intel®

System V/iRMK™ Installation and User's Guide

Order Number: 467241-001

Intel Corporation
3065 Bowers Avenue
Santa Clara, California 95052-8126

In the United States, additional copies of this manual or other Intel liberature may be obtained by writing:

Literature Distribution Center
Intel Corporation
P.O. Box 7641
Mt. Prospect, IL 60056-7641

Or you can call the following toll-free number:

1-800-548-4725

In locations outside the United States, obtain additional copies of Intel documentation by contacting your local Intel sales office. For your convenience, international sales office addresses are printed on the last page of this document. Contact your local sales office to obtain the latest specifications before placing your order.

Intel Corporation makes no warranty of any kind with regard to this material, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. Intel Corporation assumes no responsibility for any errors that may appear in this document. Intel Corporation makes no commitment to update nor to keep current the information contained in this document. Intel Corporation assumes no responsibility for the use of any circuitry other than circuity embodied in an Intel product. No other circuit patent licenses are implied.

Intel software products are copyrighted by and shall remain the property of Intel Corporation. Use, duplication or disclosure is subject to restrictions stated in Intel's Software License Agreement, or in the case of software delivered to the government, in accordance with the software license agreement as defined in FAR 52.227-7013.

No part of this document may be copied or reproduced in any form or by any means without prior wirtten consent of Intel Corporation. Intel Corporation retains the right to make changes to these specifications at any time, without notice.

The following are trademarks of Intel Corporation and its affiliates and may be used only to identify Intel products. (Registered trademarks are followed by a superscripted ® .)

Above	A	Intelevision	MICROMAINFRAME	SLD
ACE51	i	int$_e$ligent Identifier	MULTI CHANNEL	SugarCube
ACE96	i®	int$_e$ligent Programming	MULTIMODULE	SUPERCHARGER
ACE186	I²ICE	Intellec®	MultiSERVER	SatisFAXtion
ACE196	ICE	Intellink	NETPORT	SX
ACE960	iCEL	iOSP	ONCE	ToolTALK
ActionMedia	ICEVIEW	iPAT	OpenNET	UNIPATH
BITBUS	iCS	iPDS	OTP	UPI
Code Builder	iDBP	iPSC®	PRO750	VAPI
COMMputer	iDIS	iRMK	PROMPT	Visual Edge
CREDIT	iLBX	iRMX®	Promware	VLSiCEL
Data Pipeline	iMDDX	iSBC®	QUEST	WYPIWYF
DVI	iMMX	iSBX	QueX	ZapCode
ETOX	Inboard	iSDM	Quick-Erase	287
FaxBACK	Insite	iSXM	Quick-Pulse Programming	376
Genius	Intel®	Library Manager	READY-LAN	386
i486	int$_e$l®	MAPNET	RMX/80	387
i750®	Intel386	MCS®	RUPI	4-SITE
i860	int$_e$lBOS	Megachassis	Seamless	486
	Intel Certified			

IBM and PC AT are registered trademarks and PC and PC XT are trademarks of International Business Machines Corporation. XENIX, MS-DOS and Microsoft are registered trademarks of Microsoft Corporation. Ethernet is a registered trademark of Xerox Corporation. Soft-Scope is a registered trademark of Concurrent Sciences, Inc. UNIX is a registered trademark of UNIX System Laboratories, Inc.. Hazeltine and Executive 80 are trademarks of Hazeltine Corporation. TeleVideo is a trademark of TeleVideo Systems Inc. Wyse and WY-75 are registered trademarks of Wyse Technology. MetaWare and High C are registered trademarks of MetaWare, Inc. Par Lap is a trademark of Phar Lap Software, Inc.

MIX is an acronym for Modular Interface eXtension. MIX® is a registered trademark of MIX Software, Incorporated.

Rev.	Revision History	Date
-001	Original Issue	12/90

PREFACE

This manual describes how to use Release I.3 of the System V/iRMK™ Kernel product. The Kernel system calls and other interfaces are described in the *iRMK™ Kernel Reference Manual.*

Reader Level

This manual is intended for programmers familiar with system development in a UNIX environment. The manual assumes familiarity with the Intel386™ family of processors and with Multibus II architecture.

Manual Organization

This manual consists of eight chapters and four appendices:

- Chapter 1 gives an overview of the Kernel.
- Chapter 2 describes installing software packages and configuring the debugger.
- Chapter 3 describes example software included with the Kernel.
- Chapter 4 describes the development process for various languages and memory models.
- Chapter 5 describes how to bootload an application.
- Chapter 6 describes using the basic Kernel features in your application.
- Chapter 7 describes optional Kernel features; support for these is transparently included if your application uses them.
- Chapter 8 discusses the types of I/O supported by the Kernel, with an overview of using the C library functions.
- Appendix A describes the differences between Releases I.2 and I.3 of the System V/iRMK Kernel.
- Appendix B gives an overview of putting a Kernel application in ROM.
- Appendix C describes how to boot System V/386 if you accidentally corrupt the bootstrap configuration file.
- Appendix D describes the syntax used to invoke Intel tools under System V/386.

Conventions

Most calls to the Kernel begin with the standard prefix KN_. When referring to the system calls in text, this manual uses a shorthand notation that omits the prefix. You must include the prefix when specifying the system calls in your code.

System calls and command line entries are listed in **bold** when they appear in the text of this manual.

In some examples of commands entered at a terminal, this manual uses the notation **<CR>** to indicate pressing the <ENTER> or <RETURN> key at the end of a line. An instruction to "enter" a command implies pressing <ENTER> at the end of the line.

Related Publications

The following documents contain information about subjects discussed in this manual. They are available through your local Intel sales office:

- *386™ DX Microprocessor Programmer's Reference Manual*
  order number 230985
- *80386 System Software Writer's Guide*
  order number 231499
- *ASM386 Assembler Language Reference Manual*
  order number 480251
- *C: A Reference Manual*
  order number 555107
- *Debugging System V/iRMK™ Applications*
  order number 467236
- *FORTRAN-386 Compiler User's Guide*
  order number 481837
- *Firmware User's Guide for Multibus II System Architecture (MSA) Firmware*
  order number 506090
- *iC-386 Compiler User's Guide*
  order number 483326
- *Intel386™ Family System Builder User's Guide*
  order number 481342
- *Intel386™ Family Utilities User's Guide*
  order number 481343
- *iRMK™ Kernel Reference Manual*
  order number 467231

- *Microcomputer Components SAB 82258 Advanced DMA Controller for 16-Bit Microcomputer Systems (ADMA) User's Manual* 11.85. Munchen, Federal Republic of West Germany: Siemens AG, Bereich Bauelemente, Produkt-Information. Order number B2-B3372-X-X-7600
- *MPC User's Guide*
  order number 176526
- *Multibus II Interconnect Interface Specification*
  order number 149299
- *Multibus II Transport Protocol Specification and Designer's Guide*
  order number 453508
- *Peripherals Handbook*
  order number 296467
- *PL/M Programmer's Guide*
  order number 452161
- *Soft-Scope® III Debugger User's Guide*
  Target: 386™ with System V/iRMK™ Kernel
  Host: System V/386
  order number 467246
- *System V/iRMK™ C Libraries*
  order number 467226
- *Intel® System V/386 Multibus Reference Manual*
  order number 463328

For information about non-Intel tools discussed in this manual, contact the following companies:

- Phar Lap Software Incorporated
  60 Aberdeen Avenue, Cambridge, MA 02138
  Phone: (617) 661-1510
- MetaWare Incorporated
  2161 Delaware Avenue, Santa Cruz, CA 95060-5706
  Phone: (408) 429-6382

CONTENTS

Chapter 1. Overview of the System V/iRMK™ Kernel

Chapter 2. Installation and Configuration

Chapter 3. Example Programs

Chapter 4. Program Development

Chapter 4. Program Development (continued)

Chapter 5. Bootloading the Application

Chapter 6. Using Basic Kernel Services

Chapter 6. Using Basic Kernel Services (continued)

Chapter 7. Using Optional Kernel Services

Chapter 7. Using Optional Kernel Services (continued)

Chapter 7. Using Optional Kernel Services (continued)

Chapter 7. Using Optional Kernel Services (continued)

Chapter 8. Standard I/O and the C Libraries

Appendix A. Differences Between Releases I.2 and I.3

Appendix B. Putting Code in ROM

Appendix C. Booting System V with a Bad Configuration File

Appendix C. Booting System V with a Bad Configuration File (continued)

Appendix D. Syntax for Invoking Intel Tools

Index

Service Information ..... Inside Back Cover

Tables

Figures

Figures (continued)

OVERVIEW OF THE SYSTEM V/iRMK™ KERNEL 1

Introduction

The System V/iRMK™ Kernel provides an environment for developing real-time applications using UNIX System V/386. The Kernel runs on one or more boards (hosts) in a Multibus II system and provides the real-time interface between hardware and the application. The developer uses the System V/386 operating system, running on one host in the system, to develop and debug the application. In its final form, the application may use System V as a human interface, or the application and Kernel may be embedded in the system without including System V.

Developing an application in this fashion allows a single Multibus II system to be used as both the host and target for development, using the multiprocessing capabilities of the system. Figure 1-1 illustrates using the System V host for development, with the other processor boards in the system used as real-time targets.

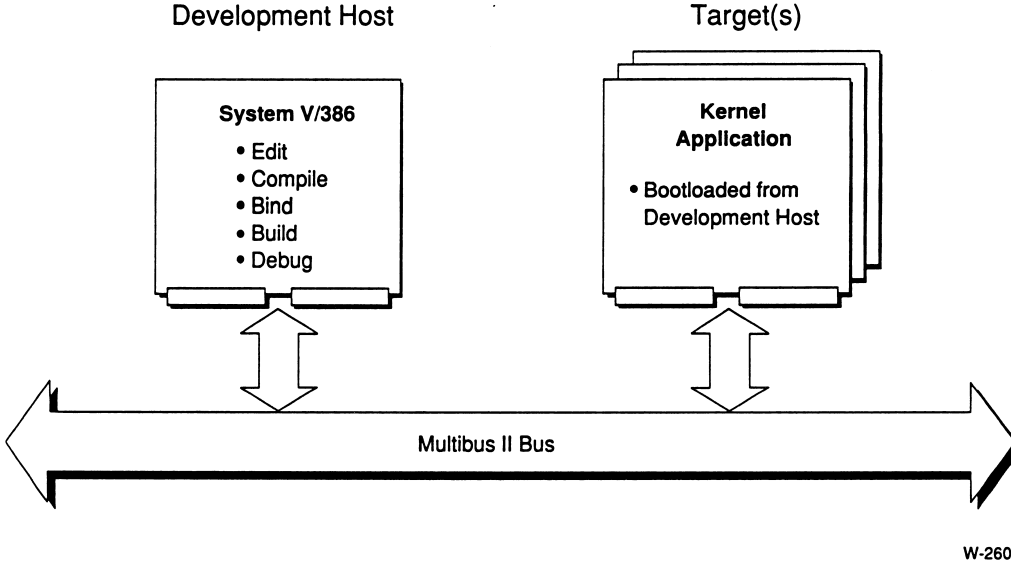


Figure 1-1. Developing the Real-Time Application

In addition to using System V for development, the application can be divided into real-time elements that run on the Kernel and less time-critical elements that run under System V. Using the Kernel as the real-time manager adds real-time capabilities to System V without impacting the UNIX environment. The System V and Kernel hosts execute their own processes concurrently, communicating via the message-passing protocols integral to Multibus II.

Figure 1-2 illustrates how the application might be divided between a host running System V and hosts managed by the Kernel. The C Server and C Libraries shown in Figure 1-2 are included in the Kernel package. The Kernel application calls C library functions for I/O capability. The library functions communicate with the C Server running on the System V host to perform the I/O. Chapter 8 describes this capability in more detail.

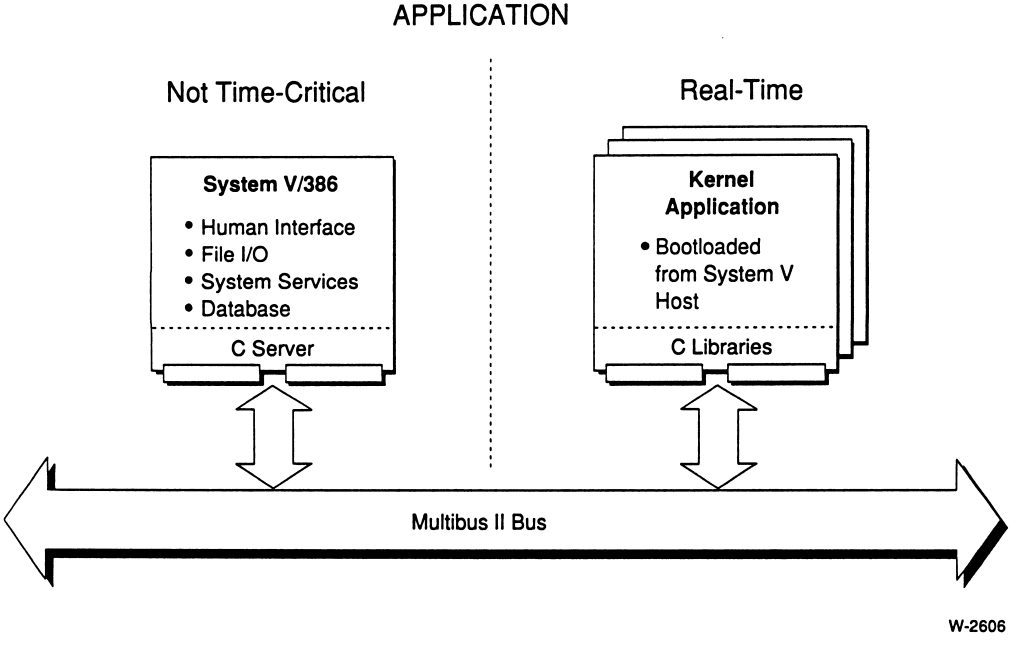


Figure 1-2. Application Running on System V and Real-Time Hosts

The Kernel can manage any board based on the Intel386™ family of processors, including the 386™DX, 386SX™, and 486™ processors, as well as the 376™ embedded processor. The Kernel includes standardized interfaces to such board devices as the programmable interrupt controller (PIC), programmable interval timer (PIT), direct memory access (DMA), and serial communications device. Although the standard device drivers are designed for Intel® boards, the developer can modify the handlers or write additional device drivers to support custom Multibus II boards.

The Kernel's real-time features include:

- support for interrupt handlers and/or interrupt tasks written by the developer
- multiple task support, with provisions for task handlers written by the developer to customize the Kernel's task scheduling
- software alarms that invoke alarm handlers written by the developer
- semaphores for synchronization
- region semaphores for mutual exclusion
- mailboxes and ports for communication between processes
- control of Multibus II message-passing for communication between hosts

Changes in Release I.3

There are two major changes between Release I.2 and Release I.3 of the System V/iRMK Kernel:

- Object format: in Release I.3, application object files must be OMF386 format, rather than the Common Object File Format (COFF) supported in Release I.2
- Debugging: in Release I.3, the debug interface is Soft-Scope III, rather than the *rdb* interface supported in Release I.2

Appendix A lists the differences between releases and describes the steps necessary to move an application to Release I.3.

Producing OMF386 code requires a different set of development tools (compiler, linker, etc.) than provided with System V. Chapter 4 describes development with Intel and non-Intel tools that produce OMF386 code.

The Soft-Scope III debugger provides source-level debugging and a generally better debugging environment than rdb. Chapter 2 describes configuring the debugging environment to enable Soft-Scope III. The *Soft-Scope® III Debugger User's Guide* describes the debugger interface. The *Debugging System V/iRMK™ Applications* manual describes the Kernel debugging support and additions to Soft-Scope III.

Languages and Models Supported

An application can be written in any of several languages, and can use the compilation model best suited to the kind of application. Modules written in assembler access the Kernel directly. The Kernel also provides interface libraries for applications written in the following languages:

- C
- PL/M
- FORTRAN

Either of the following segmentation models can be used. The model chosen affects Kernel performance and the protection available to the application:

- Small model offers the highest performance and the least protection. The developer can choose to implement paging (virtual memory managed by the processor) for more protection, but at the cost of performance.
- Compact model (or higher) offers more protection, but at a varying cost in performance, depending on how the application is written. The developer may choose to divide the application into the multiple privilege rings supported by the processor architecture.

Chapter 4 describes specific details for developing an application with these various languages and application models.

***

INSTALLATION AND CONFIGURATION 2

The Packages to Be Installed

This chapter describes installation of three packages: the Kernel files, the Soft-Scope III debugger, and a set of Intel development tools that includes the iC-386 compiler. There is a brief description of the directories created and the files installed.

You are likely installing both the Kernel and Soft-Scope III. The development tools package is not mandatory: you may use other compilers and/or utilities (Chapter 4 describes the tools evaluated with the Kernel). Follow only the instructions in this chapter that apply to the package(s) you install.

If you install the Soft-Scope III debugger, you must configure some elements of Kernel software to enable debugger communication. The configuration is described in this chapter as well as in the *Debugging System V/iRMK™ Applications* manual.

Before you begin the installation, make sure you have received the complete contents of the package(s) you will install. The packages are described in the following sections.

Contents of the Kernel Package

The Kernel package includes:

- A tape containing Kernel software and the UNXUDI utility for using Intel development tools in the UNIX environment
- *System V/iRMK™ Installation and User's Guide* (this manual)
- *iRMK™ Kernel Reference Manual*
- *System V/iRMK™ C Libraries* manual

Kernel software is complete with example files and source files for development on standard or custom boards When you install Kernel software, the installation creates a number of directories on your disk and copies files to the directories. Most Kernel files are installed under the */usr/intel/rmk* directory. A bootstrap loader file is installed in the */msa* directory. These directories are created if they do not exist.

The UNXUDI utility is needed to run Intel development tools, such as the iC-386 compiler and the BLD386 Builder. These tools are produced for the MS-DOS environment; they require the UNXUDI utility to run under System V. UNXUDI is installed in the */usr/bin* directory, overwriting any earlier version installed there. A set of scripts for invoking Intel tools is installed in the */usr/intel/bin* directory.

Contents of the Soft-Scope III Package

The Soft-Scope III package includes:

- A 5 1/4" high-density diskette containing the software to be installed
- *Soft-Scope® III Debugger User's Guide*
- *Debugging System V/iRMK™ Applications* manual

Soft-Scope III provides symbolic, source-level debugging of your application, and of Kernel objects and activities. Install the Soft-Scope III debugger after installing the Kernel. The Soft-Scope III files are placed in the */usr/intel/rmk/bin* and */usr/intel/rmk/ss* directories, which are created during the Kernel installation.

Contents of the Development Tools Package

A package of Intel development tools is available on a single tape suitable for installation under UNIX. The tape contains a variety of tools, and is shipped with the corresponding manuals. The package includes:

- iC-386 C compiler, with *iC-386 User's Guide* and *C: A Reference Manual*
- ASM386 Assembler, with *ASM386 Assembly Language Reference Manual*
- BND386 Binder utility, with *Intel386 Family Utilities User's Guide*
- MAP386 Mapper utility (described in the *Intel386 Family Utilities User's Guide*)
- LIB386 Librarian utility (described in the *Intel386 Family Utilities User's Guide*)
- BLD386 Builder utility, with *Intel386 Family System Builder User's Guide*

The manuals for these tools may contain diskettes with installation instructions suitable for an MS-DOS environment. Do not install from these individual diskettes. The tape installation places all the tools in the */usr/intel/bin* directory and the libraries in the */usr/intel/lib* directory.

