needed.
- Segmentation controls are the only subsystem-specific compiler controls used.
- All modules using the same segmentation model are automatically combined.
- Modules can be freely added or deleted.

The code segment for an open subsystem is named `code32` for 32-bit applications. The data segment for an open subsystem is named `data` for 32-bit applications. The stack subsystem is named `stack`.

Subsystem Configurations

There can be only one open subsystem in a program, but there can be multiple closed subsystems. Every module in a program is either part of a closed subsystem or by default, part of an open subsystem. A program can consist of one of these subsystem configurations:

- Only the open subsystem, which is the default configuration
- One or more closed subsystems
- One or more closed subsystems and the open subsystem

You create a subsystem configuration when you compile and bind your application program. You specify a subsystem as closed by declaring a name for it.

See also: Subsystems, *iC-386 Compiler User's Guide* or the *PL/M-386 Programmer's Guide*

Creating a Closed Subsystem

To create a closed subsystem, create a subsystem declaration at the beginning of your source code. Specify this information:

- The `compact` compiler control (to use the compact subsystem model)
- Name of the closed subsystem
- Segment in which to place constants
- Modules that belong in the subsystem using the `has` control
- Functions that are accessible outside the subsystem using the `exports` control

The PL/M application *ramdrv.p38*, in the */rmx386/demo/plm/ldd* directory, contains this closed compact subsystem declaration:

```
$compact(ramdrv -CONST IN CODE- HAS
$         ramdrv,
$           xram;
$         EXPORTS
$                ram$init$io,
$                ram$finish$io,
$                ram$queue$io,
$                ram$cancel$io)
```

This declaration defines a closed compact subsystem named `ramdrv`. It contains the modules `ramdrv` and `xram`. The declaration exports the four procedures: `ram_init_io`, `ram_finish_io`, `ram_queue_io`, and `ram_cancel_io`. The export declaration forces the interface to these calls to be far calls. This enables other subsystems to access these procedures. This same subsystem declaration must be added to each module of the subsystem.

To generate this subsystem, use the *makefile* to compile your source code modules and bind the resulting object modules to the system. First attach to the directory where the demo resides then invoke the *makefile*.

- **af /rmx386/demo/plm/ldd <CR>**
- **make <CR>**

This section from *makefile* in the */rmx386/demo/plm/ldd* directory binds the closed subsystem:

```
ramdrv:ramdrv.obj $(LIBS) $(BND3)
   $(BND) ramdrv.obj,$(LIBLIST) &
   oj($@) pr($@.mp1) $(BNDFLAGS) &
   rn(code to $@_code32)
```

This instructs the binder to:

- Bind the RAM disk driver object module
- Bind the libraries including the loadable device driver library, the iC-386 library, the UDI interface library, and the iRMX interface library
- Use the `renameseg` instruction to remap the code segment into the `ramdrv_code32` code subsystem
- Use the `rc` instruction to allocate dynamic memory with an initial size of 5 Kbytes and a maximum size of 1 Mbyte

Creating an Open Subsystem

To create an open subsystem, create a subsystem declaration at the beginning of your source code. Specify this information:

- The `compact` compiler control (to use the compact subsystem model)
- Name of the compilation module
- Segment in which to place constants

You can optionally specify the functions that are accessible outside the subsystem using the `exports` control. Do not specify a name for the subsystem as this creates a closed subsystem.

An example of an open subsystem is not included with the iRMX OS. However, you can generate an open subsystem by modifying *ramdrv.p38*, described in the previous section. First make a copy of *ramdrv.p38* called *ramdrv.org*. This will be the original backup copy. Modify the existing *ramdrv.p38* to match this:

```
$compact(-CONST IN CODE- HAS
$         ramdrv,
$           xram;
$         EXPORTS
$                ram$init$io,
$                ram$finish$io,
$                ram$queue$io,
$                ram$cancel$io)
```

This open subsystem declaration is the same as the closed compact subsystem except the subsystem is unnamed.

To compile the modified *ramdrv.p38* file, first make a copy of *makefile* call *makefile.org*. This will be the original backup copy. Modify the existing *makefile* to match this:

```
ramdrv:ramdrv.obj $(LIBS) $(BND3)
   $(BND) ramdrv.obj,$(LIBLIST) &
   oj($@) pr($@.mp1) $(BNDFLAGS)
```

This instructs the binder to:

- Bind the RAM disk driver object module
- Bind the libraries including the loadable device driver library, the iC-386 library, the UDI interface library, and the iRMX interface library
- Use the `rc` instruction to allocate dynamic memory with an initial size of 5 Kbytes and a maximum size of 1 Mbyte

For an open subsystem, do not use the `renameseg` instruction to remap the code into the code subsystem.

□□□

8 Using the Flat Memory Model

This chapter provides information on using the flat memory model with applications for the iRMX OS. Only a small number of DOS-based compilers generate code for 32-bit segmented memory models, such as compact. Most DOS/Windows-based 32-bit compilers produce flat-model applications. The iRMX OS supports these compilers; follow the guidelines in this chapter.

See also: Memory models, *80386 Programmer's Reference Manual*

Flat Model Overview

The flat model is a 32-bit memory model where an application runs entirely in a single segment. All segment registers point to this segment. The application does not modify the segment registers. The only pointers available to the application are near (offset-only).

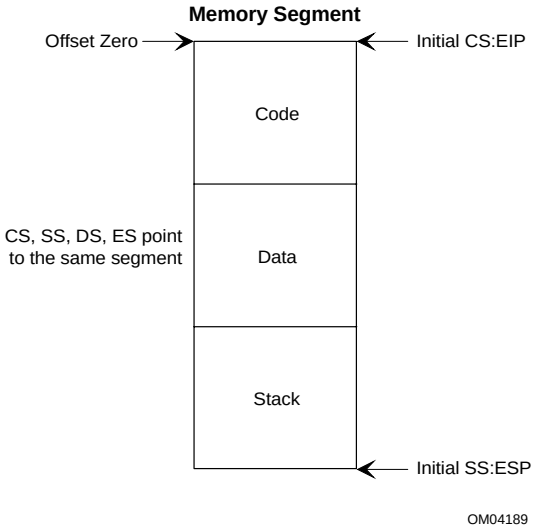


Figure 8-1. Basic Flat Model Program

Developing 32-bit flat model applications with third party tools is similar to development using the segmented third party compilers/tools (both 16- and 32-bit). The resulting flat model Microsoft Portable Executable (MPE) object model is loadable by the Application Loader. This record format is recognizable by the Soft Scope Debugger.

See also: C Compiler-specific Information, Chapter 4

Flat Model Advantages and Disadvantages

These are the advantages of using a flat model from an application point of view:

- It uses fewer iRMX objects and GDT slots since fewer segment objects are created.
- There is no need to load segment registers to de-reference pointers since all pointers are near, resulting in some performance enhancement.
- It can use common off-the-shelf 32-bit compilers.

These are the disadvantages of using a flat model from an application point of view:

- Memory allocation is less efficient since each distinct area of the application — code, data, and stack — must be a minimum of 4 Kbytes, and must be a multiple of 4 Kbytes.
- Enabling paging in the microprocessor degrades system-wide performance by approximately 4%.
- There is less protection between the code, data and stack areas of an application.

Executing Flat Model Applications on iRMX

You can load and run a flat model application on the iRMX OS through the services of the paging subsystem, flat model support code, and the Application Loader. Flat model applications run in protection ring three of the microprocessor.

The paging subsystem provides an environment in which a flat model application can dynamically add physical memory to or free physical memory from its own address space.

The Application Loader recognizes a flat model application in MPE format, creates a flat model environment for it, and loads the application into this environment. Once loaded, control is passed to the flat model application.

Using Flat Model With Paging Support

Paging support for flat model in iRMX means turning on the paging mode of the processor but not implementing demand paging. Demand paging can interfere with the running of a real-time OS because it swaps pages from memory to disk and back. The iRMX OS uses paging for virtual address translation only. When a flat model application is running, a page fault is equivalent to a general protection fault. This provides the processor-based protection that you would normally lose by not using segmentation.

With paging support, the flat model application resides in an iRMX "virtual segment," which resides in part of a virtual memory space of 4 Gbytes. Physical memory is only assigned to areas of the virtual segment that require it, such as the code, data, stack, and any dynamic storage requested while the application is running:

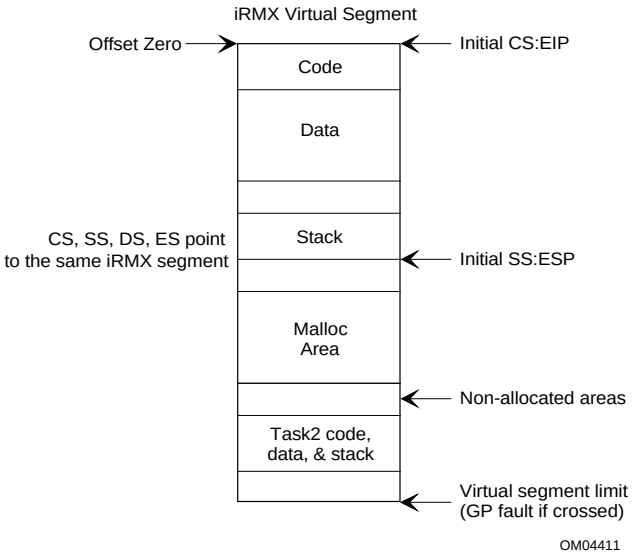


Figure 8-2. Flat Application Program on iRMX with Paging

Paging Subsystem

The paging subsystem is an extension of the iRMX Nucleus and provides the necessary paging support for flat model applications. It is available as a first-level or a loadable job.

You can configure the paging subsystem into the OS with the ICU, or load it with the **sysload** command. This subsystem is small, using less than 14 Kbytes of code and data.

The Paging Job

You can load the paging job, *paging.job*, at any time iRMX is running. This job contains the entire paging subsystem. Once loaded, the OS part of memory is identity-mapped, paging is enabled, and the **rqv_** system calls become available. To load the job, type:

```
- sysload /rmx386/jobs/paging.job [block1, block2,...block8]
```

where:

blockn consists of *memory_start*, *memory_end*

The *block* parameter defines a block of physical memory that is outside the range of physical memory managed by the Nucleus Free Space Manager (FSM). The paging subsystem identity maps all physical memory known to the FSM. If there are blocks of memory that are not known to the FSM, you should specify these so that they can be identity-mapped as well. You can define up to eight memory blocks, however, these memory blocks should not overlap. A memory block that overlaps with a previously-defined block is ignored.

The *memory_start* and the *memory_end* parameters represent the start and the end addresses of the physical memory block, respectively. The start address is rounded up to the next 4 Kbyte boundary. The end address is rounded up to the next 4 Kbyte boundary minus one. These addresses must be hexadecimal and do not need the “H” (hexadecimal) suffix.

Note

Any physical memory that is not known to either the Free Space Manager (from the ICU configuration) or the paging subsystem is not accessible from your application once paging is enabled.

Errors and initialization messages are reported to the *:config:paging.log* file. Initialization messages include the identity memory map created by the paging subsystem. Check the log file to verify that the actual physical memory has been identity-mapped correctly.

Identity Mapping

The paging subsystem identity-maps all physical memory known to the Free Space Manager. This includes memory which is configured in the ICU as a first-level job or which is added from using the **sysload** command. Identity mapping helps protect dedicated memory, such as that found on dual port memory for a custom device driver, from being over-written.

See also: MEMF, PIMM Commands, *ICU User's Guide and Quick Reference*

Flat Model Support Code

The flat model support code provides the flat-to-segmented pointer conversion libraries required to allow flat applications to make iRMX system calls and C library calls.

The flat model support code is a configurable part of the operating system. This code may be loaded via the **sysload** command. This subsystem consists of approximately 20 Kbytes of code and data.

Conversion of Flat Model Pointers in System Calls

In a flat model application, all pointers are near (offset-only) pointers. The iRMX OS requires all pointer parameters in system calls to be far pointers. Therefore, all near flat model pointers must be converted to far pointers before entering the OS itself. The *flat.job* automatically performs the conversion for each system call made by your application.

This job contains the entire flat model support code and requires the paging subsystem. Flat model applications can make iRMX system calls and C library calls once *flat.job* is loaded. To load the job, type:

```
- sysload /rmx386/jobs/flat.job <CR>
```

Errors and initialization messages are reported to the *:config:flat.log* file.

The Flat Model Job

You can load the flat model job, *flat.job*, at any time the iRMX OS is running. There are no command line options for *flat.job*.

Note

You cannot use the ICU to configure the flat memory model as a first-level flat job.

You cannot configure flat model applications as first-level jobs but you can configure them as loadable jobs.

Execution Model

The Application Loader recognizes a flat model MPE program and creates a flat environment for the program using the paging subsystem (it must be loaded or configured into the system). After the program is loaded into the flat environment, a job gets created for the loaded code the same as it does for segmented programs.

Figure 8-3 shows the loading and execution flow of a flat model program.