NOTE

Appendix D shows the syntax used to invoke these tools. The syntax under System V differs from the MS-DOS syntax given in the manuals.

Before Installing Software

Before proceeding with the installation, back up all files under the */msa*, */usr/bin*, */usr/intel/bin*, */usr/intel/lib*, and */usr/intel/rmk* directories (assuming all these directories exist on your system). The Development Tools installation overwrites any existing Intel tools in the */usr/intel/bin* and */usr/intel/lib* directories with the latest version of the tool. The UNXUDI package on the Kernel installation tape overwrites the *unxudi* utility in the */usr/bin* directory. If you previously used Release I.2 of the System V/iRMK Kernel, you may want to preserve the versions of tools used for COFF-based development in that release.

Correcting installpkg in System V 3.2.2

The Kernel and Development Tools packages are installed using the **installpkg** utility. In Release 3.2.2 of System V on a Multibus II system, the **installpkg** utility does not correctly install multiple packages from a single tape. To correct the problem, perform the following steps:

1. To determine which version of the System V/386 operating system you have on your system, enter

   ```
   uname -a
   ```

2. If the version of the operating system is shown as

   ```
   3.2 2.2
   ```

   make the changes shown below. Otherwise, ignore steps 3 and 4, and proceed to install the packages.

   In version 2.2 of release 3.2, the installation script sets the blocking buffer size to 50K bytes rather than the default 5K byte size. With the buffer size set to 50K, the wrong packages (not the ones you select) may be installed. To correct this problem, you must modify the installation script before invoking the **installpkg** utility. Do not change the installation script back to its original form after you are done.

3. Make a backup copy of the existing */usr/lbin/Install.tape* file.

4. As the root user, edit the */usr/lbin/Install.tape* installation script. Near line 170 in the script make the following changes:

FROM:

```
eval $MOVE $POS >/dev/null 2>&1
if [ -f /etc/conf/pack.d/kd]
then
cpio -iBcdu -I$DEVICE 2>${CPLOG}
else
cpio -icdu -C 51200 -I$DEVICE 2>${CPLOG}
fi
```

TO:

```
eval $MOVE $POS >/dev/null 2>&1
cpio -iBcdu -I$DEVICE 2>${CPLOG}
```

After this change you may proceed to invoke the **installpkg** utility for tapes as directed in the following sections.

Installing the Kernel Package

The **installpkg** interface is different for different releases of System V/386. Follow the instructions for either System V 3.2 or System V 4.0.

Under System V 3.2

1. Install the System V/iRMK Kernel package by invoking the **installpkg** command. The **installpkg** utility prompts you with the following message.

```
Are you installing from tape? [y/n]
```

2. Enter **y**. The **installpkg** utility instructs you to insert the tape and press <ENTER>. After about three minutes, the utility displays the list of packages on the tape:

```
Available Packages:

1     System V/iRMK I.3 Package
2     UNXUDI Intel UDI Driver and Utility Package

Do you want to install all of the above packages? <y/[n]>:
```

3. Enter **y** to install the packages.
4. When the shell prompt returns, remove the Kernel installation tape.

Under System V 4.0

1. Install the System V/iRMK Kernel package by invoking the **installpkg** command. The **installpkg** utility prompts you with the following message.

```
Please indicate the installation medium you intend to use.

Strike "C" to install from CARTRIDGE TAPE
or "F" to install from FLOPPY DISKETTE.

Strike ESC to stop.
```

2. Enter **C** to install from tape. The **installpkg** utility instructs you to insert the tape and press <ENTER>.

3. You will be asked whether to retension the tape, and given instructions for selecting packages to be installed. Enter your answers to the prompts.

   Then the utility displays the list of packages on the tape:

```
Packages available for installation:

1     System V/iRMK I.3 Package
2     UNXUDI Intel UDI Driver and Utility Package

3.    Install ALL packages shown above
4.    Exit, do not install any packages

Please enter the next package number(s) to install,
followed by ENTER.

Press ESC when all selections have been made.

Enter Package Number:
```

4. Enter **3** to select both packages. Then press <ESC> to end the selection process.

5. You will receive further prompts to confirm the selection. Press <ENTER> to respond to the prompts and begin the installation.

6. When the shell prompt returns, remove the Kernel installation tape.

Installing Soft-Scope III

The Soft-Scope III files are on a single, high-density diskette. The files are in cpio format, but are not installed with the **installpkg** utility. Install Soft-Scope III after installing the Kernel, because the files are placed in directories created during Kernel installation.

To install Soft-Scope III, place the diskette in the drive and enter the following commands:

```
cd /
cpio -iBcuvd </dev/dsk/f0q15dt
```

NOTE

The installation instructions assume you are installing Soft-Scope III on a system like Intel's System 520, which has a single 5 1/4" diskette drive. If your 5 1/4" drive is not the first diskette drive in the system, substitute the appropriate device name in the cpio command above.

Installing the Development Tools Package

The instructions in this section describe installation of the tape containing the C compiler and other Intel utilities, including the Binder and Builder. The **installpkg** interface is different for different releases of System V/386. Follow the instructions for either System V 3.2 or System V 4.0.

Under System V 3.2

1. Install the development tools tape by invoking the **installpkg** command. The **installpkg** utility prompts you with the following message.

```
Are you installing from tape? [y/n]
```

2. Enter **y**. The **installpkg** utility instructs you to insert the tape and press <ENTER>. After about three minutes, the utility displays the list of packages on the tape:

```
Available Packages:

1     Intel UDI 386 Development Tools Package

Do you want to install all of the above packages? <y/[n]>:
```

3. Enter **y** to install the tools.
4. When the shell prompt returns, remove the development tools installation tape.

Under System V 4.0

1. Install the development tools tape by invoking the **installpkg** command. The **installpkg** utility prompts you with the following message.

```
Please indicate the installation medium you intend to use.

Strike "C" to install from CARTRIDGE TAPE
or "F" to install from FLOPPY DISKETTE.

Strike ESC to stop.
```

2. Enter **C** to install from tape. The **installpkg** utility instructs you to insert the tape and press <ENTER>.

3. You will be asked whether to retension the tape, and given instructions for selecting packages to be installed. Enter your answers to the prompts.

   Then the utility displays the list of packages on the tape:

```
Packages available for installation:


1     Intel UDI 386 Development Tools Package

2.    Install ALL packages shown above
3.    Exit, do not install any packages

Please enter the next package number(s) to install,
followed by ENTER.

Press ESC when all selections have been made.

Enter Package Number:
```

4. Enter **2** to select the package. Then press <ESC> to end the selection process.

5. You will receive further prompts to confirm the selection. Press <ENTER> to respond to the prompts and begin the installation.

6. When the shell prompt returns, remove the development tools installation tape.

Installing Other Development Tools

Intel tools are typically produced for the MS-DOS environment, and run on UNIX using the UNXUDI package (Universal Development Interface for UNIX), which is installed from the Kernel tape.

In addition to the Development Tools package described earlier, you may separately install the PL/M-386 or FORTRAN-386 compiler from diskette. Install the files using the System V **mread** utility, which reads diskettes in MS-DOS format. Make sure the files are installed with lower-case filenames. Install the compiler executable files (**.exe*) in directory */usr/intel/bin*. Install the compiler library files in directory */usr/intel/lib*. Since the Kernel example files for Intel tools invoke the tools without a preceding pathname, make sure this directory is on your search path.

When UNXUDI is installed, it places scripts for invoking PL/M and FORTRAN in the */usr/intel/bin* directory. Refer to Appendix D for the syntax used to invoke these compilers under System V.

If you choose to install non-Intel tools, use the installation instructions provided with the tool. Make sure the directory where you install the tools is on your search path. The Kernel examples for non-Intel tools invoke the tools without a preceding pathname. However, the libraries are assumed to be in the Kernel's *isv* directory, for example, */usr/intel/rmk/isv/MetaWare/small/hce.lib*. If you install non-Intel tools in a different location, modify the pathnames in the example makefiles and link files for these tools. The example files are described in Chapter 3.

Contents of Installed Kernel Directories

This section describes the contents of the directories installed with the Kernel and Soft-Scope III packages.

/msa/stage2.rmk
: This file is a second-stage bootstrap loader for loading Kernel applications on a real-time host. Chapter 5 describes how to use this file.

/usr/intel/rmk
: This is the main directory containing the files for the Kernel product. The following subdirectories are contained under this directory:

bin
: This directory contains executable files needed to develop or use the Kernel software. The Soft-Scope III executable files and help files are installed here. This directory should be on your search path.

debug
: This directory contains debugger-related files that are not part of Soft-Scope III:

  bp
  : This directory contains boot parameter libraries.

  m
  : This directory contains the iM III Monitor libraries.

  rds
  : This directory contains the Remote Development Server (RDS) libraries.

  The *bp*, *m*, and *rds* directories each contain the following subdirectories:

  inc
  : This directory contains include files.

  lib
  : This directory contains libraries.

  src
  : This directory contains source code.

  obj
  : This directory contains object files.

examples
: This directory contains tested examples to use as guides when developing your own application. The examples are described in Chapter 3.

isv
: This is an empty directory you may use to install compilers and utilities by non-Intel vendors (Independent Software Vendors, or ISVs). Some Kernel examples for ISV tools assume that the tools are installed in this directory.

ss
: This directory contains the Soft-Scope III macro files and sample files.

system This directory has subdirectories that contain the Kernel libraries, include files, and source code:

inc This directory contains the include files for the Kernel product.

include This directory contains the C library header files. The C library files are described in the *System V/iRMK™ C Libraries* manual.

lib This directory contains the libraries and link files for the Kernel.

src This directory contains source code for the Kernel product, including assembly modules as well as include files, literal files, and macro files used by the source.

Configuring the Debugger

There are several pieces of software involved in debugging your application. Each must be configured or loaded properly to work with the other. Figure 2-1 shows the debugging software, which includes:

Soft-Scope III This is the high-level debugger you interact with. It runs on the System V host.

Remote Development Monitor (RDM)
RDM runs on the System V host and passes messages back and forth between Soft-Scope III and the real-time target. RDM uses a configuration file to determine the location of the real-time host and the method used to convey messages. RDM must be invoked before Soft-Scope III.

Remote Control Interface (RCI)
RCI provides a pseudo-serial link between hosts in a Multibus II chassis. It actually uses Multibus II message passing to transfer information, and is substantially faster than an RS-232 serial link. RCI runs on the System V host and depends on a configuration file to determine read and write pipes to the real-time hosts. If used, RCI must be invoked before Soft-Scope III.

Remote Development Server (RDS)
RDS runs on the real-time host and is the interface between RDM and the iM III Monitor described below. RDS is supplied as a pre-bound file, *rds.lnk*, which includes the iM III Monitor. The *rds.lnk* file must be bound and built with your application (as described in Chapter 4) to enable debugging. After the application is debugged, you may remove *rds.lnk* from the final bind and build.

iM III Monitor This is a monitor package that interrogates the hardware on the real-time host, in response to Soft-Scope III commands. Soft-Scope III sends commands to iM III and formats the reply for interactive use. The iM III Monitor includes a console interface that can use either RCI or a serial device on the real-time host to communicate with Soft-Scope III.

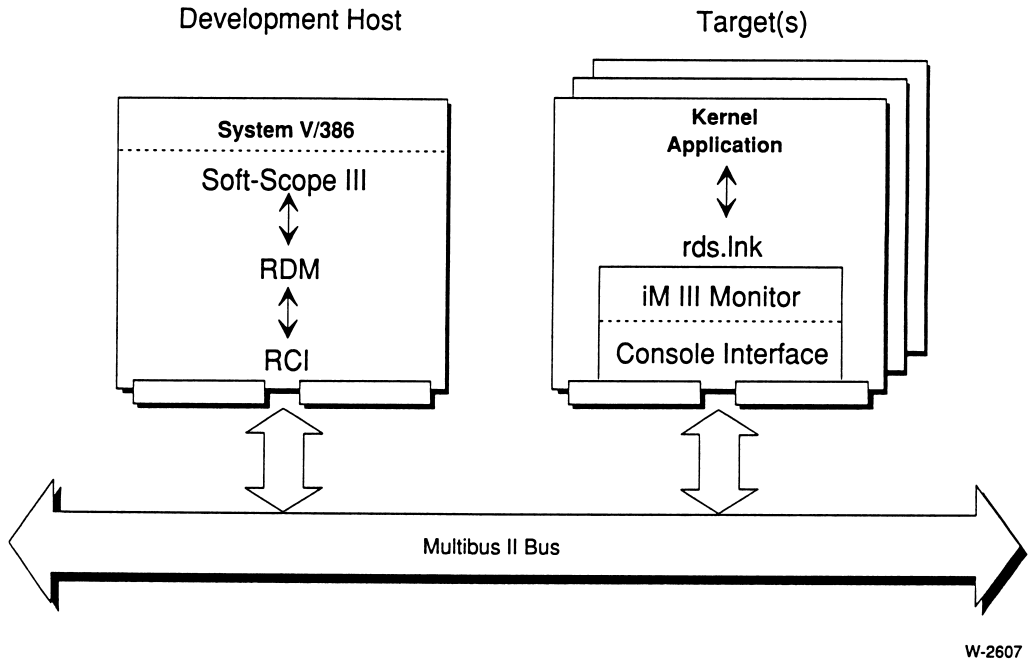


Figure 2-1. Kernel Debugging Software Used with Soft-Scope III

The executable files for Soft-Scope III, RDM, and RCI are installed in directory */usr/intel/rmk/bin*. The example commands listed in this chapter assume you have that directory on your search path.

To configure the debugger, perform the following steps.

1. Decide whether you want to use an RS-232 link and, if so, connect the appropriate cable.
2. Configure RDM.
3. Configure RCI, unless you use an RS-232 link for communications.
4. Establish appropriate BPS parameters in the bootstrap configuration file.
5. Establish an *ss.set* file that tells Soft-Scope III which real-time host to use and, optionally, the file you want to debug.

Choosing Between RCI and an RS-232 Link

You may use either the RCI interface or an RS-232 connection from the System V host to the real-time host. Typically, both hosts are in a single Multibus II chassis, and the developer uses RCI for debugging. You may, however, establish an RS-232 link between two hosts in a single chassis, rather than using RCI.

If you use a separate chassis for the System V debugging host (this would not have to be a Multibus II system), you must connect to the target real-time host with an RS-232 serial cable. In this case, you install Soft-Scope III and RDM on the debugging host, and do not use RCI. You must still use a System V host in the target chassis to store the bootloadable application.

You may establish both an RCI link and an RS-232 link to a real-time host, although you can only use one at a time. If you don't use an RS-232 link, proceed to the next section, Configuring RDM.

To use an RS-232 link, you must attach a cable between the System V and real-time hosts, make appropriate entries in the RDM configuration file, and set a BPS parameter. For information about serial devices on real-time hosts, see Table 2-1, on page 2-21. See the System V documentation for information about serial devices on the System V host. To determine the cable connections, refer to the hardware reference manuals for the boards in your system.

Configuring RDM

To configure RDM, edit file */usr/intel/rmk/bin/rdmcfg*. There can be two types of entries in the *rdmcfg* file: RCI entries and RS-232 entries. Figure 2-2 shows an example *rdmcfg* file. The first set of entries is for RCI. The final entry is for an RS-232 link. You only need to make the entries appropriate to the communications method used. If you reconfigure the file after having invoked RDM, you must stop and restart RDM for the changes to take effect. When changing RCI entries in this file, you must also stop RCI (if it is running), reconfigure it, and restart it.

The names entered in the *rdmcfg* file are the names you will use for identifying the real-time hosts to Soft-Scope III.

```
#
#Name  Mode          RCI Write pipe       RCI Read pipe
#-------------------------------------------------------
rci3  staticpipe  /usr/tmp/rcislot3.w  /usr/tmp/rcislot3.r
rci4  staticpipe  /usr/tmp/rcislot4.w  /usr/tmp/rcislot4.r
#
#
#Name    Mode     Device     Baud
#-------------------------------
board4  static  /dev/tty354  9600
```

Figure 2-2. Example rdmcfg File

RCI Entries in the rdmcfg File

The format for an RCI entry in the *rdmcfg* file is

Target_Name *Mode* /usr/tmp/rcislot*X*.w /usr/tmp/rcislot*X*.r

Target_Name This is an arbitrary name you give to a real-time host. *Target_Name* cannot be longer than 16 characters.

Mode Specifies the debugger mode. Always enter `staticpipe`.

/usr/tmp/rcislot*X*.w /usr/tmp/rcislot*X*.r
Specifies UNIX write and read pipe files. Replace *X* with a decimal value indicating the slot number of the real-time host you wish to communicate with. Each entry has a corresponding entry in the RCI configuration file.

RS-232 Entries in the rdmcfg File

RS-232 entries in the *rdmcfg* file specify the System V serial port to which the serial cable is connected. A separate entry is required in the *rdmcfg* file for each serial port used. The format for an RS-232 entry is

Target_Name Mode Device_Name

Target_Name This is an arbitrary name you give to a real-time host (even though this entry specifies the System V end of the serial cable, it is useful to refer to the connection by a name that identifies the target). *Target_Name* cannot be longer than 16 characters.

Mode Specifies the debugger mode. Always enter `static`.

Device_Name Enter the name of the device driver controlling the serial port on the System V host. *Device_Name* cannot be longer than 32 characters. The example shown in Figure 2-2 is */dev/tty354*. This driver controls port J1 on an iSBX 354™ module attached to the System V host.

Baud Specifies the baud rate used. The value you enter here must match the value specified for BPS parameter **CC_console**, which controls the device used on the target end of the cable (see BPS Parameters for the Debugger, in this chapter). Typically, System V sets the baud rate for serial ports on the host to 9600; this cannot be changed. This also applies to an iSBX 354 module attached to the System V host. With the */dev/tty354* driver, you must use 9600 baud.

Configuring RCI

If you use only an RS-232 link, you don't need to configure RCI; proceed to the next section, BPS Parameters for the Debugger.

To configure RCI, edit file */usr/intel/rmk/bin/rcicfg*. Entries in this file match RCI entries in the *rdmcfg* file. Any time you change RCI entries in *rdmcfg*, you must also change them in *rcicfg*. The *rcicfg* file provides the RCI driver the names of the UNIX read/write pipe files and the corresponding slot number of the real-time host. Figure 2-3 shows an example *rcicfg* file. Notice that the read and write pipe files are listed in the opposite order from those in the *rdmcfg* file (Figure 2-2). This is because the RDM write pipe is a read pipe to RCI, and vice-versa.

```
3  /usr/tmp/rcislot3.r  /usr/tmp/rcislot3.w
4  /usr/tmp/rcislot4.r  /usr/tmp/rcislot4.w
```

Figure 2-3. Example rcicfg File

The format of an entry in the *rcicfg* file is

Slot_Number /usr/tmp/rcislot*X*.r /usr/tmp/rcislot*X*.w

Slot_Number Specifies the slot number of the real-time host. Enter *Slot_Number* as a decimal value.

/usr/tmp/rcislot*X*.r /usr/tmp/rcislot*X*.w
Specifies UNIX read and write pipe files. Replace *X* with a decimal value indicating the slot number of the real-time host you wish to communicate with. These entries correspond to entries in the *rdmcfg* file.

Setting BPS Parameters for the Debugger

You must establish two BPS parameters for each real-time host where you debug an application. You set the BPS parameters in the bootstrap configuration file; editing this file is described in Chapter 5. Make sure to enter the parameters for each real-time host defined in the file. The BPS parameter names are `BL_debug_on_boot` and `CC_console`. The application call **initialize_RDS** (described in the *iRMK™ Kernel Reference Manual*) looks up the settings of these BPS parameters and acts accordingly.

NOTE

The bootstrap configuration file in System V 3.2 is the */etc/default/bootserver/config* file. In System V 4.0, the bootstrap configuration file is the */stand/config* file.

BL_debug_on_boot

The format of this entry in the bootstrap configuration file is

```
BL_debug_on_boot = iM;
```

This parameter specifies whether or not RDS is initialized and control is transferred to the debugger after a real-time host boots. `BL_debug_on_boot` can be set to one of the following values:

`iM` RDS is initialized and control is transferred to the iM III Monitor after the real-time host boots. When you invoke Soft-Scope III for this host, you can begin issuing debugger commands.

`on` RDS is initialized and the application starts executing after the real-time host boots. Control is only transferred to iM III if a task generates an Interrupt 3. When you invoke Soft-Scope III for this host, you cannot begin issuing commands until iM III has control.

If `BL_debug_on_boot` is set to any other value (for example, `off`), RDS is not initialized and the application cannot be debugged with Soft-Scope III.

CC_console

The format of this entry in the bootstrap configuration file is

```
CC_console = ccrci;
```

This parameter specifies the communications method for a real-time host. `CC_console` can be set to one of the following values.

`ccrci` The RCI driver is used.

`device [, baud]`

An RS-232 link is used. This entry establishes the device driver used on the real-time host, and the optional baud rate. If no baud rate is specified, the default for that device is used. Make sure that the baud rate you specify here matches the baud rate entered for this host in the *rdmcfg* file. Table 2-1 shows device names that can be used, the boards to which they apply, the default baud rates, and the maximum baud rate that can be expected to give satisfactory results. The baud rate may be limited by the port on your System V host.

Table 2-1. Serial Devices Used with the CC_console Parameter

Device	Board	Default Baud	Max Baud
cc354a	Port J1 of iSBX 354 on real-time host	9600	9600
cc354b	Port J2 of iSBX 354 on real-time host	9600	9600
cc8751	Front panel of iSBC® 386/116 or 386/120	2400	4800
ccfpa	Port J2 of iSBC 386/133 or 486/125	9600	38400
ccfpb	Port J1 of iSBC 386/133 or 486/125 or Front panel of iSBC 386/258	9600 9600	38400 19200
cc450_*n*_0 cc450_*n*_1 cc450_*n*_2 • • cc450_*n*_B	MIX450, where *n* is 0, 1, or 2, depending on the stack location of the MIX module	9600	38400
cc560_*n*	MIX560, where *n* is 0, 1, or 2, as above	9600	19200

NOTE

To avoid confusion when debugging, do not use the same serial device with the `CC_console` parameter that you use for Kernel standard I/O or character I/O. Specify a different Kernel I/O device in the **initialize_console** system call. If the same device is used, Kernel I/O functions can disrupt communications with Soft-Scope III. For more information on Kernel I/O, see Chapter 8.

Using the ss.set File

Soft-Scope III uses a file called *ss.set* to determine, among other things, which real-time host you want to establish a connection to. There can be multiple *ss.set* files in multiple directories. On invocation, Soft-Scope III uses an environment variable, SSSETPATH, to establish a search path for the various *ss.set* files you want to use. If duplicate parameters exist in these files, the last one read is the one used.

There is a default *ss.set* file in directory */usr/intel/rmk/bin*, which Soft-Scope III always uses, regardless of whether you have established an SSSETPATH. This default file tells Soft-Scope III to load the Kernel macros that provide additional debugging commands. Don't change this default *ss.set* file.

Create an *ss.set* file in the directory where you expect to debug your application. Put the following entry in this file:

```
targ.dev = hostname
```

where *hostname* is the name of the real-time host entered in the *rdmcfg* file. For example, to connect to the host in slot 4 using the RCI entry shown in Figure 2-2, *hostname* would be `rci4`.

Now establish an SSSETPATH with one of the following commands. You will probably want to put the same entry in your *.cshrc* or *.profile* file.

```
setenv SSSETPATH :directory_name:                    (C shell) or
SSSETPATH = :directory_name: ; export SSSETPATH      (Bourne shell)
```

The *Soft-Scope® III Debugger User's Guide* gives more details about using SSSETPATH and making entries in the *ss.set* file.

Testing the Debugger

To test the debugger configuration, modify the bootstrap configuration file (described in Chapter 5) to load the Kernel example file */usr/intel/rmk/examples/ic386/bist.sml/abs/testss.abs*. Use the **reboot** command to reset the real-time host, and invoke the various parts of the debugger. You must start RCI (assuming you use it) and RDM before invoking Soft-Scope III. After editing the bootstrap configuration file, issue the following commands (assuming *testss.abs* is in your current directory):

```
rci start
rdm start
reboot slot_number
ss symbols testss.abs
```

The Soft-Scope III sign-on message is displayed. If Soft-Scope III cannot connect to the target, an error message appears. If the error message is not self-explanatory, refer to the following documentation:

- The *Soft-Scope® III Debugger User's Guide* describes Soft-Scope III error messages.
- The *Debugging System V/iRMK™ Applications* manual lists RDM and iM III error messages that may be returned by Soft-Scope III.

If Soft-Scope III makes the connection, the following prompt is displayed:

```
ss>
```

Use Soft-Scope III commands (described in the *Soft-Scope® III Debugger User's Guide*) to "debug" the example. To exit Soft-Scope III, enter

```
ss>quit
```

You do not need to stop RDM and RCI when you exit Soft-Scope III, but these are the commands to stop them (at the System V prompt):

```
rdm stop
rci stop
```

Once RDM is started, all users of the system are supported; no other users need to start RDM. Before stopping RDM, make sure no other sessions are using it.

RDM Invocation Error

If you start RDM and the following error message appears, either RDM is already running or it terminated by accident:

```
/usr/tmp/rdmqueue already exists, RDM is already running
```

To see if RDM is currently running, enter

```
ps -ef | grep rdm
```

If RDM is running, the following process is displayed (among others), and you may ignore the error message:

```
/usr/intel/rmk/bin/rdm
```

If RDM is not running, enter the following to stop all RDM processes and restart RDM:

```
rdm stop
rdm start
```

RCI Invocation Error

If you attempt to start RCI when it is already running, the following error message is displayed:

```
/usr/intel/rmk/bin/rci.run fatal error: rci server still
active
```

If you receive such an error and then cannot invoke Soft-Scope, enter

```
rci stop
rci start
```

***

EXAMPLE PROGRAMS 3

The Example Files

The Kernel product includes example files, complete with source code. For some of the examples, absolute (bootable) files are provided. You may use the examples as templates for application code. Study the example source files for more information about the development process described in this manual.

In the */usr/intel/rmk/examples* directory are a set of subdirectories containing examples compiled in various languages and using different models of segmentation. The main subdirectories indicate the language used. Under these are further subdirectories indicating the example and the segmentation model. The *examples* directory contains the following subdirectories:

fortran	This directory contains examples written using the FORTRAN-386 compiler.
highc	This directory contains examples written using MetaWare's High C compiler and linked with Phar Lap's LinkLoc utility.
ic386	This directory contains examples written using the iC-386 compiler.
plm	This directory contains examples written using the PL/M-386 compiler.

Under each of the above directories are further subdirectories indicating the example name and the segmentation model used to compile the example. The subdirectory filename extensions indicate the segmentation model, shown below:

.cpt	Compact model
.sml	Small RAM model
.sro	Small ROM model

Table 3-1 indicates the *examples* subdirectories and the kinds of examples provided. For instance, in the */usr/intel/rmk/examples/fortran* directory are three further subdirectories: *bist.cpt*, *bist.sml*, and *bist.sro*.

Table 3-1. Example Subdirectories in the Kernel Package

Examples	fortran			highc		ic386			plm		
	.cpt	.sml	.sro	.cpt	.sml	.cpt	.sml	.sro	.cpt	.sml	.sro
bist	x	x	x			x	x	x	x	x	x
clib						x	x				
msg_380						x			x		
msg_pass						x			x		
rom_ex						x			x		
seg_gate						x			x		
stdio				x	x	x	x		x	x	

The following subdirectories are found under each example, along with a *README* file that describes the example:

abs — When you generate an example from source, the absolute image of the example is placed in this directory. The *bist.sml* examples for PL/M and iC-386 already contain bootloadable files in the *abs* directory, which may be used to test your real-time installation.

lst — The listing and map files produced by the compiler, Builder and Binder are placed in this directory.

obj — The object and link files produced by the tools are placed in this directory.

src — The source code for the example is in this directory, along with makefiles and Builder and Binder control files.

Actions Illustrated by the Examples

The examples perform the following functions. See the example *README* files for more information.

bist — Toggles the BIST light on the real-time host, to demonstrate the example is working. This example is useful as a simple template for applications.

clib — Demonstrates remote file access using the C library functions **open**, **close**, **read**, and **write**. This example uses a C Server daemon running on System V for file access. The *System V/iRMK™ C Libraries* manual describes the C library functions provided with the Kernel.

msg_pass — Shows message passing using the Multibus II transport protocol. In this example, client tasks send read and write requests to a server task (which simulates a file server, but does not actually do file I/O). For information about message passing, see Chapter 7.

msg_380 — This is the *msg_pass* message passing example, but using the 82380 device driver. Chapter 7 describes using this device.

rom_ex — Shows how to put an application in ROM. This is the same application as the *bist* example, but in a form that can be put in ROM. For more information on embedding code in ROM, refer to Appendix B.

seg_gate — Illustrates the Kernel's gate-based interface for multiple-segment applications. For information about using gates, see the description beginning on page 4-36.

stdio — Shows Kernel-supported standard I/O based on the Kernel character I/O functions. This example uses the Kernel's 82530 serial device driver. It calls the **printf**, **scanf**, **getchar** and **putchar** functions provided in the *kstdios.lib* and *kstdioc.lib* libraries. Chapter 7 describes using the 82530 device. Chapter 8 explains the difference between the *kstdio* library functions and the same functions provided in the C libraries.

Generating the Examples

Only the *bist.sml* examples for PL/M and iC-386 already contain bootloadable files in the *abs* directory (each includes a file built with debug support). You may use one of these pre-built absolute files to test your real-time installation (see step 4, below). This section gives a brief overview of generating absolute files from the examples.

The example permissions allow only the root user to write to the directories. This prevents other users from accidentally deleting the example files from the system. To generate an example as a user other than root, perform the following steps:

1. Log on to the system.
2. Copy example files to the user workspace. Use the appropriate language path and example name from Table 3-1. For example, to copy the iC-386 compact model message-passing example to a directory named *new_dir*, you would enter:

```
copy -v -r /usr/intel/rmk/examples/ic386/msg_pass.cpt  new_dir
```

3. You may choose to modify the example source code as a basis for your own application. After modifying the source, compile, bind and build the application. Each example *src* directory contains a makefile that automates the process. There are two makefiles for each example. File *makefile* builds the application without debugger support. File *makefile.ss* builds the application with debugger support so that Soft-Scope III can be used to debug the application. Before using one of the makefiles, remove any existing files from the *obj* directory, since the object files produced with and without debug support have the same names.

   Execute the *makefile* using the **make** command. To perform the make automatically without stopping after warnings, enter the following while in the *src* directory:

```
make -i
```

   To choose the Soft-Scope III makefile, enter:

```
make -i -f makefile.ss
```

   There may be warnings from the BND386 Binder. Ignore these since the warnings will be resolved at build time. The output files will be placed in the *abs*, *obj*, and *lst* directories.

4. To run the example, edit the bootstrap configuration file and reboot the host board, as described in Chapter 5. If you use the debugger, you must first configure the debugger software, as described in Chapter 2.

***

PROGRAM DEVELOPMENT 4

This chapter covers the following subjects:

- Languages and segmentation models that can be used to develop Kernel applications
- Items to be aware of when programming the application
- Information specific to certain programming languages
- How to bind and build the application with Intel tools
- Tools from other vendors that you may use to develop the application

Introduction

Application code for the Kernel must be OMF386 format, which is the object format produced by Intel languages and utilities for the 386 processor. The developer may use compilers whose output is not OMF386, but must use a tool such as Phar Lap's LinkLoc utility to produce the final object code in OMF386.

The Kernel supports four language interfaces, including C, PL/M, FORTRAN, and Assembler. These languages can be used to produce an application in a flat or a segmented memory model. Table 4-1 lists the compilers and assemblers that have been evaluated to work with the Kernel, indicating the memory model evaluated. Applications written with other tools may work with the Kernel, but have not been evaluated by Intel.

Notice that Table 4-1 lists the small ROM model for C and PL/M under the "compact" heading. That is because this manual uses the term compact to refer to all memory models larger than small, which generally indicates far pointers rather than near. In the iC-386 and PL/M-386 small ROM models, constants are in the code segment and pointers are far. This is not true in FORTRAN.

Table 4-1. Compilers and Assemblers Evaluated with the Kernel

Compiler/Assembler	Small Model	Compact Model
Intel iC-386	small	small ROM, compact
MetaWare High C	small	compact
Intel PL/M-386	small	small ROM, compact
Intel FORTRAN-386	small, small ROM	compact
Intel ASM386	model does not apply	
Phar Lap 386|ASM	model does not apply	

Kernel applications must be bootloadable. Figure 4-1 shows the steps used to produce a bootloadable file. After compiling, the application is bound and built (linked and located) with the Kernel libraries so that the application and Kernel can be loaded onto the host board when the board or the system is booted. When using Intel tools, you compile or assemble the application modules, use BND386 to bind the output with Kernel libraries, then use BLD386 to produce a bootloadable file. The BND386 and BLD386 utilities only accept input in OMF386 format.

Most of the sections in this chapter describe the development process using BND386 and BLD386. Instead of these utilities, you may use the Phar Lap tools described in this chapter to produce the bootloadable application.

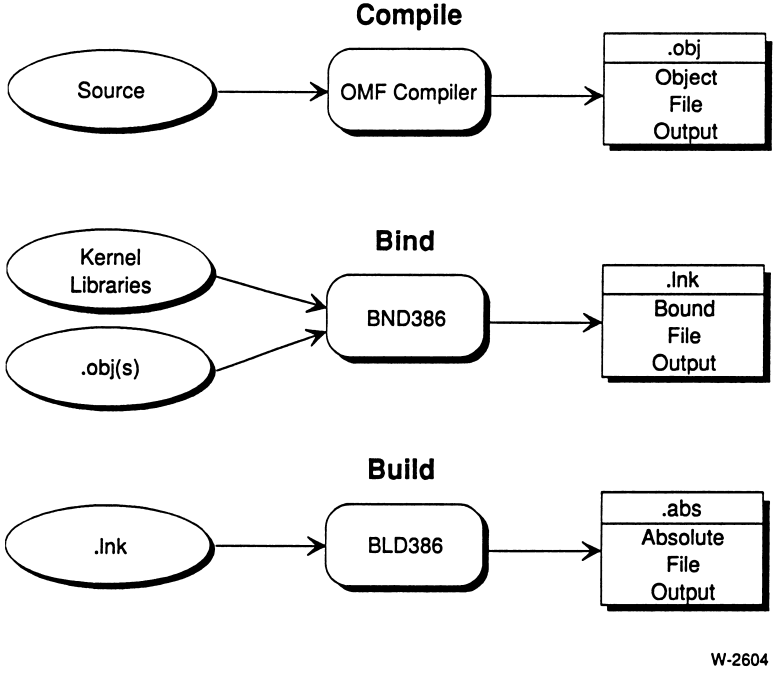


Figure 4-1. Producing a Bootloadable File

Small Model Applications

In a small model application, both application code and the Kernel share one memory segment 4 gigabytes long (16 MB for the 386SX and 376 processors). To produce a small model application, use the `small` segmentation control, with the default `ram` control. Small model applications have these characteristics:

- Constants are placed in the data segment.
- The entire system (application and Kernel) occupies a single code segment.
- There is only one data segment, which contains both data and a single stack. The SS and DS registers are set to the same selector.
- The Kernel sets the data/stack descriptor so that its limit is 4 GB, the length of linear memory. This gives the Kernel access to all of memory.
- Pointers are near, consisting of a 32-bit offset into the data/stack segment or the code segment. Pointers do not contain a segment (selector) component.
- All calls are near.
- Calls from the Kernel to handlers provided by the application are near. (The handlers must operate in ring 0, and a flag must be set telling the Kernel to make near calls.)

Because all references are near, the processor's segment registers aren't constantly being reloaded, and applications operate faster than in a segmented (compact) model. The trade-off is the lack of protection features built into the segmented architecture. All code runs in privilege ring 0. A small model application may implement processor paging for protection; the Kernel does not preclude paging nor does it provide paging support. However, paging reduces application performance, and the Soft-Scope III debugger cannot be used with paging.

Small model applications must be bound with the *c_call.lib* interface library for the Kernel. This interface uses the C calling convention. Parameters are pushed on the stack in the opposite order they appear in the call, and the caller pops parameters off the stack. For applications written in a language other than C, the Kernel provides an interface (described later in this chapter) to force the C calling convention.

The application may be written in C, FORTRAN, or PL/M using the `small` control, or in FORTRAN using `small rom`. The assembler interface is described later in this chapter.

In small model, the application makes near calls to the Kernel interface library (*c_call.lib*), and the interface library makes near calls to the Kernel. Figure 4-2 shows the flow between the application and the Kernel in small model.

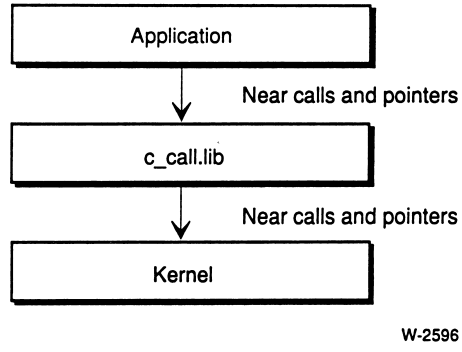


Figure 4-2. Program Flow in the Small Model

Figure 4-3 shows the steps in producing the small model application. To produce a bootloadable application, bind the application object file(s) with the Kernel libraries *c_call.lib* and *kernel.lib*. If the application calls Kernel standard I/O functions, also bind the *kstdios.lib* file (use this version of the library only in small model). Then use the Builder utility to build the bootloadable application. The bind and build operations are described later in this chapter.

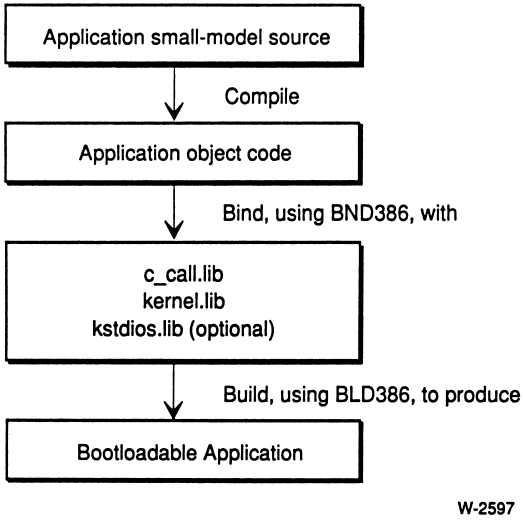


Figure 4-3. Steps in Producing a Small Model Application

Compact Model Applications

In compact model, the application can have one or more code segments, one or more data segments, and one or more stack segments. Typically, the application is compiled using the compact model, however C and PL/M applications using the small ROM model fall into this category because pointers in this model are far. Compact model applications have these characteristics:

- Calls from the application to the Kernel are near, through an interface library
- Calls to Kernel standard I/O functions are far. (Rather than explicitly make far calls in the application, you include a file that translates the calls: *kstdio.h* for C, and *kstdioc.inc* for PL/M.)
- All pointers are far, consisting of a 16-bit selector and 32-bit offset.
- Calls from the Kernel to handlers provided by the application are far. (The handlers must operate in ring 0, and a flag must be set telling the Kernel to make far calls.)