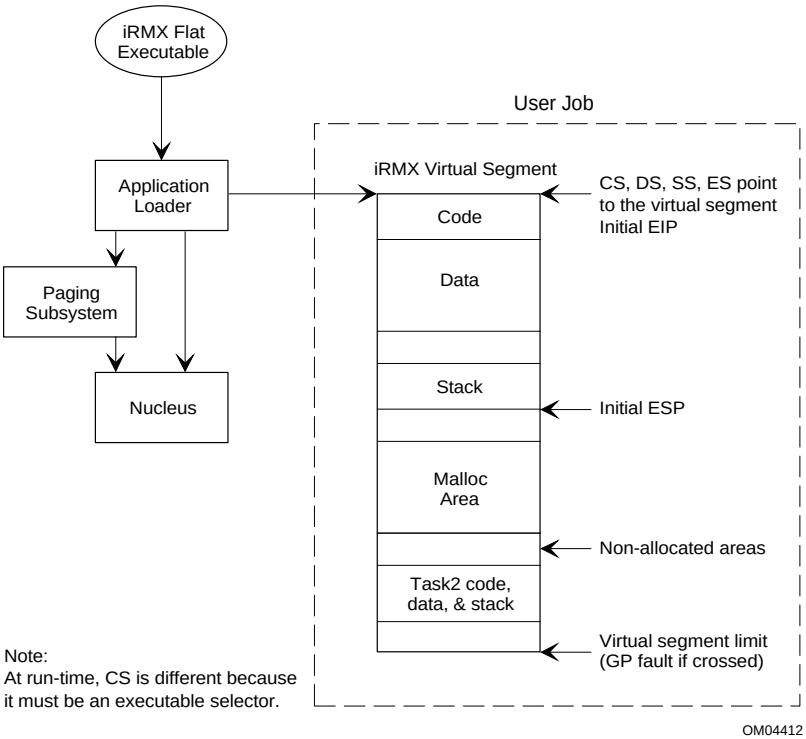


Figure 8-3. Execution of a Flat Model Program on iRMX

System Calls

The following is a list of new system calls required to manage virtual segments and provide other flat model support.

Since most flat model compilers do not support far pointers (or support "based" variables), they cannot access normal iRMX segments. Instead, several system calls are provided to either access iRMX segments, or eliminate the need for them entirely.

See also: *System Call Reference*

Virtual Memory	Nucleus	Basic I/O System
rqv_create_segment	rq_move_data	rq_wait_iors
rqv_allocate	rq_get_buffer_limit	
rqv_allocate_at	rq_validate_buffer	
rqv_free		
rqv_change_access		
rqv_map_physical		

Existing System Calls

These existing calls have been changed slightly for paging support. In all cases, the changes add functionality to work with the new virtual segments. You can continue to use these calls from segmented applications.

- rq_delete_segment
- rqe_get_address
- rq_get_size

Using the Flat Model System Calls

When developing a flat model application, be aware of these unique issues, which are not a concern if you are developing a segmented application:

- Virtual memory and the corresponding allocation and de-allocation of physical memory
- Use of iRMX segments by a flat model application

Virtual Memory

New system calls provide two levels of access to the paging mechanism. The **rqv_allocate_at** system call provides low-level access. The Application Loader, as well as other system utilities, use this system call to gain direct access to a virtual segment. Using this call enables an application to place the code, data, stack, and other segments into a unique location in the virtual segment specified by the object module being loaded.

The **rqv_allocate_at** system call provides high-level access. This allocation system call provides management of the virtual address space within a virtual segment. The call is meant to be used by applications and any other free space manager, such as **malloc** and **sbrk**. It allocates physical memory, places it within an available area of the virtual segment, and then returns a near pointer to the allocated memory. For the flat model application, this system call is preferred over **rq_create_segment**, since the latter returns a token which is not accessible using the flat memory model.

The memory required for page tables is charged to the calling job's memory pool. The first allocation to a virtual segment will incur a 4 Kbyte overhead for a page table. You should compute job memory pools with this page table overhead in mind.

Porting Compact/Large to Flat

If you need to access iRMX segments, use one of these mechanisms:

- The **rqv_allocate** system call replaces the **rq_create_segment** call in flat model applications. It allocates physical memory to the application's virtual segment with no additional objects or slots being consumed.

To share this memory with another task, pass a near pointer through a data mailbox if the other task is in the same virtual segment (job). Another method is to create a descriptor around the allocated memory and pass the token for the descriptor passed through a normal mailbox.

- The **rq_wait_iors** BIOS system call replaces either **rq_receive_message** or **rq_wait_io** after an I/O call. This call returns the asynchronous IORS into a buffer in the caller's address space, instead of in an iRMX segment.

Debugging Support

The System Debugger (SDB) understands and displays flat model versions of the iRMX system calls. The debugging procedures are similar to those used for compact and large model applications. However, with flat model applications, the stack parameters are reversed. Take this into account when viewing the stack using the **vs** or **vu** SDB commands.

See also: **vs**, **vu** commands, *System Debugger Reference*

❑❑❑

Developing Applications for ROM 9

Using the iRMX III OS, you can create ROM-based iRMX applications. Configuring a ROM-based system has several benefits. You can write-protect your stable code, load your system quicker than a RAM-based system, and incur lower costs than with a RAM-based system.

> **Note**
> You can only create ROM-based applications under the iRMX III OS. You cannot use the DOSRMX or iRMX for PCs OS.

This chapter contains information on:

- Testing your application from RAM
- Calculating size and location parameters
- Programming your application into ROM
- Creating an example ROM application

You may need to refer to one or more of these manuals:

- *ASM386 Macro Assembler Operating Instructions/ASM386 Assembly Language Reference*
- *iC-386 Compiler User's Guide*
- *C Library Reference*
- *ICU User's Guide and Quick Reference*
- *Intel386 Family Utilities*
- *PL/M-386 Programmer's Guide*
- *System Debugger Reference*

Testing a System

The normal development cycle is to load your system using the bootstrap loader, test it, correct any errors, and then reassemble or recompile any appropriate program code. Next, you must regenerate your system and load the system again. Continue this procedure until you have created a functional target system.

Once you have created your final system, fine-tune the memory allocated for the system by editing the MEMS and MEMF screens in the Interactive Configuration Utility (ICU). If your target system will reside in ROM, enable the ROM feature by entering "Yes" to the "System in ROM" entry on the ROM screen of the ICU. You must also make any necessary changes to the ROM screen.

See also: Setting the Memory Address and Size Values, in this chapter

Loading an Application into ROM

When you place an iRMX application system in EPROM/FLASH, a number of hardware assumptions are made by the iRMX initialization code regarding memory layout. These assumptions are:

- The entire iRMX application system image (minus the ROM Initialization Code) is in a contiguous section of memory described by a single entry on the MEMS screen of the ICU definition file.
- The ROM Initialization Code must reside within 64 Kbytes of the top of ROM/FLASH memory and on a 4 Kbyte boundary.
- Volatile System Memory (system RAM) must reside within the first megabyte of memory, below and directly adjacent to Free Space Memory.
- The first section of the Free Space Manager, defined on the MEMF screen of the ICU, must be large enough to contain those parts of the application system that are copied from ROM to RAM.

Preparing an Application to Reside in ROM

You can configure a ROM-based iRMX application as a first-level job. This job often contains a single initialization task that creates or starts the creation of all other objects required by the first-level job.

The root task creates the first-level jobs. Each time the root task creates a first-level job, the root task suspends itself to allow the new job's initialization task to perform synchronous initialization.

The root task creates first-level jobs using this programming loop:

```
Repeat for each first-level job
       Create first-level job
       Suspend root task (until resumed by a
       first-level job finishing its initialization)
Until finished
End
```

Synchronous initialization consists of functions that must be performed before some other first-level job is created. Typically, this requires creating objects or making resources available that subsequent tasks will use. For example, the initialization task in the EIOS job must ensure that the EIOS is ready before it can allow the root task to create other first-level jobs that would use EIOS functions.

When the initialization task finishes its synchronous initialization, it must inform the root task that it is finished so the task can resume execution and create another first-level job. The initialization task must always inform the root task that it has completed its synchronous initialization process by calling the **rq_end_init_task** system call. This call requires no parameters and causes the root task to resume execution and create the next first-level job.

Note

You must include the **rq_end_init_task** system call in the initialization task of each of your first-level jobs even if they do not require synchronous initialization; otherwise the root task remains suspended.

The amount of synchronous initialization depends on your job structure. You must determine how the pieces of your system interact and how they must synchronize.

Another important factor in initialization is the order in which the root job creates first-level jobs. Shown below is an example order. The order the root task uses to create first-level jobs depends on where the jobs are started in relation certain OS layers. This ordering depends what parameters you specify with the ICU, not on the priority of the tasks.

Order	Root Job	First-Level Job	I/O User Job
1	Root Job		
2		System Debugger	
3		Basic I/O System	
4		Extended I/O System	
5			I/O User Jobs
6		User Jobs	
7		Human Interface	
8		Shared C Library	

See also: Help message for the (SEQ) and (TPUJ) ICU screens, Interactive Configuration Utility

Methodology for Burning an Application into ROM

When burning an application into ROM, your ROM/Flash programmer should be capable of handling OMF386 or Intel hex format code. The procedure is:

1. Identify which format your ROM/Flash programmer takes.
2. The builder generates the OMF386 output file. This file is specified in the ROF entry of the ICU GEN screen. Load the code directly into the ROM/Flash programmer, splitting the code between multiple devices if necessary.
3. If your ROM/Flash programmer requires hexadecimal format, use the OH386 utility to convert the OMF386 code to OH386 code.

Both OMF386 and hex format contain both code and data. The presence of data in the input file to the ROM/Flash programmer may cause a warning, which you can ignore.

Use your Flash/ROM programmer to extract code only within the address range that will be placed in ROM.

Developing a ROM-based Application System

When developing a ROM-based application, you should develop as much of the application as possible to be a program loadable under the Human Interface CLI. Remove all the bugs possible in the loadable version of the job. Use the Soft-Scope debugger and other iRMX tools to help debug your system.

In case the target hardware does not support a full-featured iRMX environment with a Human Interface, you can write intelligent stubs that simulate the target hardware. Then run both the application and its hardware-simulating stubs in a loadable iRMX environment. This allows you to complete as much of the debugging as possible with a loadable job instead of a ROM-based job.

Once your application is ready for ROM/FLASH on the target hardware, you must use the ICU to configure the iRMX application system containing your application.

Start with the Intel-provided ICU definition file that most closely fits your target hardware. These files are located in the */rmx386/icu* directory.

If you do not find the appropriate file, you can specify a new definition file using the ICU. Once in the ICU, you must make modifications to the various layer/hardware screens until your target hardware and software environment are fully described.

See also: Example ICU Session, *ICU User's Guide and Quick Reference*

Overview of the ROM-based Application Example

The following example illustrates how a ROM-based application system is generated. The example describes the instructions for generating the example MIX486 ROM application located in the */rmx386/demo/rom/mix4demo* directory. The application system defined by this example has these attributes:

- Runs on a MIX486 board (MIX486DX33, MIX486DX66, or MIX486SX33) in a Multibus II backplane
- Loads out of FLASH into RAM and executes out of RAM
- Contains a simple Multibus II message passing program that waits at a specific port for Multibus II messages and replies to them

First, develop the application as an Human Interface-loaded program. This program, *receive.c*, does the message passing. After you make any changes to *receive.c* and it is fully debugged, the following procedure converts it to a first-level job:

1. Add a call to **rq_end_init_task** to the program's initial task after completing any required synchronous initialization. You can leave the **rq_end_init_task** call in even if you run the demo application from the Human Interface.
2. Convert the program's initial task to a public procedure (already set up as *main* in C programs).
3. Modify the bind process to produce a linkable version of the program instead of the Single Task Loadable (STL) version.
4. Modify the bind process to suppress all Public symbols except the name of the program's initial task and the name of one of the program's public variables.

Once the application program is ready as a first-level job, the next step is to configure the iRMX OS to run on the target hardware.

Generating the ROM-based Application Example

The files used to generate the example ROM application are in the */rmx386/demo/rom/mix4demo* directory. These files are:

receive.c	Receives a message from *sendmb2* and returns a new message
sendmb2.c	Sends a message to a port on a MB II agent running *receive*
makefile	File used to generate the example

To generate the example:

1. Change the directory to */rmx386/demo/rom/mix4demo*.
2. At the iRMX prompt, type: **`make <CR>`**

This creates the Human Interface programs *receive* and *sendmb2*, and the user job module, *receive.lnk*.

Configuring the iRMX OS

You must configure the iRMX OS through the ICU to recognize that the target hardware is a MIX486 board.

> **Note**
> In the following ICU screens, enter the values listed in **`bold`**. These values are specific to the example application and should not be changed.

Setting the Hardware Values

In the following HARD screen, the hardware addresses are specific to the MIX486 board. Because the application does not need a finer time granularity than 10 milliseconds, set the KTR entry to 1. Specify "Yes" for the EMU entry so the system includes an NPX Emulator. This Emulator is dormant if a math coprocessor is present (MIX486DX33 or MIX486DX66 board) but provides numeric support when no math coprocessor is present (MIX486SX33 board).

```
(HARD)      Hardware
(BUS) System Bus Type   [1=MBI / 2=MBII / 3=AT]  2
(TP)  8254 Timer Port   [0-0FFFFH]  0D0H
(CIL) Clock Interrupt Level   [0-7]  0
(CN)  Timer Counter Number   [0,1,2]  0
(CIN) Clock Interval   [0-65535  msec]  10
(KTR) Kernel Tick Ratio   [1-65535]  1
(CF)  Clock Frequency   [0-65535  khz]  1250
(TPS) Timer Port Separation   [0-0FFH]  02H
(EMU) Emulate Numeric Processor   [Yes/No]  YES
(IF)  Initialize On-board Functions   [0=No / 1-0FFH]  08H
(BIP) Board Initialization Procedure    [1-45 Chars]
```

Setting the Multibus II Addresses and Port Separation Values

In the following Multibus II screen, the Multibus II hardware addresses and port separations are specific to MIX486 boards. The application uses only aligned buffers so no message passing transfer/alignment buffers are included.

```
(MBII)       Multibus II Hardware
(MDP) Message Device Base Port Address   [0-0FFFFH]  0H
(MDS) Message Device Port Separation   [0-0FFH]  04H
(MDL) Message Interrupt Level   [Encoded Level]  04H
(MCO) Message Device Duty Cycle for One Cycle DMA   [0-0FFH]  052H
(MCT) Message Device Duty Cycle for Two Cycle DMA   [0-0FFH]  097H
(MDC) Message Device Duty Cycle for Burst DMA   [0-0FFH]  04AH
(DDP) Message Device ADMA Data Port   [0-0FFFFH]  0H
(GBR) ADMA Burst Register   [0-0FFFFH]  0H
(GDR) ADMA Delay Register   [0-0FFFFH]  0H
(AIB) ADMA Base Port Address   [0-0FFFFH]  0200H
(ACI) ADMA Channel for Input   [0-0FFFFH]  02H
(ACO) ADMA Channel for Output   [0-0FFFFH]  03H
(DIB) DMA Input Buffer Size   [0-0FFFFFFFFH]  0H
(DOB) DMA Output Buffer Size   [0-0FFFFFFFFH]  0H
(DDA) DAG Device Used   [Yes/No]  YES
(DBA) DAG Base Port   [0-0FFFFH]  0300H
(WDP) Watchdog Present   [Yes/No]  NO
(WDM) Watchdog Mboxes   [0-0FFH]  03H
(WDI) Watchdog Transmission Interval   [1-0FFFFFFFFH]  03E8H
(WDT) Watchdog Timeout   [1-0FFFFFFFFH]  03E8H
```

Setting the Master and Slave Interrupt Values

In the following INT and SLAVE screens, the Master and Slave Interrupt layout is specific to the MIX486 board.

```
(INT)       Interrupts
(MP)  8259A Master Port   [0-0FFFFH]  0C0H
(MPS) Master PIC Port Separation   [0-0FFH]  02H
(IS)  Interrupt Slaves   [Yes/No]  YES

(SLAVE)      Slave Interrupt Levels
Slave = Slave_number, Level_Sensitive,    Port,    Separation
             [0-7]        [Yes/No]     [0-0FFFFH]   [0-0FFH]
[ 1]  Slave =   7 ,            NO ,         0C4H  ,     02H
[ 2]  Slave =
```

Setting the Subsystem Values

In the following SUB screen, include the System Debug Monitor and System Debugger subsystems only as an aid to debugging. Remove these when configuring the production system.