Compact model applications must be bound with the *plm_call.lib* interface library for the Kernel. This interface uses the PL/M calling convention. Parameters are pushed on the stack in the same order they appear in the call, and the called routine pops parameters off the stack. For applications written in C, you may need to specify a compiler control to force the PL/M calling convention (see Writing Applications in C, later in this chapter).

The application may be written in C, PL/M, or FORTRAN, using the `compact` control, or in C or PL/M using the `small rom` control. The assembler interface is described later in this chapter.

When the application has one code and one data segment, with one or more stack segments (one per task), the application code and data are in the same segment as the Kernel's code and data. When the application has more than one code segment, the application code and data reside in different segments from the Kernel's code and data. The following sections describe the application flow and building strategy for these two possibilities.

Single-Segment Applications

If the total application has only one code segment, operating at privilege ring 0, the flow of application to interface library to Kernel is the same as in the small model. However, a different interface library is used. Figure 4-4 illustrates the program flow for this compact model.

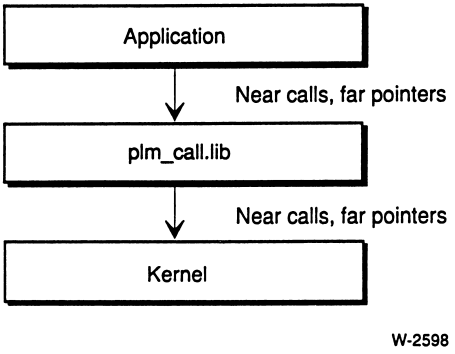


Figure 4-4. Program Flow in Single-Segment Compact Model

To produce a bootloadable application in a single ring 0 segment, follow the steps shown in Figure 4-5. Bind the object file(s) with the Kernel libraries *plm_call.lib* and *kernel.lib*. If the application calls Kernel standard I/O functions, also bind the *kstdioc.lib* file (use this version of the library only in compact model). Then use the Builder utility to build the bootloadable application.

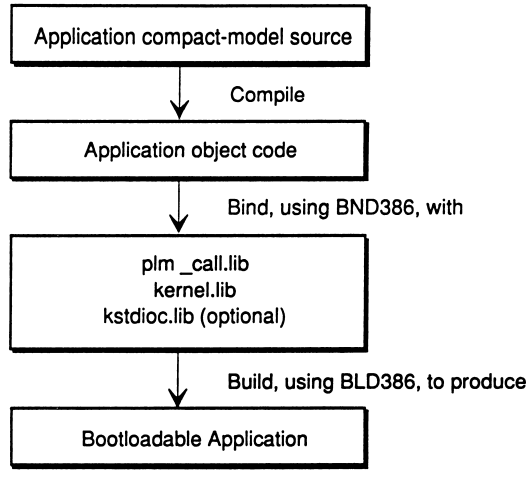


Figure 4-5. Steps in Producing a Single-Segment Compact Model Application

Multi-Segment Applications

If the application includes more than one code segment, the application makes near calls to the interface library (*plm_call.lib*). This, in turn, is bound to a false kernel called *mux.lib*, which makes far calls to the Kernel, either directly or through call gates. The Kernel has a routine that demultiplexes the user calls.

Figure 4-6 shows the flow between the Kernel and one segment of an application with multiple code segments. Each application code segment is bound to a *plm_call.lib* and a *mux.lib* in that segment. The Kernel resides in a different segment.

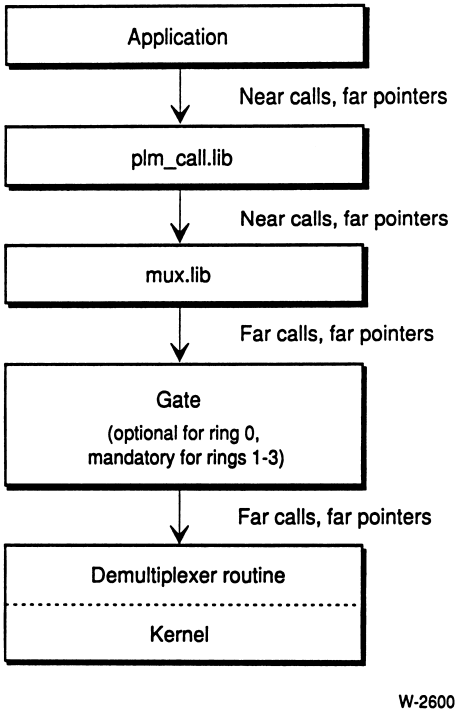


Figure 4-6. Program Flow in a Multi-Segment Compact Model Application

The compact model allows the application to take advantage of the multiple privilege rings managed by the processor. If all application segments operate at ring 0, call gates are not required between *mux.lib* and the Kernel. Any part of the application that operates in a different ring must access the Kernel through a call gate. The call gates are chosen and set up by the application.

Using multiple rings introduces a problem when applications need to share common code (for example, a Kernel *kstdio* library). This can be solved by placing the shared code in a conforming segment, allowing call gate access from code in different rings.

See also: Protection, *386™ DX Microprocessor Programmer's Reference Manual*

To produce a bootloadable, multiple-segment application, or one that uses privilege levels other than ring 0, follow the steps shown in Figure 4-7. If the application needs call gates, you may use the gates defined in the default *kn_gates.obj* file, or produce a new *kn_gates.obj* file (described later in this chapter). Bind the application object file(s) with *kn_gates.obj* and the *plm_call.lib* and *mux.lib* libraries. Separately bind the *kernel.lib* file. If the application calls Kernel standard I/O functions, include the *kstdioc.lib* file in one of these bind operations (use this version of the library only in a compact model). Then use the Builder utility to combine the output from the bind operations and build the bootloadable application.

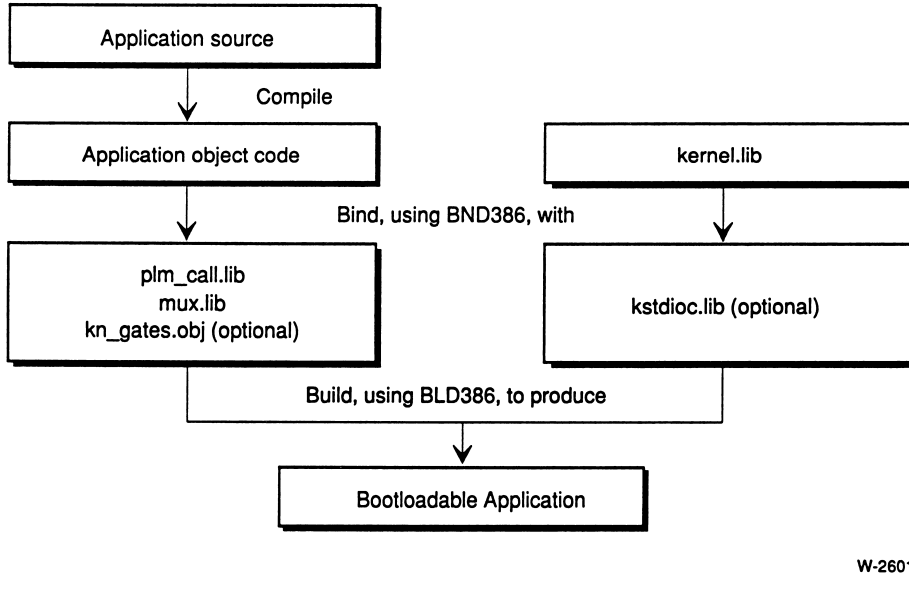


Figure 4-7. Steps in Producing a Multi-Segment Compact Model Application

General Programming Information

This section describes information needed regardless of the programming language you use. Later sections in this chapter contain additional information for specific languages.

Variable Names to Avoid

Kernel applications can be linked directly to the Kernel. Therefore, the public symbols in the code must not conflict with the public symbols the Kernel defines. To avoid problems, never define symbols that begin with the following prefixes:

KN_
KNA_
KNI_
SCH_
KNG_

When using languages that distinguish between uppercase and lowercase, do not use either the uppercase or lowercase versions of these prefixes.

Enabling Debug Support

To enable the Kernel debug support, there are two things you must do when writing and compiling your code:

- Set a compiler control that includes debug symbols in the code.
- Set up a structure with debug initialization values and invoke the **initialize_RDS** system call.

When binding and building the code, there are other requirements for debug support. These are described later in the chapter.

Compiling with Debug Symbols

To provide symbols used by the debugger, make sure to compile the application with the appropriate debug option. Intel tools use a command line option when invoking the compiler, such as the following for iC-386:

```
ic386 -debug -other_controls filename
```

The MetaWare High C compiler uses the following line, either in the source code or a profile file:

```
#pragma On (CodeView)
```

Initializing the Debugger in Code

To initialize values for the RDS part of the debugger , you must call **initialize_RDS**. This should be the first executable statement in the application. Only the code following this call can be debugged. The calling format and initialization values are shown in the *iRMK™ Kernel Reference Manual.*

The **initialize_RDS** system call sets values used by the RDS part of the debugger, and causes RDS to check debugger-related BPS parameters. If the application is not built with RDS, the call simply returns, and the debugger is not initialized. If the application is built with RDS, RDS checks the `BL_debug_on_boot` BPS parameter, with the following results:

- If `BL_debug_on_boot = iM`, the application breaks to the iM III Monitor and waits for synchronization from Soft_Scope III.
- If `BL_debug_on_boot = on`, RDS is initialized, but the application continues to run. The application does not break to the Monitor (and Soft-Scope cannot synchronize) unless an Interrupt 3 is encountered.
- If `BL_debug_on_boot` is set to any other value, RDS is not initialized and the application cannot be debugged with Soft-Scope III. The **initialize_RDS** system call has no effect on the application.

Chapter 5 describes setting the `BL_debug_on_boot` parameter in the bootstrap configuration file.

NOTE

With the `BL_debug_on_boot` parameter set to `iM`, the application stops in the middle of the **initialize_RDS** system call. The call does not return until a Soft-Scope command (**go** or **step**) starts the application running. The first executable statement you see with the debugger is the return from the **initialize_RDS** call.

Declaration and Literal Files

To produce functioning Kernel applications in the languages the Kernel supports, include the appropriate external declaration and literal files in the program source code. Table 4-2 shows the include files for each of the languages supported, and indicates the conditions under which the particular files must be included. All include files described on this page are in the */usr/intel/rmk/system/inc* directory.

Table 4-2. Include Files for Various Programming Languages

Include For	FORTRAN	PL/M	C	Assembler	Purpose
All models	rmk_ftn.inc	rmk_plm.inc	rmk.h	rmk_asm.inc	Includes files in Table 4-3
Small model	rmk_conv.ftn	rmk_conv.plm			Convert to C calling convention
Compact model, when using standard I/O		kstdioc.inc	kstdio.h		Convert kstdio functions to far calls

Table 4-3 shows the files that are automatically included when you include the files from the top line of Table 4-2. The files in Table 4-3 are part of the Kernel for compatibility with existing code. For example, if your C code already includes the files under *rmk.h* in Table 4-3, you need not include file *rmk.h*. Compilers automatically include only the code needed from include files. If you program in assembler, you may want to include only some of the files from Table 4-3, rather than including *rmk_asm.inc* (see Writing Applications in Assembler, in this chapter).

Table 4-3. Files Automatically Included

	FORTRAN	**PL/M**	**C**	**Assembler**
When you include this file	**rmk_ftn.inc**	**rmk_plm.inc**	**rmk.h**	**rmk_asm.inc**
These files are included in your code	rmk_type.par rmk_ex.par rmk_base.fun rmk_dev.fun rmk_dt.fun rmk_is.fun rmk_mp.fun rmk_base.par rmk_dev.par rmk_dt.par rmk_is.par rmk_mp.par	rmk_type.lit rmk_ex.lit rmk_base.ext rmk_dev.ext rmk_dt.ext rmk_is.ext rmk_mp.ext rmk_base.lit rmk_dev.lit rmk_dt.lit rmk_is.lit rmk_mp.lit	rmk_type.l rmk_ex.l rmk_base.h rmk_dev.h rmk_dt.h rmk_is.h rmk_mp.h rmk_base.l rmk_dev.l rmk_dt.l rmk_is.l rmk_mp.l	rmk_type.equ rmk_ex.equ rmk_base.edf rmk_dev.edf rmk_dt.edf rmk_is.edf rmk_mp.edf rmk_base.equ rmk_dev.equ rmk_dt.equ rmk_is.equ rmk_mp.equ

Aligning Allocated Memory

For performance reasons, any memory supplied to the Kernel for creating objects should be aligned on four-byte boundaries. If memory is allocated from a memory pool that is already aligned on a four-byte boundary, the memory is automatically aligned correctly. However, if memory is provided directly from the application's data segment, it may be necessary to take special steps to align the memory, for the following reasons:

- The size literals supplied by the Kernel in the literal declarations files are specified in units of bytes, causing the areas to be declared as byte arrays.
- Compilers don't necessarily align byte arrays that appear in the data segment.

To force the compiler to align arrays on four-byte boundaries, declare memory allocations as integer arrays. Since an integer is four bytes, declare one-fourth the number of array elements. For example, when declaring memory to be used by an alarm object for the **create_alarm** call, you might use the following statements in C:

```
int alarm_area [KN_ALARM_SIZE/4];
KN_create_alarm (alarm_area,...)
```

To align an 80-byte array that you need to access in byte values rather than in integer values, you might use the following statements:

```
int        y[20];
char       *x;
x = y;
```

This guideline of using an integer declaration works for all compilers evaluated with the Kernel. There are other methods that can be used instead, such as declaring the array at the beginning of a structure, or testing the alignment of the pointer and adjusting it (see the following section). If you already use another technique to align memory, make sure it still works if you change compilers.

Aligning Message-Passing Buffers

The alignment of buffers you use for message passing can affect the speed at which messages are passed across the bus. The typical Multibus II message-passing operation uses *fast mode*. Message buffers used in fast mode must be aligned on four-byte boundaries, and the data length must be a multiple of four. If not so aligned, performance of message passing is decreased. To align these buffers, use the technique described in the previous section.

Certain boards, such as the iSBC® 386/133 and 486/125 boards, have the capability of using *burst mode* for message passing. Burst mode increases throughput over fast mode. To take advantage of burst mode with these boards, buffers used for message passing must be aligned on 16-byte boundaries, and data length must be a multiple of 16 bytes.

To align a buffer on a 16-byte boundary, use the following technique:

1. When allocating memory for the buffer, include 16 extra bytes. This provides a 16-byte "window" in which you move the buffer pointer.
2. Declare a pointer to the base of the allocated memory.
3. Determine whether the buffer is on a 16-byte boundary or how far it is from the boundary, using modulo-16 arithmetic.
4. Move the pointer if necessary.

The following code illustrates these steps, aligning the pointer *msg_buffer*:

```
#define MESSAGE_SIZE 256

main ()
{
    char   buffer[MESSAGE_SIZE + 16];
    char   *msg_buffer;

    if (((int)buffer % 16) != Ø)
          msg_buffer = buffer + (16-((int)buffer % 16));
    else
          msg_buffer = buffer;
}
```

Stack Requirements

When creating a task, specify a stack to be used by the task. This stack must be large enough to handle three kinds of use. First, it must have enough room to meet the requirements of the task's code (for example, to pass parameters to procedures or to hold local variables in re-entrant procedures). In addition, when the task invokes a Kernel system call, the processing associated with the system call uses some of the task's stack. The amount of stack required depends on which system calls are used. Finally, when an interrupt occurs, the interrupt handler uses the task's stack while it services the interrupt.

When calculating the amount of stack needed for a task, add the following values:

- Amount needed by the task's code
- Amount needed by the most demanding system call the task calls
- Sum of the amounts needed by all interrupt handlers that could become active

Tables 4-4 and 4-5 list stack requirements of Kernel-provided interrupt handlers and Kernel system calls (handlers and system calls are described in the *iRMK™ Kernel Reference Manual*).

Table 4-4. Bytes of Stack Used by Interrupt Handlers

Bytes	Interrupt Handler
124	PIT handler
60	PIT handler, invoked while active
228	Message passing handler
12	NDP handler
68	level_07_handler
68	level_17_handler
68	level_27_handler

Bytes	Interrupt Handler
68	level_37_handler
68	level_47_handler
68	level_57_handler
68	level_67_handler
68	level_77_handler
68	level_M7_handler
68	level_M15_handler

Table 4-5. Bytes of Stack Used by Kernel System Calls

Bytes	System Call
44	attach_protocol_handler
108	attach_receive_mailbox
176	cancel_dl
204	cancel_tp
80	ci
80	co
40	create_alarm
124	create_area
40	create_mailbox
24	create_pool
16	create_semaphore
136	create_task
80	csts
16	current_task_token
16	delete_alarm
124	delete_area
24	delete_mailbox
16	delete_pool
16	delete_semaphore
16	delete_task
16	get_PIT_interval
8	get_code_selector
8	get_data_selector
16	get_descriptor_attributes
16	get_interconnect
16	get_pool_attributes
16	get_priority
16	get_slot
16	get_time
172	initialize
80	initialize_console
68	initialize_interconnect
40	initialize_LDT
244	initialize_message_passing
24	initialize_NDP
16	initialize_PICs
28	initialize_PIT

Bytes	System Call
256	initialize_RDS
512	initialize_stdio
16	initialize_subsystem
16	linear_to_ptr
16	local_host_ID
16	mask_slot
64	mp_working_storage_size
16	new_masks
40	null_descriptor
32	ptr_to_linear
68	receive_data
116	receive_unit
32	reset_alarm
16	reset_handler
16	resume_task
16	send_data
16	send_EOI
416	send_dl
16	send_priority_data
444	send_tp
76	send_unit
16	set_descriptor_attributes
16	set_handler
16	set_interconnect
16	set_interrupt
16	set_priority
16	set_time
64	sleep
16	start_PIT
56	start_scheduling
16	stop_scheduling
16	suspend_task
84	tick
16	token_to_ptr
16	translate_ptr
16	unmask_slot

Protected and Unprotected Stacks

If programs use the compact model of segmentation, the BLD386 utility or the Kernel's address management system calls can be used to create protected stacks for the application.

However, programs that use the small model of segmentation cannot use protected stacks. In small model, the data and stack are combined into the data segment; the compiler sets DS equal to SS. When calling **create_task**, most small model programs should pass only DS-relative stack pointers.

Because small model stacks are unprotected and reside in the same segment as data, take great care to accurately determine the stack usage of small model applications. Because the stacks are unprotected, stack overflow can cause other important data to be overwritten.