The application does not require the services of other subsystems because those provided by the Kernel, Nucleus, and Message Passing subsystem meet the application's needs.

```
(SUB) Subsystems
(UDI) Universal Development Interface   [Yes/No]  NO
(CLB) Shared C Library   [Yes/No]  NO
(HI)  Human Interface   [Yes/No]  NO
(AL)  Application Loader   [Yes/No]  NO
(NET) Networking   [Yes/No]  NO
(EIO) Extended I/O System    [Yes/No]  NO
(BIO) Basic I/O System   [Yes/No]  NO
(PGS) Paging Subsystem   [Yes/No]  NO
(VMD) VM86 Dispatcher   [Yes/No]  NO
(SDM) System Debug Monitor   [Yes/No]  REQ
(SDB) System Debugger   [Yes/No]  YES
(OE)  OS Extension   [Yes/No]  NO
```

Setting the Memory Address and Size Values

In the following MEMS and MEMF screens, change the memory parameters to reflect a ROM-based application.

```
(MEMS)       Memory for System
      SYS = low [0-0FFFFFFFFH], high [0-0FFFFFFFFH]
[ 1]  SYS =        0FFF80000H,          0FFFFFFEFH
[ 2]  SYS =

(MEMF)       Memory for Free Space Manager
    FSM = low [0-0FFFFFFFFH], high [0-0FFFFFFFFH]
[ 1]  FSM =         020000H   ,          09FFFFH
[ 2]  FSM =         0C0000H   ,          07FFFFH
[ 3]  FSM =
```

In a ROM/FLASH-based system, the MEMS entry reflects the physical address of the ROM/FLASH devices once the system is switched to Protected Virtual Address Mode. It is assumed to be contiguous, in other words, it is all defined in a single SYS entry.

On some boards, the ROM/FLASH is at a different address on reset and then is switched to its final location through I/O output operations. On the MIX486 board, this address range is fixed and encompasses the two 2 Mbit FLASH sites on the board.

Note

If you adjust the physical address of ROM/FLASH during the system initialization process, you must do it in-line in the **custom_initial_hw_setup** subroutine. No jumps or calls are allowed.

See also: Debugging the ROM Initialization Process, in this chapter

The FSM sections of the MEMF screen describe the RAM Memory available to the Free Space Manager. The space in memory between 9FFFFH and 0C0000H is required by the MIX486 board due to its use of a PC chipset. In a ROM/FLASH-based system, the first FSM section must provide enough RAM storage for system objects copied from ROM/FLASH to RAM during the system initialization process. Items that are copied from ROM to RAM are the system GDT, LDT, IDT and four TSSs. Calculate the minimum size for the first FSM section of memory as:

```
Size(FSM(0)0 = ((Final GDT size * 8) * 2) +
                Final IDT size * 8) + 200H
```

In cases where the application system executes out of RAM, the first FSM memory section must be large enough to contain the minimum FSM size, calculated above, in addition to the memory required to hold all code segments that make up the application system. Refer to the Segment Map (Figure 9-1) portion of the *.mp2* file generated by BLD386 for the application system and add up the segment sizes for all "ER" type segments listed there.

The final sum of the equation above plus the application code segments is the final minimum size of the FSM(0) section of memory.

When the system initializes (during the ROM Initialization Code and the early stages of Nucleus initialization), it removes memory from the FSM(0) memory section (beginning at the lowest specified memory address) as needed to handle the items copied from ROM to RAM. FSM(0)'s final low address is adjusted upwards accordingly.

Figure 9-1 lists the Segment Map from the *mix4dxro.mp2* file.

```
SEGMENT MAP
TABLE  BIT  DPL  ACCESS  USE  BASE  LIMIT  SEGMENT NAME

GDT
1   1  0  RW  16 FFF80000H  00000DBFH  GDT:
2   1  0  RW  16 FFF80DC0H  0000008FH  IDT:
33  1  0  RW  16 0000FAA8H  00000003H ?DUMMY_MODULE.SDM3_ALIAS_SEGMENT3
34  1  0  ER  16 FFFA8754H  0000296EH  SDM_DASM.DASM_CODE
35  1  0  RW  16 FFFA0BB0H  000013E5H  SDM_DASM.DASM_DATA
44  1  0  ER  16 FFFAB0C4H  00000015H  SDM_DASM.CODE
45  1  0  ER  32 FFFB1AD8H  0000BFC7H  M3.SDMIII_CODE32
46  1  0  RW  32 0000A3C4H  00000860H  M3.SDMIII_DATA
47  1  0  RW  16 0000FAA0H  00000003H ?DUMMY_MODULE.SDM3_ALIAS_SEGMENT
48  1  0  RW  16 0000FAA4H  00000003H ?DUMMY_MODULE.SDM3_ALIAS_SEGMENT2
49  1  0  RW  16 FFFA0AE2H  000000CCH SDM_DASM.SDM_DASM_DATA
60  1  0  RW  32 00000000H  000060CAH  NUCDAT.DATA
80  1  0  ER  32 FFF80E50H  0001FC90H  NUCDAT.CODE
85  1  0  RW  32 00006464H  000003FFH  NUCDAT.STACK
300 1  0  RW  16 0000FA18H  00000087H ?DUMMY_MODULE.SHADOW_IDT_SEG
302 1  0  RW  32 0000FB8CH  00000007H ?DUMMY_MODULE.CC_120_SEG_5
308 1  0  ER  32 FFFC0A24H  00015AE5H  SDBCNF.CODE
309 1  0  RW  32 0000B958H  00001064H  SDBCNF.DATA
310 1  0  RW  32 0000B2FCH  00000659H  SDBCNF.NEWSTACK
320 1  0  ER  32 FFFA3560H  00000139H  NTRSTK.STK_OVFW
343 1  0  ER  32 FFFBEAC8H  00001F5AH  M3.CC_CODE32
344 1  0  RW  32 0000AC28H  000006D3H  M3.CC_DATA
400 1  0  RW  16 0000FA10H  00000003H ?DUMMY_MODULE.MI_ALIAS_SEGMENT
401 1  0  RW  16 0000FA14H  00000003H ?DUMMY_MODULE.MI_ALIAS_SEGMENT2
402 1  0  ER  32 FFFAB0DCH  000069FBH  M3.MIII_CODE32
403 1  0  RW  32 00007270H  00001152H  M3.MIII_DATA
404 1  0  RW  16 000083C4H  00001FFFH  M3.STACK
426 1  0  ER  16 FFFA85B2H  000001A1H  SDM_DASM.SDM_DASM_CODE

LDT.1 (LDT1)
1   1  0  RW  16 FFFA1F96H  00000DBFH  LDT1:
72  1  0  ER  32 FFFA369CH  00004F15H  E80387.A?MED
73  1  0  RW  32 00006C84H  000001DAH  E80387.A?MSR
74  1  0  RW  32 00006E60H  0000028FH  E80387.STACK
75  1  0  RW  32 000060CCH  00000394H  NUCDAT.JOBDAT
```

Figure 9-1. Example Segment Map

```
79  1  0  RW  32 00006864H  0000001DH  NUCDAT.ESCAPE_SS
80  1  0  ER  32 FFFA3538H  00000026H  NUCDAT.ENTRY_CODE
90  1  0  ER  16 FFFFFFF0H  00000003H  NUCDAT.RESTART_CODE_ROM
91  1  0  ER  16 FFFFF000H  00000DCEH  NUCDAT.CODE_ROM
92  1  0  RW  32 00006884H  000003FFH  NTRSTK.SE_STACK
93  1  0  RWD 16 FFFF7198H  0000FF57H  SDM_DASM.STACK
94  1  0  RW  32 00007198H  000000D4H  M3.DATA
95  1  0  ER  16 FFFBDAA0H  00001024H  M3.SDMIII_NPX_CODE
96  1  0  ER  32 FFFD650CH  0000050FH  START.CODE
97  1  0  RW  32 0000C9C0H  0000004CH  START.DATA
98  1  0  RWD 32 0000FA10H  FFFFCFFFH  START.STACK
99  1  0  RW  32 0000FAACH  0000007FH ?DUMMY_MODULE.CC_120_SEG_1
100 1  0  RW  32 0000FB2CH  0000001FH ?DUMMY_MODULE.CC_120_SEG_2
101 1  0  RW  32 0000FB4CH  0000001FH ?DUMMY_MODULE.CC_120_SEG_3
102 1  0  RW  32 0000FB6CH  0000001FH ?DUMMY_MODULE.CC_120_SEG_4
```

Figure 9-1. Example Segment Map (continued)

Setting the System Debug Values

In the following SDB screen, the System Debugger is entered through a Non-Maskable Interrupt (NMI) generated across the interconnect space. Set the SLV entry to 0FFH and set the NMI entry on the NUC screen to allow an NMI to trigger the SDB.

```
(SDB)        System Debugger
(SLV) SDB Interrupt Level   [Encoded Level/NONE = 0FFH]  0FFH
(ESC) Enable Screen Scrolling Control   [Yes/No]  YES
```

Since the MIX486 board has no on-board serial devices, set the RCI entry to Primary in the SDM screen so the Remote Console Interface Driver is the SDM/SDB's I/O device.

```
(SDM)        System Debug Console MultiBus Drivers
(D51) 8251 Console Controller Driver  [Primary/Secondary/No]  NO
(A54) 354 Port A Console Controller Driver  [Primary/Secondary/No]  NO
(B54) 354 Port B Console Controller Driver  [Primary/Secondary/No]  NO
(A74) 8274 Port A Console Controller Driver  [Primary/Secondary/No]  NO
(B74) 8274 Port B Console Controller Driver  [Primary/Secondary/No]  NO
(G79) SBX 279 Console Controller Driver  [Primary/Secondary/No] NO
(A30) 82530 Port A Console Controller Driver  [Primary/Secondary/No] NO
(B30) 82530 Port B Console Controller Driver  [Primary/Secondary/No] NO
(RCI) Remote Console Interface Driver  [Primary/Secondary/No]  PRIMARY
```

```
PC Drivers
(SR1) Serial Port One   [Primary/Secondary/No]  NO
(BP1) Serial Port One Base Address   [0-0FFFFH]  03F8H
(SR2) Serial Port Two   [Primary/Secondary/No]  NO
(BP2) Serial Port Two Base Address   [0-0FFFFH]  02F8H
(CON) Console Port   [Primary/Secondary/No]  NO
```

With the SDM/SDB present in the configuration, set the Default Hardware Exception Handler and NMI Exception Mode entries in the NUC screen to enable an NMI signal to break to the monitor.

```
(NUC)        Nucleus
(NGE) Number Of GDT Entries   [440-8190]  500
(NIE) Number Of IDT Entries   [0-256]  256
(PV)  Parameter Validation   [Yes/No]  YES
(ROD) Root Object Directory Size   [0-3840]  50
(DSH) Default Software Exception Handler   [Job/Task/STask/User]  JOB
(EM)  Exception Mode   [Never/Program/Environ/All]  NEVER
(NEH) Name of Ex Handler Object Module   [1-55 Chars]
(DHH) Default Hrdwr Exception Handler [Job/Task/STask/Monitor] MONITOR
(NMI) NMI Exception Mode   [Ignore/Process]  PROCESS
(NEB) NMI Enable Byte   [0-255]  04H
(LSE) Low GDT/LDT Slot Excluded from FSM  [440-8189/NONE=0]  0
(HSE) High GDT/LDT Slot Excluded from FSM [440-8189/NONE=0]  0
(RRP) Round Robin Priority Threshold   [0-255]  140
(RRT) Round Robin Time Quota   [0-255]  5
(RIE) Report Initialization Errors   [Yes/No]  YES
(MCE) Maximum Data Chain Elements   [0-0FFFFH]  080H
(CS)  Nucleus Communication Service   [Yes/No]  YES
```

Setting the Nucleus Communications Values

In the NCOM screen, set the Nucleus Communications Services entries to standard values.

```
(NCOM)       Nucleus Communication Service
(PMT) Message Task Priority   [0-255]  128
(PDT) Deletion Task Priority   [0-255]  128
(DPT) Default Number of Port Transactions   [0-255]  16
(DHI) Default Host ID   [0=None/1-254]  0
(VBP) Validate Buffer Parameters   [Yes/No]  YES
(MST) Max No. of Simultaneous Transactions [0-0FFFFH]  040H
(MSM) Max No. of Simultaneous Messages   [0-0FFFFH]  080H
(RFT) Receive Fragment Failsafe Timeout   [0-0FFFFH]  0400H
(NTM) Number of Trace Messages   [0-255]  255
```

Setting the System Job Values

In the SYSJ screen, no system jobs are required in this application system so set all entries to “No.”

```
(SYSJ)       System Jobs
(PCI) PCI Server Job   [Yes/No]  NO
(DL)  MBII Downloader Job   [Yes/No]  NO
(ATC) ATCS/279/ARC Server Job   [Yes/No]  NO
(A50) ATCS/450 Server Job   [Yes/No]  NO
(BS)  MSA BootServer Job   [Yes/No]  NO
(FPI) FPI Server Job   [Yes/No]  NO
(SSK) SoftScope Kernel Job   [Yes/No]  NO
```

Setting the User Job Values

Set the TP entry in the USERJ screen so the priority of the first-level job, *receive*, starts at 155. Even though the job starts after the EIOS, it has no effect since there is no BIOS or EIOS in the system. Therefore, the job starts immediately after the SDB initialization job.

Since the job is written in C, the initial task's public name is *main*. Because it is coded as a far procedure, no Public Variable Name is required for the VAR entry of the USERJ screen. The initial task sets up its own data segment.

```
(USERJ)      User Jobs
(NAM) Job Name   [0-14 Chars]  RECEIVE
(SEQ) Job Sequence   [Before/After]  AFTER
(ODS) Object Directory Size   [0-3840]  10
(PMI) Pool Minimum   [20H-0FFFFFFFH]  010000H
(PMA) Pool Maximum   [20H-0FFFFFFFH]  0FFFFFH
(MOB) Maximum Objects   [1-0FFFFH]  0FFFFH
(MTK) Maximum Tasks   [1-0FFFFH]  0FFFFH
(MPR) Maximum Priority   [0-255]  129
(EHS) Exception Handler Entry Point   [1-31 Chars]

(EM)  Exception Mode   [Never/Prog/Environ/All]  NEVER
(PV)  Parameter Validation   [Yes/No]  YES
(TP)  Task Priority   [0-255]  155
(TSA) Task Entry Point   [1-31 Chars]  MAIN

(VAR) Public Variable Name   [0-31 Chars]

(SSA) Stack Segment Address   [SS:SP]  0000:0000H
(SSI) Stack Size   [0-0FFFFH]  0500H
(NPX) Numeric Processor Ext. Used   [Yes/No]   NO
```

In the USERM screen, the builder looks for the first-level job link file, *receive.lnk*, in the local directory.

```
(USERM)      User Modules
      Module = 1-55 characters
[ 1]  Module = RECEIVE.LNK
[ 2]  Module =
```

Setting the RAM and ROM Values

```
(ROM)        ROM Code
(SYR) System In ROM   [Yes/No]  YES
(CPI) Copy ROM Initialization Code to RAM   [Yes/No]  NO
(EOR) Execute System Out of Rom/Flash   Yes/No]  NO
(VSS) Volatile System Memory Starting Address   [0-0FFFFFFFH]  0H
(VSE) Volatile System Memory Ending Address   [0-0FFFFFFFH]  01FFFFH
(RBA) Base Address of ROM Init code at reset  [0-0FFFFFFFH] 0FFFFF000H
(RDA) RAM Destination Address of ROM Init code  [0-0FFFFFFFH]  0H
(SRC) Size of ROM Initialization Code   [0-0FFFFFFFH]  0H
(CRS) Custom ROM Initialization Source File   [1-45 Chars]  MIXINIT.INC
(CRO) Custom ROM Initialization Object File   [1-45 Chars]  MIX4IN.LNK
```

In this application system, the ROM Initialization Code mode of operation is RAM-less (CPI=NO). In the RAM-less mode, the ROM Initialization Code expects to be entered using a near jump placed at the Reset Vector (at FFFFFFF0H). In this case, the ROM Initialization Code immediately sets up its initial GDT/IDT in nonvolatile memory before switching the microprocessor into protected mode.

Setting the ROM Initialization Code mode of operation to RAM-full (CPI=YES) means that the ROM Initialization Code expects to be entered using a far jump from some non-iRMX initial program, such as a Flash utility. In this case, the ROM Initialization Code copies itself from nonvolatile memory into RAM and sets up its initial GDT/IDT in RAM before switching the microprocessor into protected mode. This mode allows nonvolatile memory to be remapped to a new physical address. The RAM destination address of the ROM Initialization Code (RDA) must be within the first megabyte.

See also: Calculating Volatile Memory Size, in this chapter

The system copies the OS and its associated application from ROM to RAM as part of the initialization process (EOR=NO). It defines system RAM memory excluded from the Free Space Manager in the address space from 0 to 1FFFFH. The system uses this memory as Volatile System Memory, which is static memory used for stack and data by the OS layers and application program.

This Volatile System memory must be below and contiguous to the first FSM section for Free Space Memory. It must be at least 300H bytes in length since the ROM Initialization Code uses 300H bytes of memory just below the start of the first FSM section for its own stack and data area. The OS and/or application can also use this memory since the ROM Initialization Code will already have completed its work by the time the OS begins.

Calculating Volatile Memory Size

In configuring your application system to be ROM\Flash-based, you must reserve a certain portion of Volatile System Memory as static data and stack. To identify the minimum memory requirements for your specific application, you can calculate the memory requirements based on information in the *.mp2* file generated for your application. The demonstration application generates the *mix486dx.mp2* file (Figure 9-1).

As shown in Figure 9-1, the Segment Map of an *.mp2* file lists the base address and limit of each segment defined in the application system. Using the information in both the GDT and LDT sections of the Segment Map, you can calculate the amount of code (MEMS) and data (VSS and VSE) needed by your application system, as follows.

1. Find the highest code physical address in non-volatile ROM. These addresses start with "FFF". In *mix486dx.mp2*, the highest code address is LDT Slot 96 listed in this line:

```
96 1  0  ER  32FFFD650CH 0000050FH START.CODE
```

2. Add the base address (FFF0650H) and the limit for the code address (50FH) to obtain their sum (FFF06A1BH).
3. Obtain the high address in the MEMS screen, which is FFFFFFEFH in the example.
4. The sum of the base address and limit (FFF06A1BH) must be less than or equal to the MEMS high address (0FFFFFFEFH), as is the case in the example.

Now calculate the memory requirements for RAM:

1. Find the highest data physical address in RAM. These addresses start with "0000". As seen in Figure 9-1, the highest data address is listed is GDT Slot 302:

```
302 1  0  RW  32 0000FB8CH 00000007H ?DUMMY_MODULE.CC_120_SEG_5
```

2. Add the base address (0FB8CH) and the limit for the data address (7H) to obtain their sum (FB93H).

3. The sum of the base address and limit must be less than or equal to the VSE high address (1FFFFH). Finally, adjust the VSE parameter to be equal to the low address of the MEMF entry minus one.

Note

If you do not allocate enough Volatile System Memory, you will see the following error message when you generate the system. If the *segment_name* is a data segment, check the VSS and VSE entries. If the *segment_name* is a code segment, check the MEMS entries.

```
*** WARNING 177: SEGMENT ALLOCATED OUTSIDE SPECIFIED RANGE
    SEGMENT: segment_name
```

Set the base address of the ROM Initialization Code to 0FFFFF000H using the RBA entry in the ROM screen. This address must be on a 4 Kbyte boundary and be within 64 Kbytes of the system restart vector, which resides at 0FFFFFFF0H. The restart vector does a near jump to this address.

Note

Accessing an address of 0FFFFF000H while in Real Mode is based on a feature of all Intel Architecture microprocessors. At reset, all address lines are driven high by the microprocessor and stay that way until the first far jump is made. The ROM Initialization Code makes sure the hardware descriptor tables (GDT and IDT) refer to this high memory address area by the time the first far jump is made (immediately after switching to PVAM).

You can verify the size of the ROM Initialization Code by looking at the Segment Map in the *.mp2* file generated by the BLD386 utility. Refer to Figure 9-1, LDT Slot 91.

The size of the ROM Initialization Code varies based on the amount of code your application requires to properly configure your system hardware. In the MIX486 example, the code is approximately 3500 bytes in length.

Since the RAM-less mode of ROM Initialization is used, this example sets the RDA and SRC entries to 0H.

When the ICU generates the configuration files for a ROM-based system, it creates a ROM Custom Initialization include file whose name is *definition_file_base_name.inc.inc*. The ICU places into it a set of empty ASM386 macros as well as a small amount of assembler code. In the MIX486 example, the ICU definition file is *mix486dx.bck*, so the ROM Custom Initialization include file is created as *mix486dx.inc*. See the comments in the *.inc* file for areas where you can customize the initialization code.

When developing a ROM-based iRMX system, modify this ROM Custom Initialization include file to use your custom code. Copy the file to a different file whose pathname you list in the CRS entry. The System Generation submit file copies the CRS-specified file over the ROM Custom Initialization include file. It uses this file when generating the ROM Initialization Code object files. By giving it a different name, you insure your modifications to the ROM Custom Initialization include file will not be destroyed the next time you run the ICU.

As part of the modifications made to *mix486dx.inc* to yield *mixinit.inc*, calls are made to two near procedures, `mix4_init` and `init_486`. Specify the link file containing these two procedures as *mix4in.lnk* in the CRO entry of the ROM screen. These calls are specific to the MIX486 board. The files *mix4in.lnk*, *mix486dx.bck*, and *mixinit.inc* are located in the */rmx386/demo/rom/mix486* directory.

Note

If you are programming ROM on different target hardware, you can create your own external procedures. This means you must:

- Use 16-bit code
- Name the Code Segment as “code_rom”
- Not use a data segment
- Modify the *.inc* file to call your procedure
- Modify the CRO entry of the ROM screen

```
(INCL)       Includes and Libraries  [1-30 Characters]
(UDF) UDI Includes and Libs   /RMX386/UDI/
(HIF) Human Interface Includes and Libs   /RMX386/HI/
(EIF) Extended I/O System Includes and Libs   /RMX386/EIOS/
(ALF) Application Loader Includes and Libs   /RMX386/LOADER/
(BIF) Basic I/O System Includes and Libs   /RMX386/IOS/
(MNF) Intel Monitor Includes and Libs   /RMX386/SDM/
(SDF) System Debugger Includes and Libs   /RMX386/SDB/
(NUF) Nucleus Includes and Libs   /RMX386/NUCLEUS/
(ILF) Interface Libraries   /RMX386/LIB/
```

```
(DTF) Development Tools Path Name   :LANG:
(VMF) Virtual 8086 Mode includes and libs   /RMX386/VM86/
(NET) iRMX-NET Files   /RMX386/RMXNET/
(CLF) Shared C Libraries   /RMX386/CLIB/
(ISL) Intel Support Libraries   /INTEL/
(SJM) System Jobs Object Modules   /RMX386/JOBS/
```

Use the standard iRMX generation screen and directory structure to generate the application system.

```
(GEN)        Generate File Names
(RMB) Remote Boot Translation   [Yes/No]  NO
(RBF) Remote Boot File Name   [1-55 Chars]  /RBOOT32/RMX386.386
(ROF) ROM Code File Name   [1-55 Chars]  MIX486DX.ROM
(RAF) RAM Code File Name   [1-55 Chars]  MIX486DX.RAM
```

The file you specify in the ROF parameter is the OMF386 output of the builder. This output is your iRMX application system which you can program into ROM/FLASH.

The comment record allows you to tag your definition file, specifying its contents. This record is placed in the Nucleus code segment and is available through a pointer to it in the RQSYSINFO segment cataloged in the root job.

```
(COMNT) Comments for Build file each line = 1-55 characters - IN QUOTES
[ 1]  =  'iRMX III Release 2.2 Operating System      '
[ 2]  =  'for MIX486DX33 and MIX486DX66        '
[ 3]  =  'Nucleus/SDM/SDB in ROM using 28f020 flash devices  '
[ 4]  =  'RAM-LESS ROM Init Version   '
[ 5]  =
```

Debugging the ROM Initialization Process

To help debug the ROM Initialization process, there are debug write calls at strategic points in the ROM Initialization Code path. The purpose is to send an output character through an I/O port so you can track the progress of ROM Initialization Code as it executes on your board.

Note

BX register—Your code must preserve the contents of the BX register at the beginning of the `custom_init_real_mode` macro and restore this value to the BX register before leaving the `custom_init_real_mode` macro.

When developing your own "DebugOp" code, be aware that the character to be output is passed to the `DebugOp` macro using the AL register.

The file *mixinit.inc*, derived from *mix4dxro.inc*, is listed below. This file identifies those sections of the code you must change to support your MIX486 board.