Re-entrant procedures can cause stack overflow problems, and C functions are by default re-entrant. If you create a task from a re-entrant function, be particularly aware of the potential for stack overflow. All data declared in the function are placed on the task's stack when you create the task. The stack pointer is immediately moved by the amount of data declared. The stack pointer could be set beyond the memory allocated to stack in the task's TSS, causing potential problems when you access that portion of memory. The Kernel does not protect you from such an error. In small model, the first warning may be corrupted code or data. In compact model, you should receive a protection fault.

Accessing Segments Created by the Builder

The BLD386 utility is the system Builder used to create the final code containing your application and the Kernel. (The build process is described later in the chapter.) When using the Builder to create segments, those segments are created outside the subsystem in which the application and the Kernel reside. The application must take special steps to access variables that refer to those segments. Compact programs can access the segments by performing the following steps, shown for PL/M and C.

In PL/M-386

1. Include the $LARGE subsystem control with the EXPORTS parameter to specify the symbols to be defined by the Builder. For example, in the following line, `exported_label` is the symbol for the segment defined by the Builder:

   ```
   $large(builder -const in code- EXPORTS exported_label)
   ```

2. Before using the symbol, declare it to be an external UINT_32. For example:

   ```
   DECLARE exported_label  UINT_32 EXTERNAL;
   ```

3. Create a pointer to a location in the segment as shown below, where *offset* is a number specifying the offset from the start of the segment:

   ```
   BUILD$PTR(SELECTOR$OF(@exported_label), offset);
   ```

In iC-386

1. Declare the symbols to be defined by the Builder as external. For example, in the following line `exported_label` is the symbol for the segment defined by the Builder:

   ```
   extern far exported_label;
   ```

2. Create a pointer to a location in the segment as shown below, where *offset* is a number specifying the offset from the start of the segment:

   ```
   my_ptr = buildptr((selector) &exported_label,
                     (void near *)offset);
   ```

   The `buildptr` function is defined in the iC-386 header file *i86.h*, which is included by the *i386.h* file.

Writing Applications in C

Because the C language is case-sensitive, Kernel functions must be entered exactly as shown in the *iRMK™ Kernel Reference Manual.*

Include file *rmk.h* in all modules that call the Kernel or use literals defined by the Kernel. Include file *kstdio.h* in all small ROM or compact modules that call Kernel standard I/O functions. (See Tables 4-2 and 4-3, on page 4-14.)

Modules compiled in small model must be bound to the *c_call.lib* library. Modules compiled in small ROM or compact model must be bound to the *plm_call.lib* library. Since these libraries use different calling conventions, you take different actions, depending on the library and the C compiler you use:

- The iC-386 default is to use the PL/M calling convention. In modules bound to *c_call.lib*, override the default with the **-vp** control (variable parameter list, or VPL). In modules bound to *plm_call.lib*, make sure the PL/M calling convention is set by using the **-fp** control (fixed parameter list, or FPL).
- The MetaWare High C default is to use the C calling convention. In modules bound to *c_call.lib*, use the default. In modules bound to *plm_call.lib*, (compact or larger), declare a pragma that changes the calling convention. How you use the pragma determines whether the calling convention is changed for all calls or only for calls to the Kernel. To establish the PL/M convention for all calls, use only the first and third lines shown below, at the beginning of each module or in a profile file. To selectively establish the PL/M convention, use the following method in the source code:

```
#define PLM ( !_REVERSE_PARMS | _CALLEE_POPS_STACK)
#define C _DEFAULT_CALLING_CONVENTION
#pragma Calling_convention(PLM);
    /* make a call to the Kernel */
KN_initialize_console(configuration_ptr);
            •
            •
            •
#pragma Calling_convention(C);
    /* make a call using the C convention */
my_procedure(...)
            •
            •
            •
```

Packing Structures in C

All structures used with the Kernel are expected to be *packed*. This means the compiler should not insert zero-padding to align structure elements (for instance, on a four-byte boundary). Many C compilers perform such alignment by default, or allow you to set an alignment compiler switch. If your compiler aligns structures, you must ensure that Kernel structures in your code are not aligned.

For the iC-386 compiler, the Kernel header files already contain a pragma that ensures structures are packed. When using iC-386, you don't need to take any special measures. Only Kernel structures are packed by the header files.

In the MetaWare High C compiler, the default condition is not to align structures; to specifically turn off alignment, use the following statement (in the source file or a profile file). Refer to MetaWare documentation for more details.

```
#pragma Off(Align_members)
```

Using the cstart Module

All C programs must use a startup program that initializes segment registers. The Kernel provides a startup module called *cstart* that:

- initializes segment registers
- invokes the **initialize** system call
- calls the function `main()` (assumed to be the start of your application)

This module is provided as object code assembled with ASM386 for small and compact models (*cstarts.obj* and *cstartc.obj*, respectively), in directory */usr/intel/rmk/system/lib*. The examples for iC-386 use *cstart*. The source for this module is the */usr/intel/rmk/system/src/cstart.asm* file. You may modify the source to provide other initialization for your application.

If you use the *cstart* module, inspect the source to make sure your application doesn't duplicate code provided in *cstart*. For example, the application should not invoke the **initialize** system call after including *cstart*.

Using 64-bit Long Types

The Kernel defines the UINT_64 type as a `long` integer type for use in some system calls. You may or may not want the `long` type to be a 64-bit quantity. In iC-386, the default `long` is 32 bits. Each application module can define 64-bit long types if they are needed in that module. Use the `-long64` control when invoking the compiler or place the following pragma in the source code:

```
#pragma long64
```

Modules compiled in small ROM or compact model must be bound to the *plm_call.lib* library. Modules compiled in small model must be bound to the *c_call.lib* library. Since *c_call.lib* uses the C calling convention, you must include an extra file in small model code: *rmk_conv.plm*. Including this file translates all Kernel calls in the module to the C calling convention, using the $INTERFACE control. Any calls you make to application procedures still use the PL/M calling convention.

Include file *rmk_plm.inc* in all modules. Include file *kstdioc.inc* in all small ROM or compact modules that call Kernel standard I/O functions. (See Tables 4-2 and 4-3, on page 4-14.)

Structures declared in PL/M are automatically packed (no alignment of members), as required by the Kernel.

Writing Applications in FORTRAN

Modules compiled in compact model must be bound to the *plm_call.lib* library. Modules compiled in small or small ROM model must be bound to the *c_call.lib* library (only in FORTRAN does the small ROM model use the *c_call.lib* library). Since *c_call.lib* uses the C calling convention, you must include an extra file in small or small ROM code: *rmk_conv.ftn*. Including this file translates all Kernel calls in the module to the C calling convention, using the $INTERFACE control. Any calls you make to application procedures still use the PL/M calling convention.

Include file *rmk_ftn.inc* in all modules. (See Tables 4-2 and 4-3, on page 4-14.)

Applications written in FORTRAN-386 can use Kernel interface libraries. However, the following conditions apply:

- FORTRAN does not have the POINTER data type; this affects the Kernel calls that take pointers as input. Applications using FORTRAN must be divided into two parts. The only part that can be written in FORTRAN is code not requiring Kernel calls that use the POINTER data type. The remainder of the application must be written in a language that supports the POINTER data type and is usable with the Kernel interface libraries.
- FORTRAN passes parameters by reference. To pass parameters by value, use the %VAL directive. The FORTRAN examples illustrate this.
- FORTRAN does not have the STRUCTURE data type; this affects the Kernel calls that take structures as input. For FORTRAN applications it is necessary to define the members of the structure and then ensure that they are contiguously placed in memory. Two ways to do this are:
  - Define all members of the structure as independent variables. Gather all the members under a NAMED COMMON and use the first element of the common as the address of the structure to pass as an argument.
  - Define all members of the structure as independent variables. Using EQUIVALENCE, equate the members to an INTEGER*1 variable as the address of the structure, and use each member as a member of the structure.
- A FORTRAN application cannot call a Kernel *kstdio* library directly. It must call an assembly (or other language) procedure that directly accesses the library.

Writing Applications in Assembler

In assembler, there is no concept of small or compact, because the programmer has complete control over the processor's registers. Assembler programs do not use an interface library to reference the Kernel. Instead, bind the assembler application directly to the *kernel.lib* library.

The *KN_* calls to the Kernel are stack-based calls, and not used in assembler. The assembler program uses register-based calls. These calls are prefaced with *KNA_*, for example, **KNA_create_task**. To call the Kernel, the program sets up predefined processor registers with the appropriate parameters, then makes the call. Parameters are also returned in processor registers.

Include Files for Assembler

If an assembler program includes the *rmk_asm.inc* file described earlier in Table 4-2, all the Kernel include file declarations become part of the code, whether needed or not. Rather than including file *rmk_asm.inc*, you may want to include a selective list of files; any module that makes system calls or uses Kernel-defined literals must include applicable files from the following list:

rmk_type.equ	Literal declarations of data types used by Kernel system calls, such as UINT_16 and UINT_32
rmk_ex.equ	Literal declarations of all exception codes returned by the Kernel
rmk_base.edf	Declarations of external variables and procedures for the base portion of the Kernel plus the Memory Management and Exchange Management modules (mailboxes, semaphores, and regions)
rmk_dev.edf	Declarations for Device Manager modules (8259A PIC, 8254 PIT, the 82380/82370 Integrated System Peripheral, the numeric coprocessor, and the 82530 USART device)
rmk_dt.edf	Declarations for the Descriptor Table Management module
rmk_is.edf	Declarations for the Interconnect Space Support module
rmk_mp.edf	Declarations for the Message Passing Support module
rmk_base.equ	Literals and constants used with the base portion of the Kernel plus the Memory Management and Exchange Management modules
rmk_dev.equ	Literals for the Device Manager modules
rmk_dt.equ	Literals for the Descriptor Table Management module
rmk_is.equ	Literals for the Interconnect Space Support module
rmk_mp.equ	Literals for the Message Passing Support Module

From the preceding list, include only those files appropriate to references made in the module. Include the files in the order shown. Files *rmk_type.equ* and *rmk_ex.equ* should be included in every module that makes Kernel system calls.

Making Calls to the Kernel

Table 4-6, on the following pages, lists the assembly language interface to each of the Kernel system calls. Each row in the table represents a particular system call. Each column represents a processor register. To find out which registers must hold which values for a particular system call, examine all the entries in the row designated for that system call. The numbers in the table represent the order of the parameters as listed in each system call description in the *iRMK™ Kernel Reference Manual*. For example, in the **set_priority** system call, the first parameter (task) must be placed in the EAX register and the second parameter (priority) in the EBX register.

Some of the entries in Table 4-6 also contain letters. These letters indicate which part of a multiple-part parameter should be placed in that register. The following letters are used:

n(S)	This is the selector part of the pointer that makes up parameter number *n*.
n(O)	This is the offset part of the pointer that makes up parameter number *n*.
n(H)	This is the high 32 bits of the UINT_64 that makes up parameter *n*.
n(L)	This is the low 32 bits of the UINT_64 that makes up parameter *n*.

As Table 4-6 shows, some pointers must be passed in the ES:EDI register pair. If the parameter is actually based on the DS register, ES probably does not need to be set before calling the system call, because ES is normally set equal to DS. However, if ES is not set to a different value, be aware that the Kernel might not set ES back to the DS value upon return from the system call. When writing a high-level language interface that requires ES to be set equal to DS, the interface procedures must ensure that ES is restored after these system calls return.

After setting up the registers with the correct values, invoke a Kernel system call by executing a CALL instruction.

Table 4-6. Assembly Language Kernel Interface

System Calls	Registers EAX	EBX	ECX	EDX	ES	EDI	ESI	FS	GS
KNA_attach_receive_mailbox	1	2							
KNA_attach_protocol_handler	1	2(S)	2(O)	3					
KNA_cancel_dl					1(S)	1(O)			
KNA_cancel_tp					1(S)	1(O)			
KNA_ci									
KNA_co	1								
KNA_create_alarm	3	4	2(S)	2(O)	1(S)	1(O)			
KNA_create_area	1	2							
KNA_create_mailbox	3	4		2	1(S)	1(O)			
KNA_create_pool	2				1(S)	1(O)			
KNA_create_semaphore	2				1(S)	1(O)			
KNA_create_task	1(O)	2(O)	3(O)	4(S)	1(S)	5	6	2(S)	3(S)
KNA_csts									
KNA_current_task_token									
KNA_delete_alarm	1								
KNA_delete_area	2			1(S)		1(O)			
KNA_delete_mailbox	1								
KNA_delete_pool	1								
KNA_delete_semaphore	1								
KNA_delete_task	1								
KN_get_code_selector	No ASM interface. Available only in *c_call.lib* library.								
KN_get_data_selector	No ASM interface. Available only in *c_call.lib* library.								
KNA_get_descriptor_attributes	1	2			3(S)	3(O)			
KNA_get_interconnect	1	2							
KNA_get_PIT_interval									
KNA_get_pool_attributes	1					2(O)		2(S)	
KNA_get_priority	1								
KNA_get_slot									
KNA_get_time									
KNA_initialize	1(S)	1(O)	3(S)	3(O)	2(S)	2(O)			
KNA_initialize_console	1(S)	1(O)							
KNA_initialize_interconnect	1(S)	1(O)							
KNA_initialize_LDT	1	2(S)	2(O)	3					
KNA_initialize_message_passing	1(S)	1(O)	2(S)	2(O)					
KNA_initialize_NDP	1(S)	1(O)							
KNA_initialize_PICs	1(S)	1(O)							
KNA_initialize_PIT	1(S)	1(O)							

Table 4-6. Assembly Language Kernel Interface (continued)

System Calls	Registers								
	EAX	EBX	ECX	EDX	ES	EDI	ESI	FS	GS
KNA_initialize_RDS	1(S)	1(O)							
initialize_stdio									
KNA_initialize_subsystem			1(O)					1(S)	
KNA_linear_to_ptr	1								
KNA_local_host_ID									
KNA_mask_slot	1								
KNA_mp_working_storage	1(S)	1(O)							
KNA_new_masks	1								
KNA_null_descriptor	1	2							
KNA_ptr_to_linear (small)	1	2(S)	3(O)						
KNA_ptr_to_linear	1	2(S)	2(O)						
KNA_receive_data	1	4			2(S)	2(O)	3(O)		3(S)
KNA_receive_unit	1	2							
KNA_reset_alarm	1								
KNA_reset_handler	1(O)				1(S)				
KNA_resume_task	1								
KNA_send_data	1	3					2(O)		2(S)
KNA_send_dl					1(S)	1(O)			
KNA_send_EOI	1								
KNA_send_priority_data	1	3					2(O)		2(S)
KNA_send_tp					1(S)	1(O)			
KNA_send_unit	1								
KNA_set_descriptor_attributes	1	2		3(O)	3(S)				
KNA_set_handler	1(O)				1(S)				
KNA_set_interconnect	1	2	3						
KNA_set_interrupt	1	2(S)	2(O)						
KNA_set_priority	1	2							
KNA_set_time	1(H)	1(L)							
KNA_sleep	1								
KNA_start_scheduling									
KNA_start_PIT	1								
KNA_stop_scheduling									
KNA_suspend_task	1								
KNA_tick									
KNA_token_to_ptr	1								
KNA_translate_ptr	1(O)	2		1(S)					
KNA_unmask_slot	1								

Values Returned from the Kernel

Upon completion of a call to the Kernel, the system call performs a parameterless near return. Those system calls that return values (typed procedures) use the PL/M-386 register conventions for returning values. These conventions are shown in Table 4-7.

Table 4-7. Processor Registers for Returned Values in Assembler

Returned Values	Register
8-bit value	AL
16-bit value	AX
32-bit value	EAX
64-bit value	EDX:EAX
Short Pointer (32 bits)	EAX
Long Pointer (48 bits)	EDX:EAX
Selector	AX

Using Intel Utilities

The BND386 Binder links your object code with the Kernel libraries, and the BLD386 Builder locates the output as a bootloadable application. The input to these two utilities must be OMF386 code, which means you must use Intel tools to compile or assemble the code.

To produce an application that can be debugged with Soft-Scope III, you must bind the code with the Remote Development Server (RDS) and build with the iM III Monitor. (RDS and iM III provide the target interface to Soft-Scope III, as explained in Chapter 2.) When you produce the final, debugged version of the application, you may bind and build it without this debug support, to reduce the size of the application. The example listings in this section show how to bind and build both with and without debug support. The default makefiles for Kernel examples do not include RDS and iM III. Each example includes an alternate makefile, called *makefile.ss*, which you use to include the support while debugging.

The listings of bind and build files in this section assume the same directory structure as the Kernel example code, using four subdirectories called *src*, *obj*, *lst*, and *abs*. For example, a line might specify the relative pathname *../obj/appl.obj*. The subdirectories are as follows:

- *src* holds the source files
- *obj* holds the object files
- *lst* holds the listing files
- *abs* holds the absolute (bootloadable) files

Figure 4-8 gives an overview of the process involved in binding and building the application. The files used in this process are described throughout the remainder of the chapter. The interface.lib file can be either *c_call.lib* or *plm_call.lib*. The kstdiox.lib file can be either *kstdios.lib* or *kstdioc.lib*.

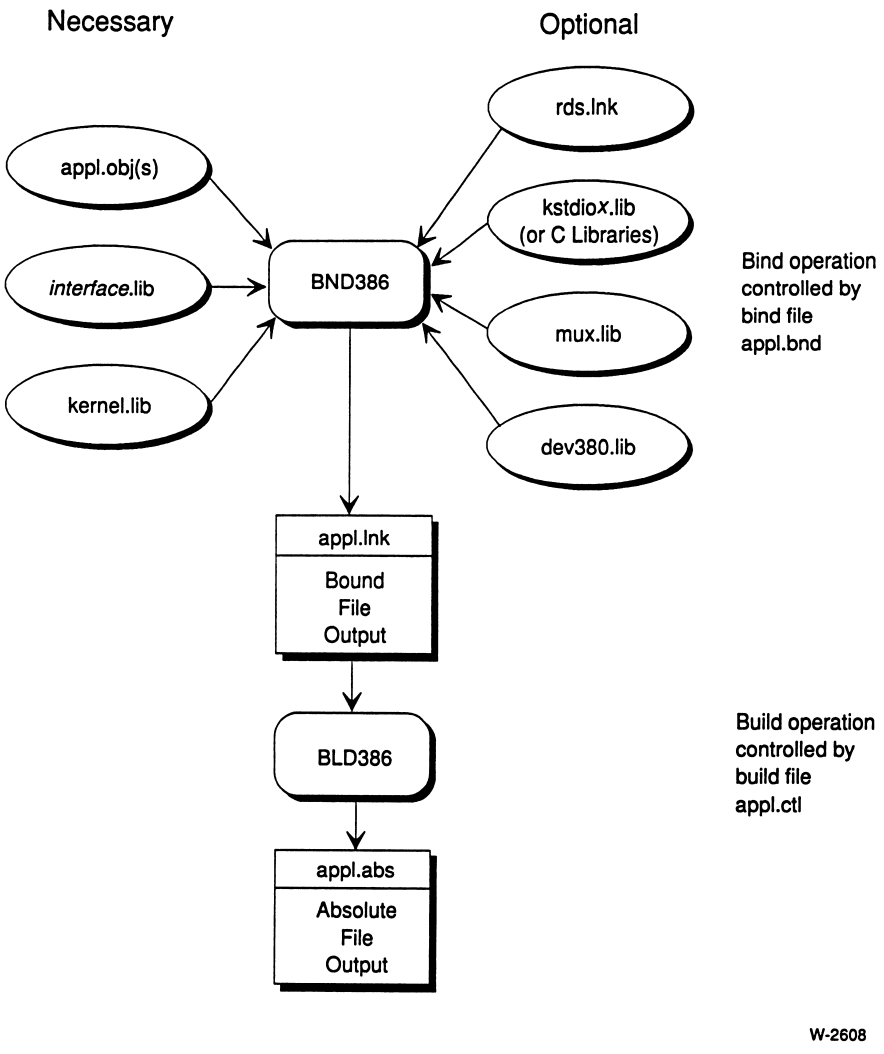


Figure 4-8. Files Used to Bind and Build an Application

Binding the Code

The output of the compilation or assembly is one or more object modules. These modules may be bound (linked) directly to the Kernel or separately, as described in the following sections. Use the BND386 utility, which is described in the *Intel386 Family Utilities User's Guide*.