If you wish to have debug characters sent to your output device, you must initialize and activate your device by placing the appropriate code in `DebugOp` and `custom_initial_hw_setup` macros.

```
%*define(DebugOp(val))  (%'
; Place any debug output/notification instructions here for aid in
; debugging the rom initialization code.  Only I/O instructions are
; recommended since the same routine will operate in both real and
; protected mode
;
; Code which prints debug information to COM1
;
;       mov             dx, 03F8H
;       mov             al,     %val
;       out             dx, al
;       mov             dx,     03FDH
;do_input:
;       in              al, dx
;       and             al, 40H
;       jz              do_input
;purge  do_input
;
        nop
)%'
```

```
%*define(custom_extrn_1) (%'
;
; Place any external procedure declaration here which will be jumped TO
; from custom_initial_hw_setup.  Since the stack is NOT set up at this
; time, only a jump instruction is allowed if an external procedure is
; to be activated.  In this case, a label must be placed after the jump
; instruction in custom_initial_hw_setup so that the execution flow can
; return there via a jump in the external procedure.

;EXTRN  my_initial_hw_setup_proc
)%'

%*define(custom_extrn_2) (%'
;
; Place any external procedure declaration here which will be CALLED or
; JUMPED TO from custom_init_real_mode and/or
; custom_init_protected_mode. In the case of RAM-LESS rom
; initialization, the stack is NOT set up until just before the call to
; custom_init_protected_mode; therefore, only a JMP instruction is
; allowed in custom_init_real_mode if an external procedure is to be
; activated.  In this case, a label must be placed after the JMP
; instruction in custom_init_real_mode so that the execution flow
; can return there via a JMP instruction in the external procedure.
; In the case of RAM-FULL rom initialization, the stack will be set up
; before custom_init_real_mode is called.  Thus, a CALL instruction is
; allowed in the custom_init_protected_mode subroutine in both RAM-LESS
; and RAM-FULL modes of rom initialization but is only allowed in
; custom_init_real_mode in the RAM-FULL mode of rom_initialization.

;EXTRN  my_custom_init_real_mode_proc
;EXTRN  my_custom_init_protected_mode_proc

EXTRN   mix4_init: near
EXTRN   init_486: near
)%'

%*define(custom_initial_hw_setup)  (%'
;
; Place any board initialization code here which must be done when the
; system resets, i.e. before the ROM Initialization code starts to run
;
       mov     cr2, edx
)%'
```

```
%*define(custom_init_real_mode)  (%'
;
; Place any board initialization code here which must be done
; while the system is still running in Real Mode, i.e. before the
; ROM Initialization Code switches the processor to Protected
; Mode.  If an external procedure must be accessed from
; custom_rom_init, be sure to use a JMP instruction if the rom
; initialization mode is RAM-LESS.
;
       nop
)%'

%*define(custom_init_protected_mode)  (%'
;
; Place any board initialization code here which must be done
; immediately after the ROM Initialization Code has switched the
; system to Protected Mode
;
;*******************************************************

       push ds
       push es
       push fs
       push gs
       mov  edx, cr2
       push dx
    call mix4_init                  ;             mix4_init(cpu_sig)

    call init_486                   ;             /* enable 486
internal cache */
       pop  gs
       pop  fs
       pop  es
       pop  ds
)%'

%*define(custom_clear_rnc)  (%'
;
; Procedure clear_rnc which is called after switch to protected mode
;
code_rom       segment er use16 public
;
```

```
; Dummy procedure clear_rnc.  The real clear_rnc procedure is
; required for Multibus II systems.  Therefore, if your target
; system runs on Multibus II, comment out this dummy clear_rnc
; procedure by placing a ';' in front of each of its four lines.
;
;       public clear_rnc
;clear_rnc     proc
;       ret
;clear_rnc     endp
code_rom ends
)%'
%*define(monitor_break_option)  (%'
;
; Variable used to indicate if the user wishes to break to the
; SDM monitor upon completion of the ROM Initialization Code and entry
; into the nucleus initialization code. Set to 0FFH if monitor break is
; desired, otherwise set to 0.
;
; NOTE: Only set MONITOR_BREAK to 0FFH if you have iSDM configured into
; the iRMX application system.

        PUBLIC  MONITOR_BREAK
MONITOR_BREAK  DB  0H
)%'
```

To verify that the iRMX Nucleus initialization code has been entered, set the SLV entry in the SDB screen to 0FFH (you can change this later if you do not want SDM configured in your final system).

With the "DebugOp" macro modified and the output device initialized, the following ASCII characters will appear on a terminal connected to your output device:

1 <====== Sent to output device by initialization code above

2 <====== Sent to output device immediately after call to custom_init_real_mode macro - will probably be overwritten by the next character if code switches successfully into protected mode

3 <====== Sent to output device immediately after call to custom_init_protected_mode macro

1 <====== Sent to output device before microprocessor type is determined

Next 8 characters are the base address in RAM in reverse order at which the iRMX GDT will be placed

0 <===== Translates to 18000H

0

0

8

1

0

0

0

4 <====== Sent to output device as delimiter before the iRMX GDT has been copied from nonvolatile memory to RAM and expanded

Next 8 characters are the base address in RAM in reverse order of the iRMX GDT just prior to loading it using an LGDT instruction; this is the address to which the LGDT instruction will point

0 <===== Translates to 18000H

0

0

8

1

0

0

0

5 <====== Sent to output device immediately after LGDT instruction has been issued; ROM Initialization Code now running out of the iRMX GDT

6 <====== Sent to output device immediately after LIDT instruction has been issued; ROM Initialization Code now running out of the iRMX IDT

7 <====== Sent to output device immediately after LTR instruction has been issued; ROM Initialization Code now running out of a temporary Hardware Task defined in the iRMX GDT

8 <====== Sent to output device immediately before jumping to the iRMX Hardware Task; the ROM Initialization Code has just set up the iRMX Hardware TSS to reflect the new Free Space Memory base address

After the series 5, 6, 7, and 8 appear on the terminal, the flow of control leaves the ROM Initialization Code and enters the iRMX nucleus initialization code.

Testing the Application

There are two ways to execute the test application. You can execute the *receive* program from the Human Interface during RAM-based testing or as a user job which is executed from ROM on the Multibus II target.

To run the *receive* application from the Human Interface on the Multibus II target, attach to the */rmx386/demo/rom/mix4demo* directory containing the application and type:

- **receive <CR>**

No messages will be displayed and the program will continue to run until terminated by a Ctrl-C character.

The *receive* program waits to receive a message at port 0x801 sent by the *sendmb2* application. When it receives the message, it forms a new message and returns it to *sendmb2*.

To run the *receive* application from ROM, first follow the directions in this chapter to generate the application and burn the ROMs. Install the ROMs in the target and then apply power to the system.

To test whether the *receive* application is running successfully, regardless of whether it runs from the Human Interface or from ROM, execute the *sendmb2* program. From the Human Interface on another Multibus II board, attach to the */rmx386/demo/rom/mix4demo* directory containing the application and type:

- **sendmb2 *slot_id* <CR>**

Where *slot_id* is the slot number of the Multibus II agent running the *receive* application.

The *sendmb2* program sends a message to port 0x801 on the Multibus II agent running the *receive* program.

The final display from the *sendmb2* program is:

```
Attempting to send 50 messages to slot X
Messages sent/received [50]
Program terminated successfully.
```

❑❑❑

Developing Applications for Multibus II 10

This chapter provides a conceptual explanation for most of the Multibus II examples provided with the iRMX OS. These examples provide a more complete understanding of message passing techniques using the iRMX OS.

Code Examples

Each example in the manual includes a brief description of the example. Source code for each example is provided with the iRMX OS.

> **Note**
> The files *dcomext.h* and *dcomlit.h* are common to the examples in this chapter.

The source code for the examples are located in the */rmx386/demo/c/mb2/intro* directory. To attach to this directory, type:

```
- af /rmx386/demo/c/mb2/intro <CR>
```

To generate the proper executable 32-bit modules for these examples, run the generation command (DOS batch file) for your compiler:

Compiler	Generation Command
iC-386 demo	- make
Microsoft C	- mscdemo
Watcom C	- watdemo

If each host has its own disk, enter this command on both host's terminals. If one of the hosts is diskless, use the file server to generate the example.

Examples Using Nucleus Communication System Calls

The examples in this chapter are presented in an order similar to their use in a real system. The examples step you through these concepts:

Module	Use
icscan.c	Scanning the system to determine what boards are in the system. This example runs independently of all the other modules.
tranport.c	Creating a data transport protocol port to use in message passing.
sndrsvp.c	Sending an RSVP message to another board and waiting for a reply. This module must be run with *rcvrsvp.c* or *sndfrag.c*.
rcvrsvp.c	Answering an RSVP message from the receiving board. This module must be run with *sndrsvp.c*.
sndmsg.c	Sending a contiguous buffer. This example must be run with either *rcvmsg.c* or *dcrcvmsg.c*.
dcsndmsg.c	Sending a data chain message. This example must be run with either *rcvmsg.c* or *dcrcvmsg.c*.
rcvmsg.c	Receiving a contiguous buffer. This example must be run with either *sndmsg.c* or *dcsndmsg.c*.
dcrcvmsg.c	Receiving a data chain message. This example must be run with either *sndmsg.c* or *dcsndmsg.c*.
sndfrag.c	Sending a fragmented message. This example must be run with *sndrsvp.c*.
rcvfrag.c, *sfrag.c*	Receiving a fragmented message.

Note

The examples make certain assumptions about the locations of the host boards in the Multibus II system that they run on. The REMHOSTID definition in the *sndrsvp.c*, *sndmsg.c*, *dcsndmsg.c*, *sfrag.c* examples assume the processor location board is in slot 0. Change this definition if you want to change the remote host to any processor board in the board.

Interconnect Space Example - iscan.c

Before passing messages between agents (boards) in your system, you need to determine what boards are in your system and the message addresses (cardslot number for boards on the PSB) for the boards. Writing a board scanner task will provide you with this information. This task accesses an interconnect register, allowing you to dynamically determine host IDs, board type, and multiple occurrences (instances) of a board type.

This section presents an example of getting the interconnect information for an entire system. The example performs the board scan, get the slot number and board type of each board in the system and places the information into an array of structures called `sys_map`. When the board scan is complete, `sys_map` is displayed on the console screen.

Figure 10-1 presents a board-scanning algorithm. The read statements in this figure refer to the **rq_get_interconnect** system call. For a map or template of a particular board's interconnect registers, refer to the board's hardware reference manual.

```
FOR i = 0 to number of slots minus 1
DO;
   Read board(i) vendor ID register;
   IF vendor ID <> 0 then
   DO;
     Read board(i) class and subclass ID registers  /*
                                          Determine
                                          board type */
     Write the board information into the system map
   END;
   ELSE;
     Write 'empty' into the sys_map for the slot number
   END;
END;
Get ID of local host
FOR i = 0 to number of slots minus 1
DO:
   Print slot numbers and board types to console screen
END;
```

Figure 10-1. Board Scanning Algorithm

In the fourth line of the board scanner algorithm, a vendor ID of 0 (for PSB hosts only) indicates that either the board was manufactured by a non-licensed vendor or the cardslot is empty. If you are also scanning the iLBX II bus, replace the 0 with 0FFFFH.

To run the board scanner example, type:

```
- icscan <CR>
```

The source code for this example is located in the */rmx386/demo/c/mb2/intro* directory.

Creating a Port for Message Passing - tranport.c

Once you have information on what boards are in your system, the next step is to create a port for message passing and associate a buffer pool with it. This module creates a buffer pool, releases a number of 400H byte buffers to it, creates a data transport type port, and then creates a token to use as a reference to the port.

The source code for this example is located in the */rmx386/demo/c/mb2/intro* directory.

Sending Data Using Send_rsvp

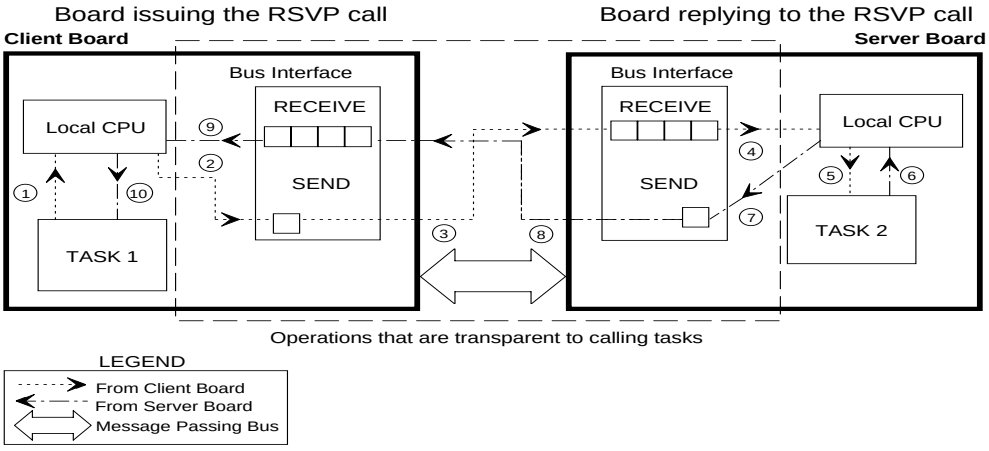
Now that you have information on the boards in the system and a data port, you are ready to send data in message form. The next example illustrates one of the most common message passing formats, the request/response, typically used between two iRMX hosts. Two terms, client and server, are used to describe the boards involved in request/response messages. The client is the requesting board and the server is the responding board.

Figure 10-2 shows the logical representation of the message-passing model for a request/response transaction. A task on the client board initiates the transaction by sending an **send_rsvp** call to a well-known port on the server board (see Figure 10-3). Because the ports on a remote board cannot be dynamically determined, this example assumes a port that is created on all boards as a starting point for message passing. Once you have a `host_id` for a remote board (REMHOSTID), you combine it with the `port_id` (REMPORT) of the well-known port to create the socket for the destination of a message. When the server board receives the message, it replies with the **send_reply** call (see Figure 10-4). The request/response messages continue until the data requested in the original **send_rsvp** system call is received by the task on the client board.

See also: **send_rsvp**, **send_reply**, *System Call Reference*

For this example, we assume:

- The port on the client board has one buffer large enough for the requested data.
- The port receiving the RSVP message is not being used as a sink port.



1. Task 1 on the Client board issues a **send_rsvp** system call. In an RSVP/REPLY transaction, the board that issues the call is the client; the board that replies is the server.
2. The Nucleus Communication Service (NCS) turns the information from the **send_rsvp** system call into a message then sets the buffer space for the expected reply.
3. The Message Passing Coprocessor (MPC) sends the message across a message passing bus to the remote agent specified in the **send_rsvp** system call.
4. The CPU on the server board receives a PIC interrupt because a Multibus II message has been received.
5. The NCS on the server board directs the message to the appropriate port (and, therefore, task).
6. Task 2 responds with a **send_reply** system call that contains information about the data being sent.
7. The NCS on the server board turns the information in the **send_reply** system call into a message that is sent by the MPC.
8. The message travels across the message passing bus, an operation transparent to the operating systems on both boards.
9. The MPC on the client board places the message into the buffer that was set up in step 2, and then sends an interrupt to the CPU, informing it of the completion of the message transaction.
10. The NCS on the client board directs the message to the correct task using the port ID (REMPORT). The CPU on the client board is aware of operations in steps 1, 2, 9, and 10.

Figure 10-2. An RSVP/REPLY Transaction between Two iRMX Hosts

Figure 10-3 is an algorithm for the client board in this transaction.

```
Client board
       Call an external procedure called get$dport that returns a
       TOKEN for the local port to be used in the RQ$SEND$RSVP call.

       Initialize the socket structure, declared externally.

       Set the message size to be zero length.

       Initialize the global variable rsvp_size to the LITERAL RSVPB
       (128 bytes).

       Issue the RSVP system call using the previously initialized
       variables.

       Use the RQ$RECEIVE$REPLY system call to wait for an answer.

       Send the reply message, "This is a send$reply message" to the
       console screen.

       Exit from the example.
```

Figure 10-3. Algorithm for the Client Board

Figure 10-4 is an algorithm for the server board in this transaction.

```
Server board

       Call an external procedure, get$dport, that returns a TOKEN to
       be used in the RQ$RECEIVE and RQ$SEND$REPLY calls.

       Perform an RQ$RECEIVE using the TOKEN returned from get$dport.

       Perform an RQ$SEND$REPLY on successful completion of the
       RQ$RECEIVE.

       IF the data arrives correctly (msg_ptr <> NIL)
              Return the buffer to the buffer pool.

       End server procedure.
```

Figure 10-4. Algorithm for the Server Board

The send message example must be run with the corresponding receive message example. To run these examples, first type this command on the host in slot 0, or in the slot as server defined by the REMHOSTID parameter:

- **`rcvrsvp <CR>`**

Then type this command on the host in any slot:

- **`sndrsvp <CR>`**

The source code for this example is located in the */rmx386/demo/c/mb2/intro* directory.

Sending and Receiving Messages

This section presents examples of sending and receiving buffers (messages) either as contiguous buffers or as data chains. The example is presented in two modules, one that sends a message and one that receives it. A port's ability to receive messages in data chain form is set according to the attributes of the port's associated buffer pool.

The programs for sending messages:

File	Action	Object
sndmsg.c	Send	Contiguous buffer
dcsndmsg.c	Send	Data chain buffer
rcvmsg.c	Receive	Contiguous buffer
dcrcvmsg.c	Receive	Data chain buffer

The source code for this example is located in the */rmx386/demo/c/mb2/intro* directory.

Receiving a Message

The receive example must be run with the corresponding send message example. To run a receive example, first type one of these commands on the server in slot 0, or in the slot as server defined by the REMHOSTID parameter:

- **`rcvmsg <CR>`** for sending a contiguous buffer

or

- **`dcrcvmsg <CR>`** for sending a data chain

After setting the host in slot 0 to receive, run the respective send example on another host. After receiving the message, the host terminal in slot 0 displays:

```
Message received by [rcvmsg|dcrcvmsg] as a
  [contiguous buffer|data chain] is as follows:
This is the message sent by [sndmsg|dcsndmsg] as
  a [contiguous buffer|data chain].
```

Sending a Message

The send example must be run with a receive message example. To run a send example, type one of these commands on the host in any slot other than 0:

- **`sndmsg <CR>`** for sending a contiguous buffer

or

- **`dcsndmsg <CR>`** for sending a data chain

Sending a Message in Fragments

This section presents an example of sending and receiving a message that is broken into fragments. The example is presented in two modules, one that sends the fragmented message and one that receives it. A port's ability to receive messages in fragment form is set according to the attributes given to the port at the time of its creation.

The send fragment example must be run with the send RSVP procedure. To run these examples, first type this command on the server in slot 0, or in the slot as server defined by the REMHOSTID parameter:

- **`sndfrag <CR>`**

This procedure breaks the data into fragments and sends them to a processor board. Then type this command on the host in any slot other than 0:

- **`sndrsvp <CR>`**

This procedure receives the fragmented data and displays it on the host terminal from which the **sndrsvp** command was executed:

```
This is a reply sent in fragments.
```

Receiving a Message in Fragment Form

This section presents an example of sending a message and receiving it in fragment form. The example is presented in two modules. The first module, *sfrag*, initiates a transaction which forces receiving in fragment form. The second module, *rcvfrag*, receives the message and prints it on the console screen. To run these examples, first type this command on the server in slot 0, or in the slot as server defined by the REMHOSTID parameter:

```
- rcvfrag <CR>
```

Then type this command on the host in any slot other than 0:

```
- sfrag <CR>
```

The host terminal from which the **sfrag** command was executed displays:

```
This is a reply to a fragmented message.
```

The host terminal in slot 0 displays:

```
This was received via fragmentation.
This is the second fragment.
```

The Name Server Example

This is the most complex example provided with the iRMX OS. This example implements a table that dynamically catalogs the names of all the ports created in a system. Two tasks, one for remote requests and one for local requests, manage the name server table.

The remote server task uses both control and data messages to service requests. The local server services requests through data mailboxes. The name server table is implemented as a circular list which is accessed by procedures that insert or delete port names, get or change socket information, and set up the table for these accesses.

When a client board makes a request to the name server, the request is sent, the calling task waits for a reply, and the name server returns information specific to the request (e.g., the result of modifying an entry in the table or the socket for a remote port).

The example, written in PL/M, for the name server is located in the */rmx386/demo/plm/mb2/nservr* directory. This command makes the directory containing the name server example the current directory.

- **af /rmx386/demo/plm/mb2/nservr <CR>**

To generate the executable name server, run the *makefile* by entering:

- **make <CR>**

Note

If an error is generated after running the makefile, you may need to modify the file. Edit this file and delete the WORD16 switch from this line:

```
PLMFLAGS=$(DEBUG) Set(r_32) word16
```

The name server can be run as a background job to one of the processors. To start the name server running as a background job, enter:

- **background nservr > nservr.doc <CR>**

See also: **background** command, *Command Reference*

Two modules demonstrate the use of the name server: *nssndmsg* and *nsrcvmsg*, which execute as a pair. *Nsrcvmsg* must execute first. It posts a socket with the name server under the name receiver. *Nssndmsg* then executes, sending the name server a look-up request on the name receiver. *Nssndmsg* then sends a message to receiver and *nsrcvmsg* prints the message:

On the same host in which you invoked nservr as a background process, enter:

- **nsrcvmsg <CR>**

On another host, enter:

- **nssndmsg <CR>**

The host terminal displays:

```
This is a simple message.
```

This process can be demonstrated on either host board, but the order of module execution cannot be changed.

The General Examples

The two examples presented in this section are located in the */rmx386/demo/plm/mb2/general* directory. The concepts they demonstrate are:

- Example 1: sending and receiving unsolicited messages
- Example 2: sending and receiving asynchronous solicited messages

To examine the examples, attach their directory by entering:

```
- af :rmx:demo/plm/mb2/general <CR>
```

To generate the executable modules for all of these examples, run the *makefile* by entering:

```
- make <CR>
```

If each host has its own disk, this command must be entered on both host's terminals. If one of the hosts is diskless, enter this command on the host which is acting as its fileserver.

Note

The module *utils.lit* contain default client and server PSB slot definitions. They can be changed for running the examples. All PSB slot numbers are in hexadecimal.

Example 1: Sending and Receiving Unsolicited Messages

This example demonstrates sending and receiving unsolicited messages. It can be executed on any Multibus II boards running the OS or on any single board running as both the CPU and the communications board (short-circuit mode). The client and server boards must be situated in slot CLIENTPSBSLOT and SERVERPSBSLOT, respectively. These slots are defined in *utils.lit*, located in this example's directory.

In this example, the client is defined as host 4 and the server as host 2.

Execution of Client and Server Programs

This table shows the various steps the client and server programs perform during the execution of example one.

Table 10-1. Flow of Program Execution for Example 1

Steps	Program	Action
1	client server	Enable in-line exception handling Enable in-line exception handling
2	client server	Create port object; associate port with a default remote socket Create port object; associate port with a default remote socket
3	client	Prompt user for message Encrypt message Send message asynchronously to server Wait for response from board in slot SERVERPSBSLOT
4	server	Receive message and display in encrypted form Decrypt message and display in decrypted form Send decrypted message back to client board
5	client	Display decrypted message Prompt user for another message

This cycle repeats steps 3 through 5 until six messages have been sent and received. The programs then terminate.

Running Example 1

To run this example, first enter this command on the host in slot 4:

- **clnt32 <CR>**

The terminal displays:

```
Enter any string of characters:
```

Second, enter this command on the host in slot 2:

- **srvr32 <CR>**

The server will wait for input from the host in slot 4. For example, your message on host 4 can be:

My exciting message! <CR>

When host 0 receives the message, it first displays the encrypted version, then the decrypted version.

```
Message received is:  [encrypted version is displayed]
Converted message:  My exciting message!
```

The server then sends the converted message back to the client, which displays the message and prompts for the next input.

```
Message received is:  My exciting message!
Enter any string of characters:
```

After six messages, both programs terminate.

Example 2: Sending Asynchronous Solicited Messages

This example demonstrates sending asynchronous solicited messages and using buffer pools. It can be executed on any Multibus II boards running the OS or on any single board running as both the CPU and the communications board (short-circuit mode). The client and server boards must be situated in slot CLIENTPSBSLOT and SERVERPSBSLOT, respectively. These slots are defined in *utils.lit*, located in this example's directory.

In this example, the client is defined as host 4 and the server as host 2.

Execution of Client and Server Programs

This table shows the various steps the client and server programs perform during the execution of example two.

Table 10-2. Flow of Program Execution for Example 2

Steps	Program	Action
1	client server	Enable in-line exception handling Enable in-line exception handling
2	client server	Create port object; associate port with a default remote socket Create port object; associate port with a default remote socket
3	client server	Create buffer pool; associate pool with the port created earlier Create buffer pool; associate pool with the port created earlier
4	client server	Create buffers and release them to the pool Create buffers and release them to the pool
5	client	Prompt user for message Encrypt message and send message asynchronously to server Wait for asynchronous send transmission message
6	client server	Wait for response from board in slot SERVERPSBSLOT Receive encrypted msg from board in slot CLIENTPSBSLOT Move message from buffer pool buffer to application buffer Release the buffer back to the buffer pool Decrypt message and display decrypted form Send decrypted message synchronously to client board
7	client	Release buffer back to buffer pool; display decrypted message Prompt user for another message

This cycle repeats steps 5 through 7 until eight messages have been sent and received. The programs then terminate.

Running Example 2

To run this example, first enter this command on the host in slot 4:

- **solclnt32 <CR>**

The terminal displays:

```
Enter any string of characters:
```

Second, enter this command on the host in slot 2:

- **solsrvr32 <CR>**

The server will wait for input from the host in slot 4. For example, your message on host 4 can be:

- **My exciting message! <CR>**

When host 0 receives the message, it first displays the encrypted version, then the decrypted version.

```
Message received is:  [encrypted version is displayed]
Converted message is:  My exciting message!
```

The server then sends the converted message back to the client which displays the message and prompts for the next input.

```
Message received is:  My exciting message!
Enter any string of characters:
```

After eight messages, both programs terminate.

❑❑❑

Developing Applications in Assembly Language 11

This chapter provides information on invoking system calls from assembly language. It also provides an example of an interrupt handler and an OS extension interface.

Files referred to in this chapter are located in the */rmx386/demo/asm/intro* directory.

Invoking System Calls from Assembly Language

To invoke system calls from assembly language programs, the assembly language programs must obey the Fixed Parameter List (FPL) procedure-calling protocol used by C and PL/M. For example, if your ASM386 program uses the SendMessage system call, then you must call the RqSendMessage interface procedure from your assembly language code.

In general, to call a C or PL/M procedure, do this:

1. Push all the parameters onto the stack.

   Push the parameters in the order they are listed in the system call reference manuals; that is, starting with the leftmost parameter and working towards the right.

   Push long pointers (complete addresses consisting of a selector and an offset) selector (as a 16-bit value) first, then the offset (as a 32-bit value for PL/M 32-bit mode).

2. Call the procedure.

   The CALL instruction also places the return address of your calling procedure onto the stack. This enables control to return to your program after the system call completes.

Some system calls return values. In assembly language, the returned values are available in registers as listed in Table 11-1.

Table 11-1. Registers Containing Returned System Call Values

Type	32-bit Register
BYTE	AL
WORD	AX
DWORD	EAX
INTEGER	AX
POINTER	DX:EAX
SELECTOR	AX

The file *reg.inc* (used by the interrupt handler example) contains macro definitions used to produce common source code for iRMX II and III. These macro definitions define the register values shown in Table 11-1. The interrupt handler description on page 167 shows how to invoke these definitions.

When writing assembly language routines that call iC-386 or PL/M-386 interface procedures, use the compact model with ASM near calls.

If some of your application code is written in either C or PL/M, your assembly language code should use the same interface procedures as those used by your code. However, if your application is written entirely in assembly language, using the compact interface library and coding your application to make NEAR calls will produce size and performance advantages.