Invoke the Binder with the following command, where *control.bnd* is a user-written control file containing the BND386 controls and input file specifications.

```
bnd386  CF '(control.bnd)'
```

Several examples of bind control files are listed in the following pages.

Binding to the Kernel without Multiple Segments or Gates

If the application is small model, or compact model with a single ring 0 code segment, bind the object modules together with the Kernel code and an interface library. This produces a linkable object module that can be used as input to the Builder.

Figure 4-9 shows the data to be placed in the *control.bnd* file to bind the application with the Kernel, including debug support (the *rds.lnk* file). Figure 4-10 shows the control file without debug support.

```
; This is a comment

/usr/intel/rmk/system/lib/kernel.lib (DS_START),      &
../obj/appl.obj,                                      &
/usr/intel/rmk/debug/rds/lib/rds.lnk,                 &
/usr/intel/rmk/system/lib/kstdio.lib,                 &
/usr/intel/rmk/system/lib/interface.lib,              &
/usr/intel/rmk/system/lib/kernel.lib                  &
NAME(modname)                                         &
OBJECT(../obj/appl.lnk)                               &
PRINT(../lst/appl.mpl)                                &
NOLOAD
```

Figure 4-9. BND386 Control File with Debug Support (Gateless)

```
/usr/intel/rmk/system/lib/kernel.lib (DS_START),      &
../obj/appl.obj,                                      &
/usr/intel/rmk/system/lib/kstdio.lib,                 &
/usr/intel/rmk/system/lib/interface.lib,              &
/usr/intel/rmk/system/lib/kernel.lib                  &
NAME(modname)                                         &
OBJECT(../obj/appl.lnk)                               &
PRINT(../lst/appl.mpl)                                &
NOLOAD
```

Figure 4-10. BND386 Control File without Debug Support (Gateless)

Make the following substitutions in Figures 4-9 and 4-10, but use the filename extensions shown:

appl.obj — Name of the file containing the compiled or assembled application code. If the application consists of more than one object file, specify each file on a separate line at this point in the control file.

kstdio.lib — Name of the appropriate Kernel *kstdio* library: *kstdios.lib* or *kstdioc.lib*. Only bind with one of these libraries if your application uses the Kernel standard I/O functions. The *s* and *c* in the names stand for small and compact; *kstdios.lib* expects near calls and pointers, while *kstdioc.lib* expects far calls and pointers. Table 4-8 shows which library to use, depending on the programming language and memory model.

interface.lib — Name of the appropriate Kernel interface library: *c_call.lib* or *plm_call.lib*. Table 4-8 shows which interface library to use, depending on the programming language and memory model:

modname — Name that BND386 assigns to the resulting linkable object module. This is not a file name, but a name used within the object module to refer to the entire module. Use the name specified here when you build the final application system.

appl.lnk — Name of the file in which BND386 places the linkable output module.

appl.mp1 — Name of the print file that BND386 generates for listing output and errors during the bind. (This filename extension ends with a "one", not an "L".)

Table 4-8. Interface Library Usage

Language	Model	Interface Library	kstdio Library
iC-386	small small ROM, compact	c_call.lib plm_call.lib	kstdios.lib kstdioc.lib
PL/M-386	small small ROM, compact	c_call.lib plm_call.lib	kstdios.lib kstdioc.lib
FORTRAN-386	small, small ROM compact	c_call.lib plm_call.lib	cannot call from FORTRAN, use assembler
ASM-386		Don't use an interface library	Use the library appropriate to the application

Binding when Using the Gate-based Interface

In earlier releases of the Kernel, which did not have a gate-based interface, Kernel applications had to be bound with the Kernel itself. The application and the Kernel shared the same code segment at privilege ring 0. Tasks not at ring 0 had to be created by a ring 0 application. The user had to set up the call gates and write the interface code.

Although the new gate-based interface does not provide new functionality, it makes using multiple privilege levels much easier. In an application without gates, high-level application code makes a near call to an interface routine. The interface routine translates the KN_* system call into a KNA_* system call. (An assembly application bypasses the interface code and uses the KNA_* system call directly.)

With a gate-based application, the interface code now makes a near call to a multiplexer routine in the *mux.lib* library. (An assembly-language call goes directly to the multiplexer routine.) The multiplexer routine makes a far call to a Demultiplexor routine in the Kernel's code segment. The Demultiplexor routine then calls the appropriate Kernel system call, identified as a KNG_* call.

A file called *kn_gates.obj* must be bound with all modules that are not in the same subsystem as the Kernel. The Kernel provides a default version of *kn_gates.obj*. This file identifies Kernel KNG_* calls with particular gate numbers. Without adding *kn_gates.obj* to the bind, a large number of unresolved references would occur.

You may define a different set of gates using files *gates.p38* (a PL/M-386 source file) and *gates.bld* (a build script used with the BLD386 utility to build a new version of file *kn_gates.obj*). The source file *gates.p38* is a dummy module whose only purpose is to provide a public symbol for the build. The only reason *gates.p38* has any executable statement at all is to avoid an "empty procedure" warning. If you do not use PL/M, write a dummy program for the appropriate language, for example, *gates.c*. In *gates.c*, simply define a procedure called *dummy_entry* (the build script assumes this name).

To establish a new set of gates, modify the *gates.bld* script according to the gates you want to define. Compile *gates.p38* (or *gates.c*) to obtain an object file, and build the object file using the modified *gates.bld* as the control file for BLD386. The result is a new version of *kn_gates.obj*.

NOTE

The final build also assigns gate numbers to the KNG_* calls, and the user must take care to assign the same numbers in the final build as were assigned when building *kn_gates.obj*. The Kernel sets up a number of gates. High-performance system calls may have their own gates, while other system calls may share gates.

Figure 4-11 illustrates the gate-based interface and the bind and build operations that produce the bootloadable application. To use the gate-based interface you must perform two separate binds, one for the Kernel's subsystem and one for the application's subsystem. The two resulting bind files are then input to the Builder. The result is a bootable absolute file. Perform the two binds as follows:

- First, bind the part of the application that is in the same code segment as the Kernel with *kernel.lib*. Be sure to call out the Demultiplexor module as shown below (see Figure 4-12):

  ```
  kernel.lib(DMUX)
  ```

- Then bind the part of the application that is in a different code segment from the Kernel. Bind the application with the Kernel's language interface library (for example, *plm_call.lib*), *mux.lib*, and *kn_gates.obj*. Specify that the KNA_* and KNG_* system calls are not public by using the bind control shown below (see Figure 4-13):

  ```
  PUBLICS EXCEPT (KNA*,KNG*)
  ```

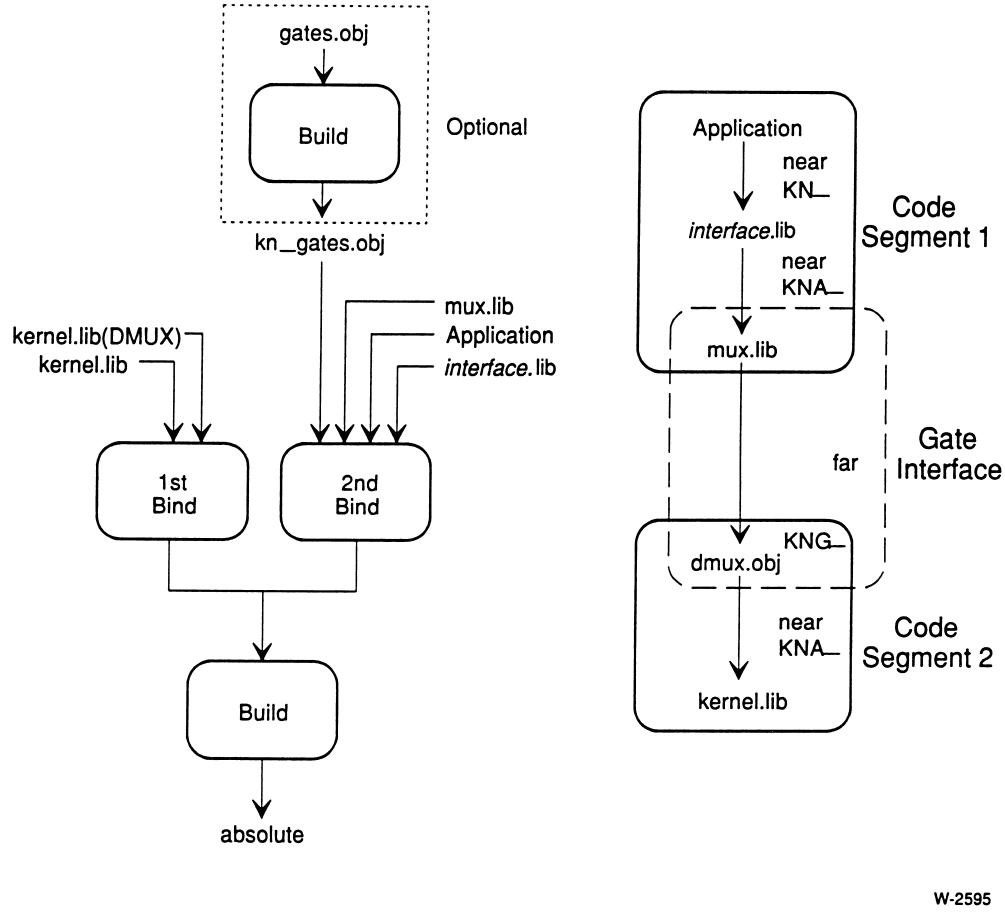


Figure 4-11. Using the Kernel's Gate-based Interface

Figure 4-12 shows the bind sequence for the part of the application that is in the same subsystem as the Kernel. Figure 4-13 shows the bind sequence for the module(s) that call the Kernel through gates. Substitute the appropriate names for *appl_1*, *appl_2*, *modname_1* and *modname_2*, as described in the previous section.

The listings in these figures include debug support. To bind the application without debug support, remove the line printed in bold in Figure 4-12. The *rds.lnk* file should be bound with the part of the application that performs initialization (*appl_1* in the examples below).

```
; This is the bnd386 control file for the
; Kernel-bound part of the segmentation/gate application.
;
/usr/intel/rmk/system/lib/kernel.lib (DS_START),      &
../obj/appl_1.obj,                                    &
/usr/intel/rmk/debug/rds/lib/rds.lnk,                 &
/usr/intel/rmk/system/lib/kstdioc.lib,                &
/usr/intel/rmk/system/lib/plm_call.lib,               &
/usr/intel/rmk/system/lib/kernel.lib (DMUX),          &
/usr/intel/rmk/system/lib/kernel.lib                  &
NAME(modname_1)                                       &
OBJECT(../obj/appl_1.lnk)                             &
PRINT(../lst/appl_1.mpl)                              &
NOLOAD                                                &
PUBLICS EXCEPT (KN_*)
```

Figure 4-12. BND386 Control File for a Gate-based Kernel Subsystem

```
; This is the bnd386 control file for the sub tasks (which
; go through a gate) for the segmentation/gate/subsystem
; application.

../obj/kn_gates.obj,                          &
../obj/appl_2.obj,                            &
/usr/intel/rmk/system/lib/plm_call.lib,       &
/usr/intel/rmk/system/lib/mux.lib             &
NAME(modname_2)                               &
OBJECT(../obj/appl_2.lnk)                     &
PRINT(../lst/appl_2.mpl)                      &
NOLOAD                                        &
PUBLICS EXCEPT(KNA*, KNG*)
```

Figure 4-13. BND386 Control File for a Gate-based Application Subsystem

Binding with the 82380/82370 Functions

If you use the default PIC, PIT, and ADMA managers (described in Chapter 7), you don't need to specify a particular device library when binding the application. The *kernel.lib* file contains the managers for the 8259A, 8254, and ADMA devices. If, instead of these devices, you use an 82370 or 82380 device for the PIC, PIT, and DMA functions, you must bind with the 82380/82370 library. When using the 82380/82370 functions, link the 82380/82370 library ahead of the Kernel library in the bind sequence as shown in Figure 4-14. The added line is shown in bold.

```
/usr/intel/rmk/system/lib/kernel.lib (DS_START),      &
../obj/appl.obj,                                      &
/usr/intel/rmk/system/lib/plm_call.lib,               &
/usr/intel/rmk/system/lib/dev380.lib(*),              &
/usr/intel/rmk/system/lib/kernel.lib                  &
NAME(appl)                                            &
OBJECT(appl.lnk)                                      &
PRINT(appl.mpl)                                       &
NOLOAD
```

Figure 4-14. BND386 Control File for the 82380/82370 Device

Building the Application

After binding the application into a linkable module, use the BLD386 utility to set up the descriptor tables and assign absolute addresses to the resulting module. The Builder is an extremely powerful utility and, depending on the application, the user might want to set up an extensive set of descriptors for segments, tasks, and interrupts. This section describes only the basic information that must be specified to make the Kernel function with a small model application. For an example of building a multiple-segment application, see the build files in one of the following example directories:

/usr/intel/rmk/examples/ic386/seg_gate.cpt/src
/usr/intel/rmk/examples/plm/seg_gate.cpt/src

See also: *Intel386 Family System Builder User's Guide*

Invoke the Builder with the following command, where *control.ctl* is a user-written control file containing the BLD386 controls and input file specifications.

```
bld386  CF '(control.ctl)'
```

The Build Control File

Figure 4-15 shows the data to be placed in the control file. Substitute the appropriate application name for *appl*.

```
; This is a comment

../obj/appl.lnk              &
OBJECT(../abs/appl.abs)      &
BUILDFILE(appl.bld)          &
PRINT(../lst/appl.mp2)       &
BOOTLOAD
```

Figure 4-15. Contents of BLD386 Control File

The controls listed in Figure 4-15 are as follows:

appl.lnk	This is the name of the linked object file generated by BND386.
OBJECT	This control specifies where the absolute output module generated by BLD386 will be placed.
BUILDFILE	This control specifies the file that contains build instructions for BLD386. An example of such a file is shown in Figure 4-16, later in this section.
PRINT	This control specifies the name of the listing file that will contain the build file map of segments, gates, page tables, and tasks, along with warning and error messages.
BOOTLOAD	This control causes BLD386 to generate a bootloadable object module, assigning absolute addresses to all descriptor tables and segments.

If you are building code to be placed in ROM, add the following line at the end of the control file:

```
BOOTSTRAP(startup)
```

The BOOTSTRAP control places at location 0FFFFF0H a short jump instruction to the location designated by the symbol *startup*, in your code. This symbol must be a public symbol representing the location where execution should begin.

See also: Appendix B

Build File Definitions

Figure 4-16 (which continues on the following pages) shows a build file that defines how BLD386 produces the final application. This file must have the name specified with the BUILDFILE control (as in Figure 4-15). Lines shown in bold in Figure 4-16 are for the iM III Monitor, used for Soft-Scope III debug support. These lines may be removed when producing the final debugged application. When removing lines, notice that the last line of each section of the file must end with a semicolon, rather than a comma. The example shown is similar to file */usr/intel/rmk/examples/ic386/bist.cpt/src/testss.bld.*

The description of controls in the file accompanies each section:

CREATESEG

The CREATESEG definition instructs the Builder to create segments outside the subsystem where the application and Kernel reside. In this example, only segments for the iM III Monitor are created.

```
-- This is a comment

appl;

CREATESEG
        MI_ALIAS_SEGMENT (SYMBOL=MI_alias_slot),
        MI_ALIAS_SEGMENT2 (SYMBOL=MI_alias_slot2),
        HW_task_00_stack (SYMBOL=HW_task_00_stack),
        HW_task_01_stack (SYMBOL=HW_task_01_stack),
        HW_task_02_stack (SYMBOL=HW_task_02_stack),
        HW_task_03_stack (SYMBOL=HW_task_03_stack),
        HW_task_04_stack (SYMBOL=HW_task_04_stack),
        HW_task_05_stack (SYMBOL=HW_task_05_stack),
        HW_task_06_stack (SYMBOL=HW_task_06_stack),
        HW_task_07_stack (SYMBOL=HW_task_07_stack),
        HW_task_08_stack (SYMBOL=HW_task_08_stack),
        HW_task_09_stack (SYMBOL=HW_task_09_stack),
        HW_task_0A_stack (SYMBOL=HW_task_0A_stack),
        HW_task_0B_stack (SYMBOL=HW_task_0B_stack),
        HW_task_0C_stack (SYMBOL=HW_task_0C_stack),
        HW_task_0D_stack (SYMBOL=HW_task_0D_stack),
        HW_task_0E_stack (SYMBOL=HW_task_0E_stack),
        HW_task_0F_stack (SYMBOL=HW_task_0F_stack),
        HW_task_10_stack (SYMBOL=HW_task_10_stack);
```

Figure 4-16. Build File for Small Model

SEGMENT

The SEGMENT definition specifies information about the individual fields in the segment descriptors. Notice that the name in the SEGMENT definition (*appl*) is the name that was assigned with the NAME control of BND386.

The example locates the application's data segment at 1D000H. This is the first location used by the application (the MEMORY definition later in the file shows that locations 0 through 1CFFFH are reserved). The application code segment will be placed in higher memory than the data segment. Placing the data segment first is not essential to program execution, but it enables using a debugging technique to track down accesses to nonexistent memory (see Limiting the Size of the Data Segment, on page 4-52). However, this can affect the alignment of descriptor tables: see the note in the MEMORY section.

The bold entries in the SEGMENT section are for the iM III Monitor, and can be removed in a final, debugged application.

```
SEGMENT
        appl
                (DPL = 0),
        appl.stack
                (DPL = 0),
        appl.data
                (BASE = 1D000H),
        MI_alias_segment        (NOT EXPANDDOWN, LIMIT=3, USE16),
        MI_alias_segment2       (NOT EXPANDDOWN, LIMIT=3, USE16),
        HW_task_00_stack        (DPL=0),
        HW_task_01_stack        (DPL=0),
        HW_task_02_stack        (DPL=0),
        HW_task_03_stack        (DPL=0),
        HW_task_04_stack        (DPL=0),
        HW_task_05_stack        (DPL=0),
        HW_task_06_stack        (DPL=0),
        HW_task_07_stack        (DPL=0),
        HW_task_08_stack        (DPL=0),
        HW_task_09_stack        (DPL=0),
        HW_task_0A_stack        (DPL=0),
        HW_task_0B_stack        (DPL=0),
        HW_task_0C_stack        (DPL=0),
        HW_task_0D_stack        (DPL=0),
        HW_task_0E_stack        (DPL=0),
        HW_task_0F_stack        (DPL=0),
        HW_task_10_stack        (DPL=0);
```

Figure 4-16. Build File for Small Model (continued)

TASK

The TASK definition creates a TSS segment for the module and establishes the module as the initial task. Notice that interrupts are disabled.

The bold entries in the TASK section are for the iM III Monitor, and can be removed in a final, debugged application.

```
TASK
        initial_kernel_task
                (DPL = Ø, OBJECT = appl, IOPRIVILEGE = Ø,
                NOT INTENABLED, INITIAL),
        HW_TASK_ØØH     (CODE=HW_VECTOR_ØØH, DPL=Ø, NOT INTENABLED,
                        DATA=HW_task_ØØ_stack,STACKS=(HW_task_ØØ_stack)),
        HW_TASK_Ø1H     (CODE=HW_VECTOR_Ø1H, DPL=Ø, NOT INTENABLED,
                        DATA=HW_task_Ø1_stack,STACKS=(HW_task_Ø1_stack)),
        HW_TASK_Ø2H     (CODE=HW_VECTOR_Ø2H, DPL=Ø, NOT INTENABLED,
                        DATA=HW_task_Ø2_stack,STACKS=(HW_task_Ø2_stack)),
        HW_TASK_Ø3H     (CODE=HW_VECTOR_Ø3H, DPL=Ø, NOT INTENABLED,
                        DATA=HW_task_Ø3_stack,STACKS=(HW_task_Ø3_stack)),
        HW_TASK_Ø4H     (CODE=HW_VECTOR_Ø4H, DPL=Ø, NOT INTENABLED,
                        DATA=HW_task_Ø4_stack,STACKS=(HW_task_Ø4_stack)),
        HW_TASK_Ø5H     (CODE=HW_VECTOR_Ø5H, DPL=Ø, NOT INTENABLED,
                        DATA=HW_task_Ø5_stack,STACKS=(HW_task_Ø5_stack)),
        HW_TASK_Ø6H     (CODE=HW_VECTOR_Ø6H, DPL=Ø, NOT INTENABLED,
                        DATA=HW_task_Ø6_stack,STACKS=(HW_task_Ø6_stack)),
        HW_TASK_Ø7H     (CODE=HW_VECTOR_Ø7H, DPL=Ø, NOT INTENABLED,
                        DATA=HW_task_Ø7_stack,STACKS=(HW_task_Ø7_stack)),
        HW_TASK_Ø8H     (CODE=HW_VECTOR_Ø8H, DPL=Ø, NOT INTENABLED,
                        DATA=HW_task_Ø8_stack,STACKS=(HW_task_Ø8_stack)),
        HW_TASK_Ø9H     (CODE=HW_VECTOR_Ø9H, DPL=Ø, NOT INTENABLED,
                        DATA=HW_task_Ø9_stack,STACKS=(HW_task_Ø9_stack)),
        HW_TASK_ØAH     (CODE=HW_VECTOR_ØAH, DPL=Ø, NOT INTENABLED,
                        DATA=HW_task_ØA_stack,STACKS=(HW_task_ØA_stack)),
        HW_TASK_ØBH     (CODE=HW_VECTOR_ØBH, DPL=Ø, NOT INTENABLED,
                        DATA=HW_task_ØB_stack,STACKS=(HW_task_ØB_stack)),
        HW_TASK_ØCH     (CODE=HW_VECTOR_ØCH, DPL=Ø, NOT INTENABLED,
                        DATA=HW_task_ØC_stack,STACKS=(HW_task_ØC_stack)),
        HW_TASK_ØDH     (CODE=HW_VECTOR_ØDH, DPL=Ø, NOT INTENABLED,
                        DATA=HW_task_ØD_stack,STACKS=(HW_task_ØD_stack)),
        HW_TASK_ØEH     (CODE=HW_VECTOR_ØEH, DPL=Ø, NOT INTENABLED,
                        DATA=HW_task_ØE_stack,STACKS=(HW_task_ØE_stack)),
        HW_TASK_ØFH     (CODE=HW_VECTOR_ØFH, DPL=Ø, NOT INTENABLED,
                        DATA=HW_task_ØF_stack,STACKS=(HW_task_ØF_stack)),
        HW_TASK_1ØH     (CODE=HW_VECTOR_1ØH, DPL=Ø, NOT INTENABLED,
                        DATA=HW_task_1Ø_stack,STACKS=(HW_task_1Ø_stack));
```

Figure 4-16. Build File for Small Model (continued)

GATE

The GATE definition specifies the gates to be established by the Builder. Because this example is for a small model application, the only gates are those for the iM III Monitor. The gates shown in Figure 4-16 can be removed in a final debugged application. In a compact application (see the *seg_gate.cpt* example build files), gates can also established for the application.

```
GATE
        M_WAKE_UP               (DPL=3, ENTRY = MI_save_registers, WC=2),
        M_UTILITY_MGR           (DPL=Ø, ENTRY = MI_utility_mgr, WC=Ø),
        M_EXECUTE_COMMAND       (DPL=Ø, ENTRY = MI_execute_command, WC=12),
        HW_GATE_ØØH             (DPL=3, ENTRY = HW_TASK_ØØH, TASK),
        HW_GATE_Ø1H             (DPL=3, ENTRY = HW_TASK_Ø1H, TASK),
        HW_GATE_Ø2H             (DPL=3, ENTRY = HW_TASK_Ø2H, TASK),
        HW_GATE_Ø3H             (DPL=3, ENTRY = HW_TASK_Ø3H, TASK),
        HW_GATE_Ø4H             (DPL=3, ENTRY = HW_TASK_Ø4H, TASK),
        HW_GATE_Ø5H             (DPL=3, ENTRY = HW_TASK_Ø5H, TASK),
        HW_GATE_Ø6H             (DPL=3, ENTRY = HW_TASK_Ø6H, TASK),
        HW_GATE_Ø7H             (DPL=3, ENTRY = HW_TASK_Ø7H, TASK),
        HW_GATE_Ø8H             (DPL=3, ENTRY = HW_TASK_Ø8H, TASK),
        HW_GATE_Ø9H             (DPL=3, ENTRY = HW_TASK_Ø9H, TASK),
        HW_GATE_ØAH             (DPL=3, ENTRY = HW_TASK_ØAH, TASK),
        HW_GATE_ØBH             (DPL=3, ENTRY = HW_TASK_ØBH, TASK),
        HW_GATE_ØCH             (DPL=3, ENTRY = HW_TASK_ØCH, TASK),
        HW_GATE_ØDH             (DPL=3, ENTRY = HW_TASK_ØDH, TASK),
        HW_GATE_ØEH             (DPL=3, ENTRY = HW_TASK_ØEH, TASK),
        HW_GATE_ØFH             (DPL=3, ENTRY = HW_TASK_ØFH, TASK),
        HW_GATE_1ØH             (DPL=3, ENTRY = HW_TASK_1ØH, TASK);
```

Figure 4-16. Build File for Small Model (continued)

TABLE

The TABLE definition establishes descriptor tables. The example in Figure 4-16 establishes a GDT and an IDT. If the application requires LDTs as well, set them up here.

GDT entry 1 is a descriptor for the GDT, and entry 2 is a descriptor for the IDT. The Builder generates these entries automatically. The GDT definition reserves entries 3 through 63 for the iM III Monitor and MSA bootstrap loader. Be sure to reserve these entries even in a final debugged application. Reserved entries 1024 and 1025 establish the upper end of the GDT entries.

The bold entries in the TABLE section are for the iM III Monitor, and can be removed in a final, debugged application.

```
TABLE
        GDT
                (DPL = 0,
                RESERVE = (3..63, 1024..1025),
                ENTRY = (
                231: MI_ALIAS_SEGMENT2,
                232: MI_ALIAS_SEGMENT,
                233: M_WAKE_UP,
                234: M_EXECUTE_COMMAND,
                235: M_UTILITY_MGR,
                236: HW_GATE_00H,
                237: HW_GATE_01H,
                238: HW_GATE_02H,
                239: HW_GATE_03H,
                240: HW_GATE_04H,
                241: HW_GATE_05H,
                242: HW_GATE_06H,
                243: HW_GATE_07H,
                244: HW_GATE_08H,
                245: HW_GATE_09H,
                246: HW_GATE_0AH,
                247: HW_GATE_0BH,
                248: HW_GATE_0CH,
                249: HW_GATE_0DH,
                250: HW_GATE_0EH,
                251: HW_GATE_0FH,
                252: HW_GATE_10H,
```

Figure 4-16. Build File for Small Model (continued)

The field `300: (appl)` on this page specifies that all the descriptors in application module *appl* should be placed into the GDT, beginning at entry 300 (this could be a different value, such as 298 or 400). Without this field, BLD386 will create an LDT and place the application descriptors there.

```
253: HW_TASK_00H,
254: HW_TASK_01H,
255: HW_TASK_02H,
256: HW_TASK_03H,
257: HW_TASK_04H,
258: HW_TASK_05H,
259: HW_TASK_06H,
260: HW_TASK_07H,
261: HW_TASK_08H,
262: HW_TASK_09H,
263: HW_TASK_0AH,
264: HW_TASK_0BH,
265: HW_TASK_0CH,
266: HW_TASK_0DH,
267: HW_TASK_0EH,
268: HW_TASK_0FH,
269: HW_TASK_10H,
270: HW_task_00_stack,
271: HW_task_01_stack,
272: HW_task_02_stack,
273: HW_task_03_stack,
274: HW_task_04_stack,
275: HW_task_05_stack,
276: HW_task_06_stack,
277: HW_task_07_stack,
278: HW_task_08_stack,
279: HW_task_09_stack,
280: HW_task_0A_stack,
281: HW_task_0B_stack,
282: HW_task_0C_stack,
283: HW_task_0D_stack,
284: HW_task_0E_stack,
285: HW_task_0F_stack,
286: HW_task_10_stack,
300: (appl))
),
```

Figure 4-16. Build File for Small Model (continued)

The TABLE definition continues on the following page with IDT entries.

The IDT definition (part of the TABLE section) reserves entries 126 and 127, which guarantees that the Builder will generate an IDT with slots 0 through 127 available to the application. In the application, the **set_interrupt** system call can be used to place interrupt gates in the IDT. However, make sure that the Builder generates an IDT with enough slots to handle all the interrupt handlers in the system.

The IDT entries shown are for the iM III Monitor, so that system errors will break to the Monitor for debugging. If you remove the iM III Monitor from the final build (removing debug support), you may want to establish application interrupt handlers to handle interrupts 0-16.

```
        IDT
                (DPL = Ø,
                RESERVE = (126..127),
                ENTRY = (
                Ø:HW_GATE_ØØH,
                1:HW_GATE_Ø1H,
                2:HW_GATE_Ø2H,
                3:HW_GATE_Ø3H,
                4:HW_GATE_Ø4H,
                5:HW_GATE_Ø5H,
                6:HW_GATE_Ø6H,
                7:HW_GATE_Ø7H,
                8:HW_GATE_Ø8H,
                9:HW_GATE_Ø9H,
                1Ø:HW_GATE_ØAH,
                11:HW_GATE_ØBH,
                12:HW_GATE_ØCH,
                13:HW_GATE_ØDH,
                14:HW_GATE_ØEH,
                15:HW_GATE_ØFH,
                16:HW_GATE_1ØH
                                )
                );

MEMORY
                (RESERVE = (Ø..1CFFFH, Ø4ØØØØØH..ØFFFFFFFFH));

END
```

Figure 4-16. Build File for Small Model (continued)

MEMORY

The last definition in the build file is the MEMORY definition. This tells BLD386 which absolute addresses to use when assigning addresses to the module. The listing in Figure 4-16 reserves two areas of memory. BLD386 uses the memory between the two reserved areas to assign absolute addresses.

In the example, the memory from 0 to 1CFFFH (120K bytes) is reserved for use by code in firmware (items such as BPS tables, diagnostics data, etc.). This value is known to work with all versions of Intel firmware on Multibus II boards. For newer, MSA firmware, you only need to reserve 64K bytes (0 - 0FFFFH). If you decide to change the amount reserved, use the following guidelines. This applies only to Intel boards with the original firmware as shipped from the factory.

Low Memory to be Reserved for Firmware

Boards that may need 120K	Boards that only need 64K
iSBC 386/100	iSBC 386/120
iSBC 386/116	iSBC 386/133
iSBC 386/258	iSBC 486/125
	iSBC 386/020 (MIX)

The memory from 0400000H and above is reserved for dynamic memory and for addresses for which no physical memory is present. This reserved block can vary depending on the amount of physical memory present and the size of the application. Find out the size of the application by examining the link map created by BND386 (file *appl.mp2*). Add to the size of the application the extra memory assigned for stacks with BLD386.

NOTE

Normally, the Builder locates the descriptor tables in the first available memory, immediately following low memory that is reserved. Specify reserved memory so the first available byte is aligned on a four-byte boundary. The Builder assumes that the GDT and IDT are mode_16 segments, and will place them on two-byte boundaries if allowed to. If the GDT and IDT are not on four-byte boundaries, performance and interrupt latency suffer. If you place the application data segment first (see the BASE = statement in the SEGMENT section), the descriptor tables will be placed immediately after the data, and might not be aligned on a four-byte boundary.

Limiting the Size of the Data Segment

When the Kernel is initialized, it modifies the descriptor for the application's data segment, setting the limit to four gigabytes. By doing this, the Kernel can access any portion of memory without having to reload a segment register. However, having a four gigabyte limit on the data segment can make debugging certain kinds of conditions more difficult.

For example, one kind of error that can be difficult to detect is passing an invalid token to the Kernel. Because there are no error-checking facilities in the Kernel (for performance reasons), the Kernel will attempt to access the memory referenced by the token even if the token doesn't represent a Kernel object. With a four gigabyte data segment limit, the memory reference can be anything (even a reference to nonexistent memory), and the processor will still attempt to access the memory. Such an invalid memory reference can cause unpredictable results, and it can often be hard to pinpoint the cause of the problems.

One way to minimize such problems is to write debugging code or use the debugger to modify the segment limit specified in the data segment's descriptor. Instead of having a four gigabyte limit, set the limit to the actual amount of the memory in the system. This change should occur after the Kernel is initialized. To make the modified limit take effect immediately, reload the DS and ES registers to overwrite the descriptor information stored in the invisible portions of those registers. The debugger or an assembly language procedure can be used to reload DS and ES.

Lowering the limit of the data segment to the size of physical memory in the system still gives the Kernel access to all of memory, but the user can pinpoint invalid tokens that specify nonexistent memory. With a reduced data segment limit, references to nonexistent memory cause the processor to return a general protection (GP) fault, something that is relatively easy to track down with debugging tools.

However, this technique works only if your application's segments are set up in a particular order when using the Builder. This order requires the application's data segment to be the first segment (have the lowest addresses) in the application. The data segment must be first because the Kernel assumes it can access any part of the application using an offset from the DS register. Specify the order with a BASE statement for the application data in the SEGMENT part of the build file, as shown in Figure 4-16.

Using Non-Intel Tools

This section describes tools that have been evaluated to produce code that works with the Kernel. Tools not listed here may also work, but have not been evaluated.

Kernel applications can be developed using Phar Lap's 386|ASM assembler and/or MetaWare's High C compiler. The output of these tools is not OMF386. The Intel Binder and Builder cannot be used with this object code. However, Par Lap's LinkLoc utility accepts the object code, and can produce OMF386 output. In addition, the LinkLoc utility can accept object code produced from Intel compilers and assemblers (OMF386), allowing you to incorporate modules from a variety of sources.

Phar Lap provides a number of tools that can be used to develop applications for the Intel386 family of processors. These tools are available in Phar Lap's 80386 Software Development Series. The Phar Lap tools that can be used with the System V/iRMK Kernel include:

- LinkLoc. This linker/locator tool accepts object modules and object libraries in both Intel's OMF386 and Phar Lap's Easy OMF-386 formats. It can produce executable output files in bootloadable OMF386 format, as well as other formats.
- 386|ASM assembler. Phar Lap's assembler may also be used to produce application code.

MetaWare's High C compiler, available from MetaWare or from Phar Lap, generates code in Phar Lap's Easy OMF-386 format.

The following sections briefly describe using these products. For additional information, refer to the manuals for these products, or contact:

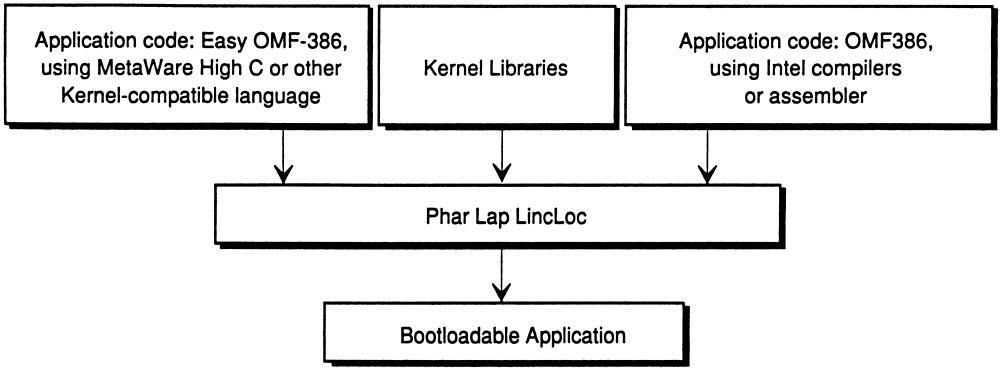
Phar Lap Software Incorporated

or

MetaWare Incorporated

Application Development

Figure 4-17 shows an overview of the development process when using non-Intel tools.



MetaWare and High C are registered trademarks of MetaWare,Inc.
Phar Lap is a trademark of Phar Lap Software, Inc.

W-2602

Figure 4-17. Producing an Application with Non-Intel Tools

The basic steps in producing the application are:

1) Develop your application using compilers and assemblers that generate either Intel OMF386 output, Phar Lap's Easy OMF-386 output, or a combination of the two.

2) Use the LinkLoc utility to produce Intel OMF386 output. Invoke this utility with the command **x1386**.

   a) Link the standard Kernel libraries, including *kstdios.lib* or *kstdioc.lib*, as described earlier for the Intel utilities.

   b) Use the proper switches to set up specific items such as the task's TSS and the requirements for the debug tools being used.

   c) Use the LinkLoc switch *omfboot* to select bootloadable OMF386 output.

Using the High C Compiler and LinkLoc Utility

When using the MetaWare High C compiler, observe the following requirements:

- Do not use the MetaWare default linker; use LinkLoc instead.
- The MetaWare file *init.asm* should be modified. Remove the comment symbol from the line `'PHAR_LAP_CAN_GROW_HEAP equ 1'` and reassemble the file.
- Link the modified *init.asm* file. Link file *hce.lib* to resolve symbols generated by the MetaWare C compiler.

Two examples included with the Kernel are compiled with MetaWare's High C and built with Phar Lap's LinkLoc. The simplest way to build your application with these tools is to copy the appropriate example files and substitute your application source code for the example source code (making the appropriate name changes to the makefile and link file). The example directory is */usr/intel/rmk/examples/highc*. The examples are a small application, *stdio.sml*, and a compact application, *stdio.cpt*. Substitute your object code for files *omfinit.obj* and *intelomf.obj*.

Figure 4-18 shows the files involved in producing an application with the LinkLoc utility. The interface.lib file can be either *c_call.lib* or *plm_call.lib*. The kstdio*x*.lib file can be either *kstdios.lib* or *kstdioc.lib*.