See also: Using Compact and Large Memory Models, Chapter 7

This listing of *reg.inc* shows definitions for common sourced code.

```
       ; macro definitions for common sourced code
%IF    (%RMX EQ 3)  THEN   (%'
       %define      (ax)         (eax)
       %define      (bx)         (ebx)
       %define      (cx)         (ecx)
       %define      (dx)         (edx)
       %define      (si)         (esi)
       %define      (di)         (edi)
       %define      (bp)         (ebp)
       %define      (sp)         (esp)
       %define      (mov16)      (movzx)
       %define      (pusha)      (pushad)
       %define      (popa)       (popad)
       %define      (pushf)      (pushfd)
       %define      (popf)       (popfd)
       %define      (iret)       (iretd)
       %define      (dw)         (dd)
       %define      (dd)         (dp)
)      ELSE   (%'
       %define      (ax)         (ax)
       %define      (bx)         (bx)
       %define      (cx)         (cx)
       %define      (dx)         (dx)
       %define      (si)         (si)
       %define      (di)         (di)
       %define      (bp)         (bp)
       %define      (sp)         (sp)
       %define      (mov16)      (mov)
       %define      (pusha)      (pusha)
       %define      (popa)       (popa)
       %define      (pushf)      (pushf)
       %define      (popf)       (popf)
       %define      (iret)       (iret)
       %define      (dw)         (dw)
       %define      (dd)         (dd)
)      FI%'
```

This example shows how to call iRMX system calls from assembly language. The example assumes that the compact segmentation model is used.

```
DATA segment RW PUBLIC
     seg_tok  DW ?
     excep    DW ?
     DATA ENDS

     CODE segment ER PUBLIC

     extrn rqcreatesegment:  near

     my_prog PROC near
     ;
     ; Get addressability to parameters
     ;
     push  ebp
     mov   ebp, esp
     ;
     ; Save caller's DS and obtain local DS
     ;
     push ax
     push  ds
     mov   ax, data
     mov   ds, ax
          .
          .   Typical ASM statements
          .
     ;
     ; seg_tok = rq$create$segment (400H, @excep);
     ;
     movzx  ax,400H
     push   eax
     push   ds
     push   offset excep
     call   rqcreatesegment
     mov    seg_tok, ax
     ;
     ; IF except <> E$OK THEN GOTO error;
     ;
     cmp  excep, 0
     jnz error
          .
          .   Typical ASM statements
          .
     my_prog ENDP
     CODE    ENDS
     END
```

Interrupt Handler Example

The assembly language application, *inthand.asm*, provides an example of an interrupt handler. The include file, *reg.inc*, used by this application provides macro definitions used for various versions of the iRMX OS. The proper definitions are invoked using one of these ASM invocation lines (from *makefile*):

```
asm86  .ob1  .ls1
asm286  .ob12 .ls2
asm386  .obj  .lst
```

Generating the Interrupt Handler Example

The *inthand.asm* file contains the assembly language code for the interrupt handler. To examine the example, attach to the directory by entering:

```
- af /rmx386/demo/asm/intro <CR>
```

The *inthand.asm* file contains the assembly language code for the interrupt handler. To generate the object file from *inthand.asm*, run the *makefile* utility by entering:

```
- make <CR>
```

OS Extension Example

This assembly language code provides a listing of the recommended interface to an OS extension.

Once a call gate has been reserved for use as an OS extension (either using the ICU in the iRMX III OS, or using the *rmx.ini* configuration), it can be bound to an application using the **rqe_set_os_extension** system call. Other applications can access the OS extensions using assembly language interface procedures described below.

This ASM module is a sample interface to Call Gate 441, which is one of the user-accessible gates. The OS Extension procedure tied to the gate has this FAR external interface:

```
   out$char: PROCEDURE(value, status$p) EXTERNAL;
GATE_441     equ     441

$GENONLY
%*DEFINE(CALL_G(ARG))
(DB    9AH
DD    0, %ARG*8)

Name    Interface

Code    Segment   ER    PUBLIC

Public Outchar

Outchar Proc    Near

;
;      PLM CALL -  CALL  OUT$CHAR  (VALUE, @STATUS);
```

Figure 11-1. OS Extension Code in Assembly Language

```
        ;
        ;               STACK  FRAME AFTER PUSHING EBP
        ;
        ;                      ,_____________________________,
        ;           ESP ->     |        OLD EBP              |  [EBP]
        ;                      |_____________________________|
        ;                      |  OFFSET OF RET. ADD.        |  [EBP + 4]
        ;                      |_____________________________|
        ;                      |  OFFSET OF STATUS           |  [EBP + 8]
        ;                      |_____________________________|
        ;                      |  SEGMENT OF STATUS          |  [EBP + 12]
        ;                      |_____________________________|
        ;                      |     VALUE (PARAMETER)       |  [EBP + 16]
        ;                      '_____________________________'
        ;
        ;

        push    ebp
        mov     ebp, esp
        push    dword ptr ss:[ebp + 16]        ; value
        push    dword ptr ss:[ebp + 12]        ; selector of status_p
        push    dword ptr ss:[ebp + 8]         ; offset of status_p

%call_g(GATE_441)

;CALL gate_441 - Invoke entry procedure through the call gate
        les     edi, pword ptr ss:[ebp + 8]   ; Load status_p in es:edi
        mov     cx,es:[edi]                   ; Load condition code
                                              ;  in cx
        jcxz    done                          ; If CX=0, then no error
        ;
        ; Error processing code IF CX <> 0
        ;
        done:
        ;
        pop     ebp
        ret     12
        Outchar Endp

        Code    Ends
        End
```

Figure 11-1. OS Extension Code in Assembly Language (continued)

□□□

Developing Applications in PL/M 12

This chapter contains specific information about using PL/M to create application code. It discusses:

- Making calls to the operating system
- Using include files and libraries
- Linking or binding object modules
- A multitasking demo program that uses iRMX system calls
- A <Ctrl-C> handler example

You should already be familiar with these concepts as well as the PL/M language and PL/M segmentation models.

See also: *PL/M-386 Programmer's Guide*,
Introducing the iRMX Operating Systems

You can compile, bind, and run the demo program from the Human Interface, or you can use the code and this discussion purely as an example of how to perform certain operations in PL/M under the OS.

Invoking System Calls from PL/M

Invoking iRMX system calls is just like calling any PL/M procedure. Because you do not define the system calls in your programs, they must be external procedures. Therefore, include external declarations for each system call you invoke.

See also: Binding Your Code to Interface Libraries

Including External Declaration Files

When you call a procedure that is not defined in your current program module (a separately compiled portion of code), declare that procedure to be external. The binder can then satisfy the references to that procedure as it links the program modules together. A program in one module can then call a procedure in another module.

Include files are supplied with the iRMX OS. These files are placed permanently in one location and provide the external procedure declarations for all iRMX system calls. The declarations are written once, placed in an include file, and then used in place of repeating the actual declaration in each module.

For example, to use the PL/M include file *nuclus.ext*, place this statement at the beginning of your PL/M source code. This statement declares all the Nucleus system calls to be external.

```
$include(/rmx386/inc/nuclus.ext)
```

See also: Header files, *System Calls*, for a list of external declaration include files for PL/M which are supplied with iRMX

Because each include file contains external declarations for many system calls, including a particular file will probably result in external declarations for several system calls your program does not invoke. These extra external declarations pose no problems for the compilers and cause no error conditions.

Binding Your Code to Interface Libraries

After you have written your programs and inserted include statements for the necessary system calls, compile the code and bind it to the appropriate iRMX interface library.

Interface libraries supplied with the iRMX OS provide a standard interface to the system calls. The interface libraries contain procedures that correspond to iRMX system calls. These procedures have the same names and use the same parameters as the system calls. The interface procedure performs operations to invoke the actual system call. For example, iRMX interface procedures make calls to call gates when accessing system calls.

After compiling the program code, satisfy the external references to the system calls by using BND386, which binds the compiled code to the appropriate interface libraries. There are several interface libraries to choose from. The library you choose depends on the system calls and the segmentation model used.

See also: Interface Libraries, *System Call Reference*, for the general iRMX libraries and the UDI libraries

Using the UDI calls exclusively enables an application to be easily transported between Intel OSs. To help you choose which library to bind your program to, consider this:

- If your code includes only UDI system calls or if it uses the I/O support provided by the language, bind your program only to the UDI library.
- If your code does not invoke UDI system calls, or you do not plan to include the language's I/O support, bind the code just to the iRMX library.
- If your code invokes both UDI and other iRMX system calls, bind the code to both of the libraries for the segmentation model you chose.

PL/M Multitasking Example

The PL/M example program is called *exampl32*. In addition to studying this program and its discussion, you can use the files as a starting point in developing your application code. This could save you time when creating the source module, adding include statements, or producing code that attaches the console, etc.

These sections provide:

- An overview of the demo program
- The location of the code in the standard iRMX directory structure
- Information on how to build and run the program

Example Overview

The multitasking example demonstrates some iRMX programming concepts by printing prompts to the console screen and accepting input from the user. To accomplish this, the program uses two tasks: the initial task and a second task called Task2. The main program code contains the initial task and it creates Task2.

The function of the initial task in the main program code is this:

- Set up the programming environment by creating objects, the second task, etc.
- Prompt the user for and capture keyboard input
- Pass the captured input to Task2
- Exit with an error after receiving three user-supplied keystrokes

The function of Task2 is to receive user-supplied keystrokes from the initial task and process them. The processing consists of printing the received keystroke to the screen once every second.

Because the job uses two tasks, each task can perform its function separately from the other task. Communication and data passing between the initial task and Task2 are handled using some basic iRMX programming techniques.

Location of Multitasking Example Code

The files for the multitasking example are in the */rmx386/demo/plm/intro* directory.

Before attempting to understand this example, produce hard copies of the source code or be able to view them from a console screen.

These files are the source files for the examples:

makefile	File to generate 32 bit example
demo.plm	Main program code containing the initial task
task2.plm	Second task code
crbpool.plm	Buffer pool code
except.plm	Exception handling code
strng.plm	String manipulation utility
condec.plm	Decimal conversion utility

Compiling and Binding the Multitasking Example Code

In addition to the example source code files, there is a file you can use to compile and bind the example. The file *makefile* compiles and binds the source files using PL/M-386 and BND386 and creates two 32-bit executable programs named *exampl32* and *tskcom32*.

Note

The example, *tskcom32*, is the PL/M version of the task communication example described in Chapter 5, Debugging Applications.

Before running *makefile*, first attach to the directory where the examples are kept. Then run the *makefile* utility:

```
- af /rmx386/demo/plm/intro <CR>
- make <CR>
```

Now run the example by entering:

```
- tskcom32 <CR>
```

The **make** command executes *makefile*. This initiates the compilation and binding of all the job's source code files.

> **Note**
> If you wish to generate the example as another user, create a new directory, copy the example's files to the new directory, move to that directory and invoke *makefile*. Generating the example from another directory enables you to alter source code, while keeping the original version intact.

Running the Multitasking Example

You should now have a file called *exampl32* that you can execute. To run the example, type its name as follows:

```
- exampl32 <CR>
```

After typing the filename, the example prompts you with this message:

```
iRMX PL/M Example, Vx.y

Welcome to the PL/M Demo Program!

At the prompt you will be given 60 seconds to hit any key.
If you do not hit a key the demo will continue anyway.
You may hit an "E" if you wish to exit the program.

You now have <xx> seconds left to hit a key.
```

At this point, the example is executing code in the promptandwait procedure from the file *demo.plm*. The example is counting down from 60, waiting for you to press a key to begin the demo. The string <xx> in the previous screen is the decrementing count. To continue, press a key. After pressing any key, the example clears the screen and prompts you with this message:

```
Please hit a key which will be forwarded to task2 for
processing.
```

Assuming you enter the letter X for the first keystroke, the main program, containing the initial iRMX task, reads the X from the terminal and passes it on to Task2. Task2 wakes up and prints out this message to the screen:

```
TASK2 PROCESSING
Please hit a key which will be forwarded to task2 for
processing
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX ....
```

The X characters that Task2 prints to the screen continue to appear at the rate of one per second. The character will repeat indefinitely until you enter another keystroke. Also, notice that the prompt to enter another keystroke is buried in the middle of Task2's processing message and the string of letters that it displays. A close examination of the initial task and Task2 show the synchronization used to time the output of these tasks. The tasks use a semaphore to achieve task communication.

Entering the next two keystrokes concludes the example. This output assumes you enter the characters Y and Z:

```
TASK2 PROCESSING Y
Please hit a key which will be forwarded to task2 for
processing
YYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYY ....
TASK2 PROCESSING Z

This concludes the PL/M Demo Program.

This demo now exits by generating an internal error.

INTERNAL ERROR AT # 340 STATUS = 0023: E$SUPPORT
```

After you enter the final keystroke, the initial task recognizes that you have entered three characters. This signals the code to end the program. Notice that the initial task ends the program before Task2 can begin printing the third character to the console screen.

Programming Concepts Illustrated by the Multitasking Demo

This example demonstrates the use of iRMX system calls from a PL/M program. For simplicity, in the discussions of these calls, the iRMX system call prefix (**rq**) is usually dropped. The example illustrates nine common iRMX programming concepts as listed below.

In-line exception processing	The processing of all errors resulting from iRMX system calls in your application code rather than using the default exception handler, which deletes tasks that get errors.
Using literal files	Using separate files that contain PL/M data structure definitions and literal definitions needed to make system calls. Providing separate literal files relieves you from repeating data structure and literal definitions throughout modules.
Getting and setting terminal attributes	Using iRMX system calls to get the current terminal attributes. After getting and altering the attributes, you can use another iRMX system call to set them.
Creating tasks	Using an iRMX system call to create additional tasks from an existing task.
Cataloging objects	Describing to the system where key objects the job uses reside. Tasks can easily share cataloged objects.
Using response pointers during inter-task communication	Instructing server tasks where to respond with information that signals the completion of a request task. Response pointers allow server tasks to keep track of which request tasks they are responding to.
Using buffer pools	Creating areas of memory for a job that tasks can use as a common memory resource. Once a buffer pool and its buffers have been created, the system can use the memory by simply requesting and releasing buffers.
Performing screen input/output	Reading and writing data to the physical terminal screen.
Performing simultaneous input/output	Tasks performing I/O operations independent of one another. For example, one task may wait for terminal input while another task processes data and writes it to the terminal.

In-line Exception Processing

In-line exception processing provides a way for your application to handle errors generated from system calls. You can process them in-line, use the default exception handler, or assign your own exception handler. The example in this section shows how to process exceptions in-line. In order to do this, first create your own in-line exception handler routine, and then, explicitly pass control to your exception handler routine instead of to the default exception handler routines.

To get the OS to pass control to your routine instead of a default routine, reset the value of the current task's exception mode and code your tasks to call your exception handler routine.

The example uses a procedure called `set$exception` in the file *except.plm* to reset the exception mode to a value of zero. A value of zero tells the OS never to pass control to default exception handler routines. If you examine the beginning code of both the initial task and Task2, you will see that the very first executable statement is a call to the `set$exception` procedure as follows:

```
CALL set$exception(NO$EXCEPTIONS);
```

This call passes a zero value parameter (`NO$EXCEPTIONS` supplied from a literal file) to the procedure. When `set$exception` executes, it calls `get$exception$handler`, which returns exception handler information to the data structure addressed by `except$info`.

The procedure then replaces the exception mode with zero using this statement:

```
except$info.mode = except$mode;
```

The procedure then calls **set$exception$handler** to reset the exception handler information with the altered data addressed by `except$info`.

See also: **set_exception_handler** and **rq_set_exception_handler** system calls, *System Call Reference*

The technique of setting the exception mode to zero is not always desirable. You should understand managing exceptions before deciding on a specific technique.

See also: Exception handlers, Chapter 3,
Exceptional condition management, *System Concepts*

Since (with exception mode set to 0) the OS will no longer pass control to exception handler routines, your tasks must check for individual errors or provide your own in-line exception handler routine. This example uses a procedure called `error$check` in the file *except.plm* as the inline exception handler routine. Notice that in the source code for the initial task and Task2, a call to `error$check` follows every system call. This code is from *task2.plm*.

```
CALL rq$s$open (co$conn, WRITE$ONLY, 0, @status);
CALL error$check(510,status);
mail$box = rq$lookup$object (CALLER, @(3,'MBX'),
                             INFINITE$WAIT, @status);
CALL error$check(520,status);
pool$tkn = rq$lookup$object (CALLER, @(6,'BUFFER'),
                             INFINITE$WAIT, @status);
CALL error$check(530,status);
```

Each time a system call is made, a subsequent call is made to `error$check`; passing it a line number and a word containing the status from the previous system call. The routine `error$check` tests the value of `status` and returns to the calling task if it is zero (no error occurred). If the value of `status` is not zero (an error occurred), then error$check builds an error message, prints it to the screen, and exits the job.

The line numbers passed as the first parameter in calls to `error$check` have no implicit meaning. These numbers are arbitrary numbers that can be associated with a system call. This technique enables you to easily find a system call that generates an error.

Use of Literal Files

Within the iRMX directory structure are find Intel-supplied literal files. These files are located in the directory */rmx386/inc* and have a file extension of *.lit*. Literal files provide many data structure definitions used by iRMX system calls and useful literal definitions for PL/M code. Use include statements to include those literal files that apply to a code file.

These PL/M statements are from the initial task's code in the file *demo.plm*. These statements show how to include six literal files.

```
$include(/rmx386/inc/error.lit)
$include(/rmx386/inc/common.lit)
$include(/rmx386/inc/nstexh.lit)
$include(/rmx386/inc/tscrn.lit)
$include(/rmx386/inc/iaiors.lit)
$include(/rmx386/inc/io.lit)
```

Table 12-1 shows which Intel-supplied literal files are useful for various types of system calls.

Table 12-1. PL/M Literal Files for Use with iRMX System Calls

Nucleus System Call	**Literal File**
create$job	nstexh.lit
get$exception$handler	nstexh.lit
get$task$tokens	ngttok.lit
get$type	ngttyp.lit
set$exception$handler	nstexh.lit
BIOS System Call	**Literal File**
agetconnection$status	iagtcs.lit, io.lit
agetfile$status	iagtfs.lit, iotyp.lit, io.lit
a$open	io.lit
a$physical$attach$device	io.lit
a$seek	io.lit
a$special	tscrn.lit
EIOS System Call	**Literal File**
createiojob	nstexh.lit, iexioj.lit
e$create$io$job	nstexh.lit
exitiojob	iexioj.lit
get$logical$device$status	io.lit
get$logical$attach$device	io.lit
sgetconnection$status	isgtcs.lit, io.lit
sgetfile$status	isgtfs.lit, ifltyp.lit, io.lit
s$open	io.lit
s$seek	io.lit
s$special	isiors.lit, tscrn.lit
Human Interface System Call	**Literal File**
cgetoutput$connection	hgtocn.lit
cgetoutput$pathname	hgtocn.lit
message passing calls	nmesgs.lit
buffer pool calls	nbpool.lit

Aside from the literal files shown in Table 12-1, two other important literal files exist: *common.lit* and *iaiors.lit*. The file *common.lit* contains many literal declarations commonly used in PL/M programming. You should include this file in all your PL/M programs. The file *iaiors.lit* contains the structure for the I/O Result Segment (IORS) returned in most BIOS system calls. You should include this file in all your PL/M programs that make BIOS system calls.

□□□

Resource and Stack Size Guidelines A

This appendix discusses guidelines for using memory to support iRMX object types. It also discusses stack size requirements and calculations.

Resource Requirements

The Nucleus obtains memory from the calling job's memory pool when creating objects or borrowing memory. When a job borrows memory from its parent, the Nucleus uses three 16-byte paragraphs in addition to the amount it uses for object creation. Table A-1 lists the memory requirements of the iRMX OS.

Table A-1. Nucleus Memory Requirements

Object	Number of 16-byte Paragraphs Required
job	3 + object directory
object directory	1 per entry in the directory
task	5 + 6 (if the task uses the NPX) + stacksize/16 (if the Nucleus allocates the stack)
mailbox	2 + size of high performance queue/4
semaphore	2
region	2
segment	1 + segsize/16
extension	2
composite	3 + number of positions available for components/8

The BIOS obtains memory from the calling job's memory pool when creating objects. These values are shown in Table A-2.

Table A-2. BIOS Memory Requirements

Object	Number of 16-byte Paragraphs Required
I/O Result Segment	4 (5 for an internal IORS that the operating system creates when attaching a device)
Connection to named file	6
Connection to physical file	4
User object	3 (minimum)

RAM Requirements

This information helps estimate the amount of RAM needed to use the EIOS. The descriptions that follow state explicitly from which pool the RAM is taken. Use this information when deciding how large to make the memory pools of the jobs in your application.

Attaching a Logical Device

Each time one of your tasks uses the **rq_logical_attach_device** system call, the EIOS uses 98 bytes of RAM from your job's pool and 64 bytes of RAM from the pool of the EIOS job created during the configuration process. This RAM is in addition to the RAM required by the BIOS for a device connection.

Both quantities of RAM are eventually returned to the memory pools from which they originated, but they are returned at different times. The memory taken from the EIOS pool is returned only when the device is detached. In contrast, the memory taken from your job's pool is returned as soon the **rq_logical_attach_device** system call finishes running.

Creating an I/O Job

Whenever one of your tasks creates an I/O job, the EIOS uses 176 bytes of RAM from the pool of this new I/O job. This is in addition to the RAM used by the Nucleus to create the job. All of this memory returns to the pool of the parent job after the I/O job has been deleted.

In addition to the memory requirement, **rq_create_io_job** and **rqe_create_io_job** also require five entries in the object directory of the I/O job being created.

See also: Configuration, *Programming Concepts for DOS and Windows*, Memory Screens, *ICU User's Guide and Quick Reference*

Opening a Connection

When a task opens a file connection using the **rq_s_open** system call, the EIOS uses some RAM from the pool of the calling job to create objects. The amount of RAM required depends on whether the connection is opened for buffered I/O or nonbuffered I/O.

- If the connection is not buffered, the EIOS uses 64 bytes of RAM.
- If the connection is buffered, use this expression to compute the RAM size. This amount is a function of the buffer size in bytes (*S*) and the number of buffers (*N*):

  $$number\_of\_bytes = 80 + 5N + N(S + 64)$$

Regardless of whether the connection is buffered or not, all RAM returns to the memory pool when the connection is closed or deleted.

Other RAM Requirements

For system calls other than those discussed above, the EIOS has varying memory requirements. However, when you make an EIOS call, the call requires no more than:

- 300 bytes of your job's memory pool
- 400 bytes of the calling task's stack

This RAM returns to your job's pool as soon as each system call finishes.

Object Counts

You can assume that the EIOS creates no more than 10 objects during the execution of any system call.

Except in a few cases, all of these objects are deleted before the system call has finished running. The few exceptions are the system calls that explicitly create objects at the request of your application tasks, such as the **rq_s_attach_file** system call (which creates a file connection) and the **rq_logical_attach_device** system call (which creates a device connection).

Stack Size Limitations

You must know the stack size limitations depending on your application. Three primary cases are listed below and are explained in these sections:

- Tasks that create iRMX jobs or tasks
- Interrupt handlers
- Tasks to be loaded by the Application Loader or tasks to be invoked by the Human Interface

To use this information, you should already be familiar with the System Debugger (SDB), and should know which system calls are provided by the various layers of the OS. You also should know the difference between maskable and nonmaskable interrupts.

Stack Size Limitation for Interrupt Handlers

Interrupt handlers, invoked by maskable or nonmaskable interrupts, use the stack of the interrupted task. The OS assumes a maximum of 256 bytes of stack for interrupt handlers. Exceeding this maximum causes stack overflow errors.

To stay within the 256 byte limitation, restrict the number of local variables that the interrupt handler stores on the stack. For interrupt handlers serving maskable interrupts, you can use up to 20 bytes of stack for local variables. For handlers serving nonmaskable interrupts, use no more than 10 bytes. The balance of the 256 bytes is consumed by the **rq_signal_interrupt system** call and by storing the registers on the stack.

See also: Interrupts, *System Concepts*

Stack Guidelines for Creating Tasks and Jobs

When you create a task by invoking the **rq_create_task** system call, 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 designate a stack size when you create a job.

Specifying a stack size that is too small causes the task to overflow its stack. If the stack overflows, the hardware will detect the error and cause the Nucleus to invoke an exception handler. The exception handler either deletes the offending task or activates SDM. Specifying a stack size that is too large wastes memory. Ideally, you should specify a stack size that is only slightly larger (500 to 1000 bytes) than what is actually required. This also minimizes problems resulting from unforeseen situations.

These sections illustrate arithmetic and empirical techniques for estimating a task's stack size. For best results, start with the arithmetic technique and then use the empirical technique to adjust your original estimate.

If your programs are recursive, do not rely solely on either of these techniques. Stack usage in recursive routines varies because of run-time events and should be tracked carefully.

Stack Guidelines for Tasks to be Loaded or Invoked

If you are creating a task which will be loaded by the Application Loader or invoked by the Human Interface, you must specify the size of the task's stack during the bind process. These techniques will help you estimate stack size requirements.

Arithmetic Technique for Estimating Stack Size

The arithmetic technique slightly overestimates a task's stack size. Estimate the stack size by:

- Accommodating the needs of two interrupt handlers: one for maskable interrupts and one for nonmaskable interrupts.
- Allocating enough stack to satisfy the requirements of the most demanding OS layer to satisfy the requirements of all system calls used by your task.
- Fulfilling requirements of the task's code (for example, the stack used to pass parameters to procedures or to hold local variables in reentrant procedures).

Estimate the size of a task's stack by adding the amount of memory required to accommodate these factors. This section explains how to compute these values.

See also: Stack Size Limitation for Interrupt Handlers

Table A-3 shows the stack size required by a task to support the system calls of each layer. These figures include the 256 bytes required by the interrupt handlers.

Table A-3. Stack Requirements for Interrupts and System Calls

Layer	Number of Bytes Required
UDI	6000
Human Interface	5000
C libraries	5000
Application Loader	2000
Extended I/O System	2000
Basic I/O System	1200
Nucleus	800

Computing Stack Size

To compute stack size, add these numbers:

- The number of bytes required for interrupts and system calls, according to the most demanding layer you intend to use.
- The amount of stack required by the task's code. This amount is determined by looking at the information about the STACK segment in the *.mp1* map file thatBND386 produces. This usage is the result of calling local procedures and using the stack for local variables when your code is reentrant.

This sum is a conservative, but reasonable, estimate of a task's stack size requirements. For more accuracy, use the sum as a starting point for the empirical method.

Empirical Technique

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

To use this technique, change your program code to break to the monitor at the beginning and at the end of the program. Use the convention appropriate to your application for breaking to the monitor.

- When coding in C, use the `void causeinterrupt (unsigned char 3);` statement.
- When coding in PL/M, use the `CAUSE$INTERRUPT(3)` statement.
- When using ASM, use `INT3`.
- When using the Human Interface to load the application, use the **debug** command.

When SDM first receives control, fill the unused portion of the stack with a value that would not normally appear there. For example, use the SDM's **s** command to fill the remaining stack with a value of 0CCH.

Continue running the program. When SDM receives control at the end of the program, examine the stack and see how much of it still contains the value you filled in earlier. That portion was unused throughout the entire execution of the program.

Use this technique to estimate stack usage; the value you determine usually will not be exact because a typical run of the program may not take the deepest path (use the most stack) through the program. Also, a typical run may not encounter interrupts on these paths.

□□□

Index

A

B

C

D

E

F

G

H

I

L

M

N

O

P

R

S

T

U

V

W

X