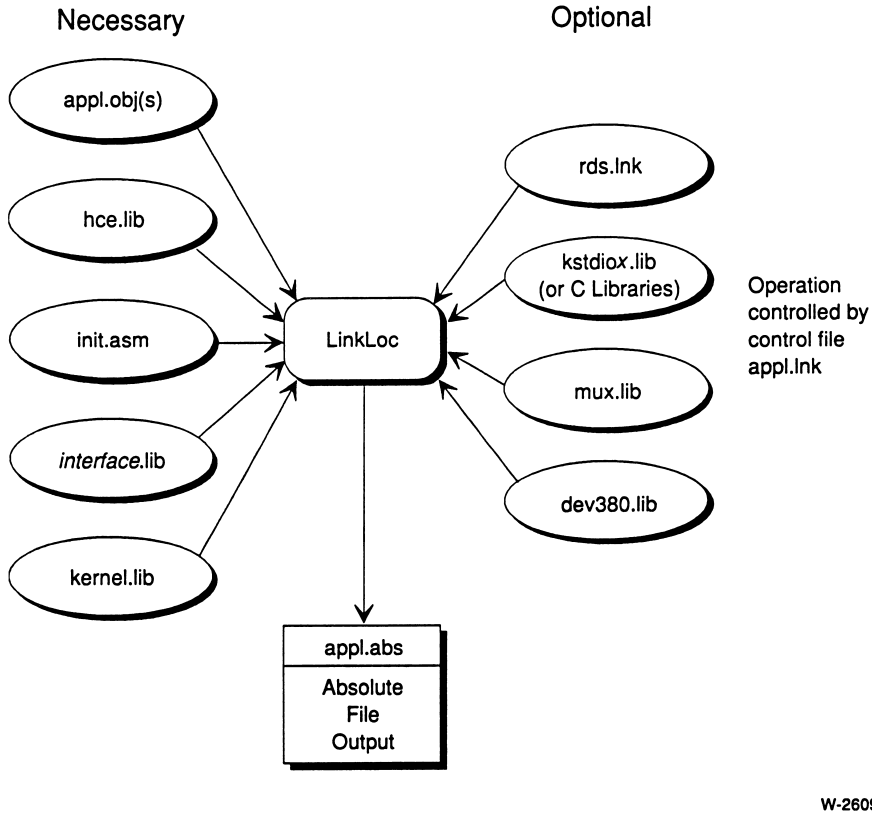


Figure 4-18. Files used to Produce an Application with LinkLoc

Figure 4-19 (continued on the following pages) gives an example of linking a small model High C application with the LinkLoc utility. This file assumes that the MetaWare *hce.lib* file is in the directory */usr/intel/rmk/isv/MetaWare/small*. Change this if you have installed the High C compiler in a different directory. The example shown is similar to the */usr/intel/rmk/examples/highc/stdio.sml/src/testss.lnk* file. To link a compact model application, change the linked libraries from *c_call.lib* and *kstdios.lib* to *plm_call.lib* and *kstdioc.lib*.

The lines shown in bold in Figure 4-19 provide debug support, and may be removed in a final, debugged application. Compare this file to the build file in Figure 4-16. Refer to the discussion accompanying that figure for an explanation of interrupts, gates, segments, and reserved memory.

```
! This link file links the application files (omfinit.obj intelomf.obj ) with
! file init_ex.obj, which is MetaWare's initialization file (init.asm), with
! the statement  ' PHAR_LAP_CAN_GROW_HEAP equ 1' uncommented. The last 3 files
! are for the debugger.

! FILES LINKED:
omfinit.obj intelomf.obj init_ex.obj dbinit.obj mi_alias.obj rds.lnk

! Link the standard kernel libraries, kernel.lib, c_call.lib and kstdios.lib.
! hce.lib is linked to resolve the symbols generated by the MetaWare compiler.

! LIBRARIES LINKED:
-lib /usr/intel/rmk/system/lib/kstdios.lib /usr/intel/rmk/system/lib/c_call.lib
/usr/intel/rmk/system/lib/kernel.lib /usr/intel/rmk/isv/MetaWare/small/hce.lib

-80386                              ! Target CPU
-symbols
-locate 100000h                     ! Specify the start address.
-TASK sys_tss START=main STACK=DGROUP DATASEG=DGROUP ! Initialize the TSS and
                                                    ! the registers.
-start main                         ! Start symbol.
-GROUP CGROUP SEGMENT CODE32        ! Kernel libs are of type CODE32, combine
                                    ! with CGROUP generated by Meta Ware.
-attribute class code ER            ! Set the code, data and stack segment
-attribute class data RW            ! access rights. The defaults are
-attribute class stack RW           ! defined in the linkloc documentation.
-selector segment sys_gdt 8h        ! Set the GDT and IDT aliases to values
-selector segment sys_idt 10h       ! expected by the Kernel.
-selector segment code 300h         ! Set the code segment selector
```

Figure 4-19. Example Link File for Small Model

```
-inttask 0 HW_TASK_00H
-inttask 1 HW_TASK_01H
-inttask 2 HW_TASK_02H
-inttask 3 HW_TASK_03H
-inttask 4 HW_TASK_04H
-inttask 5 HW_TASK_05H
-inttask 6 HW_TASK_06H
-inttask 7 HW_TASK_07H
-inttask 8 HW_TASK_08H
-inttask 9 HW_TASK_09H
-inttask 10 HW_TASK_0AH
-inttask 11 HW_TASK_0BH
-inttask 12 HW_TASK_0CH
-inttask 13 HW_TASK_0DH
-inttask 14 HW_TASK_0EH
-inttask 15 HW_TASK_0FH
-inttask 16 HW_TASK_10H
-limit selector 10h 400h              ! Set the IDT limit
-nullselector  18h-1f8h
-CALLGATE 748h  MI_save_registers NAME=M_WAKE_UP
-CALLGATE 758h  MI_utility_mgr NAME=M_UTILITY_MGR
-CALLGATE 750h  MI_execute_command NAME=M_EXECUTE_COMMAND
-TASKGATE 7B0h  HW_TASK_00H NAME=HW_GATE_00H
-TASKGATE 7B8h  HW_TASK_01H NAME=HW_GATE_01H
-TASKGATE 7C0h  HW_TASK_02H NAME=HW_GATE_02H
-TASKGATE 7C8h  HW_TASK_03H NAME=HW_GATE_03H
-TASKGATE 7D0h  HW_TASK_04H NAME=HW_GATE_04H
-TASKGATE 7D8h  HW_TASK_05H NAME=HW_GATE_05H
-TASKGATE 7E0h  HW_TASK_06H NAME=HW_GATE_06H
-TASKGATE 7E8h  HW_TASK_07H NAME=HW_GATE_07H
-TASKGATE 800h  HW_TASK_08H NAME=HW_GATE_08H
-TASKGATE 808h  HW_TASK_09H NAME=HW_GATE_09H
-TASKGATE 810h  HW_TASK_0AH NAME=HW_GATE_0AH
-TASKGATE 818h  HW_TASK_0BH NAME=HW_GATE_0BH
-TASKGATE 820h  HW_TASK_0CH NAME=HW_GATE_0CH
-TASKGATE 828h  HW_TASK_0DH NAME=HW_GATE_0DH
-TASKGATE 830h  HW_TASK_0EH NAME=HW_GATE_0EH
-TASKGATE 838h  HW_TASK_0FH NAME=HW_GATE_0FH
-TASKGATE 840h  HW_TASK_10H NAME=HW_GATE_10H
-segment HW_task_00_stack LIMIT=1000h RW
-segment HW_task_01_stack LIMIT=1000h RW
-segment HW_task_02_stack LIMIT=1000h RW
-segment HW_task_03_stack LIMIT=1000h RW
-segment HW_task_04_stack LIMIT=1000h RW
-segment HW_task_05_stack LIMIT=1000h RW
-segment HW_task_06_stack LIMIT=1000h RW
-segment HW_task_07_stack LIMIT=1000h RW
```

Figure 4-19. Example Link File for Small Model (continued)

```
-segment HW_task_08_stack LIMIT=1000h RW
-segment HW_task_09_stack LIMIT=1000h RW
-segment HW_task_0A_stack LIMIT=1000h RW
-segment HW_task_0B_stack LIMIT=1000h RW
-segment HW_task_0C_stack LIMIT=1000h RW
-segment HW_task_0D_stack LIMIT=1000h RW
-segment HW_task_0E_stack LIMIT=1000h RW
-segment HW_task_0F_stack LIMIT=1000h RW
-segment HW_task_10_stack LIMIT=1000h RW
-selector segment MI_alias_slot_seg 740h
-selector segment MI_alias_slot2_seg 738h
-selector segment HW_task_00_stack 848h
-selector segment HW_task_01_stack 850h
-selector segment HW_task_02_stack 858h
-selector segment HW_task_03_stack 860h
-selector segment HW_task_04_stack 868h
-selector segment HW_task_05_stack 870h
-selector segment HW_task_06_stack 878h
-selector segment HW_task_07_stack 880h
-selector segment HW_task_08_stack 888h
-selector segment HW_task_09_stack 890h
-selector segment HW_task_0A_stack 898h
-selector segment HW_task_0B_stack 8A0h
-selector segment HW_task_0C_stack 8A8h
-selector segment HW_task_0D_stack 8B0h
-selector segment HW_task_0E_stack 8B8h
-selector segment HW_task_0F_stack 8C0h
-selector segment HW_task_10_stack 8C8h
-TASK HW_TASK_00H START=HW_VECTOR_00H STACK=HW_task_00_stack
                 DATASEG=HW_task_00_stack
-TASK HW_TASK_01H START=HW_VECTOR_01H STACK=HW_task_01_stack
                 DATASEG=HW_task_01_stack
-TASK HW_TASK_02H START=HW_VECTOR_02H STACK=HW_task_02_stack
                 DATASEG=HW_task_02_stack
-TASK HW_TASK_03H START=HW_VECTOR_03H STACK=HW_task_03_stack
                 DATASEG=HW_task_03_stack
-TASK HW_TASK_04H START=HW_VECTOR_04H STACK=HW_task_04_stack
                 DATASEG=HW_task_04_stack
-TASK HW_TASK_05H START=HW_VECTOR_05H STACK=HW_task_05_stack
                 DATASEG=HW_task_05_stack
-TASK HW_TASK_06H START=HW_VECTOR_06H STACK=HW_task_06_stack
                 DATASEG=HW_task_06_stack
-TASK HW_TASK_07H START=HW_VECTOR_07H STACK=HW_task_07_stack
                 DATASEG=HW_task_07_stack
```

Figure 4-19. Example Link File for Small Model (continued)

```
-TASK HW_TASK_08H START=HW_VECTOR_08H STACK=HW_task_08_stack
                 DATASEG=HW_task_08_stack
-TASK HW_TASK_09H START=HW_VECTOR_09H STACK=HW_task_09_stack
                 DATASEG=HW_task_09_stack
-TASK HW_TASK_0AH START=HW_VECTOR_0AH STACK=HW_task_0A_stack
                 DATASEG=HW_task_0A_stack
-TASK HW_TASK_0BH START=HW_VECTOR_0BH STACK=HW_task_0B_stack
                 DATASEG=HW_task_0B_stack
-TASK HW_TASK_0CH START=HW_VECTOR_0CH STACK=HW_task_0C_stack
                 DATASEG=HW_task_0C_stack
-TASK HW_TASK_0DH START=HW_VECTOR_0DH STACK=HW_task_0D_stack
                 DATASEG=HW_task_0D_stack
-TASK HW_TASK_0EH START=HW_VECTOR_0EH STACK=HW_task_0E_stack
                 DATASEG=HW_task_0E_stack
-TASK HW_TASK_0FH START=HW_VECTOR_0FH STACK=HW_task_0F_stack
                 DATASEG=HW_task_0F_stack
-TASK HW_TASK_10H START=HW_VECTOR_10H STACK=HW_task_10_stack
                 DATASEG=HW_task_10_stack
-selector first 200h                  ! First selector 200h
-reserve memory 0h-1cfffh             ! Reserve memory required by firmware
-reserve memory 1000000h-0ffffffffh   ! memory not in the system
-OMFBOOT testss                       ! Produce Bootloadable OMF
```

Figure 4-19. Example Link File for Small Model (continued)

***

BOOTLOADING THE APPLICATION 5

After producing a bootloadable application, you must modify the system bootstrap configuration file and then reboot either the entire system or the host where code is to be loaded. This section describes the bootstrap configuration file, modifications you make to the file, and commands to reset boards in the system.

NOTE

The bootstrap configuration file in System V 3.2 is the */etc/default/bootserver/config* file. In System V 4.0, the bootstrap configuration file is the */stand/config* file.

Before modifying the *config* file, make a backup copy in the same directory. For example, you might keep a copy named *config.orig*. Be careful when editing the *config* file. Mistakes you make here could keep the UNIX system from booting. If this should occur, Appendix C describes how you can reboot UNIX.

An Example Configuration for Bootloading

To simplify the examples shown in this section, assume that the board configuration in your system is as shown in Table 5-1. The processor boards shown in slots 2-4 could be a variety, such as iSBC 386/120 or 486/125 boards, or others. The Kernel does not limit you to the configuration shown in Table 5-1.

Table 5-1. Example System Configuration

Slot	Board	Purpose
0	iSBC 386/258 with CSM 002	PCI File Server
1	slot taken by iSBX 279 on board in slot 0	Graphics controller, System V console
2	processor board	System V host
3	processor board	Real-time host
4	processor board	Real-time host
5	slot taken by iSBX 354 on board in slot 4	Serial I/O for board in slot 4

Understanding the Bootstrap Configuration File

In a Multibus II system running System V/386, the bootstrap configuration file controls what code is loaded on the boards in the system. The MSA (Multibus System Architecture) bootstrap loader reads instructions from the file to determine the boards to boot and the software to load on them.

The instructions in this file are called *Bootstrap Parameter Strings*, or BPS parameters. BPS parameters in the configuration file determine which slots contain boards to be booted and the files to load on the boards, along with other information used by the system. The firmware stores BPS parameters in memory on the appropriate boards in the system. System software interrogates the BPS parameters in memory to determine its actions.

Figure 5-1 shows the basic bootstrap configuration file for a system that does not contain a real-time host. In our example configuration (Table 5-1), the hosts in slots 2, 3, and 4 would all boot System V using this file. The file is divided into headings, which are surrounded with brackets ([ ]). The BPS parameters immediately below the entry

```
[BL_Host_id=GLOBAL];
```

apply to all boards in the system. These parameters (for example, `unix_hostc = 4`) define names for the slot numbers. Notice that these names are used later in the file with the $ replacement symbol. For example, the next-to-last line is

```
[BL_Host_id=$unix_hostc;BL_method=Quasi];
```

This refers to the board in slot 4, and any BPS parameters listed below it only apply to that board.

In the example configuration, there is no processor board in slot 1. The four lines that refer to a board in slot 1 (beginning with `[BL_Host_id=$pci_hostb];`) could be commented out, using the # symbol.

The following section describes examples of changes to make to the file shown in Figure 5-1, so that you can boot a Kernel application. The actual changes you make to the file may be different from the examples shown.

For more information on BPS parameters, see the *Firmware User's Guide for MSA Firmware.*

```
#  Copyright (c) 1988  Intel Corporation
#  All Rights Reserved
#  INTEL CORPORATION PROPRIETARY INFORMATION
#  This software is supplied to AT & T under the terms of a license
#  agreement with Intel Corporation and may not be copied nor
#  disclosed except in accordance with the terms of that agreement.
#  ident  "@(#)mbus:cmd/bootutils/config.d/config.520  1.1"
#  sample configuration file for SYP520
#  This configuration file is set up to work with both flavors
#  of the SYP520.
#  (I) iSBC 386/258 with CSM 002 and iSBX 279 in slot 0,
#  iSBC 386/120 in slot 2, etc. and
#  (II) CSM 001 in slot 0, iSBC 386/258 with iSBX 279 in slot 1,
#  iSBC 386/120 in slot 3, etc.
#
[BL_Host_id=GLOBAL];
        pci_hosta = 0;
        pci_hostb = 1;
        unix_hosta = 2;
        unix_hostb = 3;
        unix_hostc = 4;

# PCI Host

[BL_Host_id=$pci_hosta];
        BL_QI_Master = $unix_hosta;
        BL_Target_file = /etc/default/bootserver/pci258;
        BL_mode = p;

# PCI Host

[BL_Host_id=$pci_hostb];
        BL_QI_Master = $unix_hostb;
        BL_Target_file = /etc/default/bootserver/pci258;
        BL_mode = p;

# Unix Host  #1 or

[BL_Host_id=$unix_hosta;BL_method=Quasi];
        BL_Target_file = /unix;

# Unix Host  #1

[BL_Host_id=$unix_hostb;BL_method=Quasi];
        BL_Target_file = /unix;

# Unix Host  #2

[BL_Host_id=$unix_hostc;BL_method=Quasi];
        BL_Target_file = /unix;
```

Figure 5-1. Basic Bootstrap Configuration File

Editing the Bootstrap Configuration File

The first step in bootloading your application is to edit the bootstrap configuration file. The changes you make include:

- defining names for the real-time hosts
- identifying the second-stage bootstrap loader used to boot Kernel applications on real-time hosts
- specifying the bootloadable (target) files to load on the various boards
- providing information used by the debugger software

Figure 5-2 shows the type of changes to make in the bootstrap configuration file. The lines printed in bold indicate changes from the file shown in Figure 5-1. The changed lines are described below. Use these as a guideline; the changes you make in your file depend on the system configuration, the files you want to load, and your debugger configuration. The **BL_debug_on_boot** and **CC_console** parameters are part of the debugger configuration described in Chapter 2.

rmk_hosta = 3;
rmk_hostb = 4;

Specifies your names for the real-time hosts in slots 3 and 4.

rmk_stage2 = /msa/stage2.rmk;

Specifies a name for the second-stage bootstrap loader, file */msa/stage2.rmk*. This file is installed with the Kernel.

[BL_Host_id=$rmk_hosta;BL_second_stage = $rmk_stage2];

Identifies the host in slot 3 and the second-stage bootstrap loader to use. The following three BPS parameters apply to this host.

BL_Target_file = /usr/donna/rmk_app/abs/realtime.abs;

Specifies a bootloadable Kernel application to load when the host is rebooted.

BL_debug_on_boot = 1M;

Specifies that the debugger will be invoked before running the bootloaded file.

CC_console = ccrc1;

Specifies RCI software for debugger communications on this host.

[BL_Host_id=$rmk_hostb;BL_second_stage = $rmk_stage2];

Identifies the host in slot 4 and the second-stage bootstrap loader to use. The following three BPS parameters apply to this host.

BL_Target_file =
/usr/intel/rmk/examples/ic386/bist.sml/abs/testss.abs;

Specifies one of the Kernel examples to load when the host is rebooted. Notice that the BPS parameter can be on more than one line.

BL_debug_on_boot = on;
Specifies that the loaded code will begin running immediately, without invoking the debugger.

CC_console = cc354a;
Specifies that debugger communications will use an RS-232 serial link to the iSBX 354 module on this host.

```
[BL_Host_id=GLOBAL];
        pci_hosta = 0;
#       pci_hostb = 1;
        unix_hosta = 2;
        rmk_hosta = 3;
        rmk_hostb = 4;

        rmk_stage2 = /msa/stage2.rmk;

# PCI Host

[BL_Host_id=$pci_hosta];
        BL_QI_Master = $unix_hosta;
        BL_Target_file = /etc/default/bootserver/pci258;
        BL_mode = p;

# PCI Host

#[BL_Host_id=$pci_hostb];
#       BL_QI_Master = $unix_hostb;
#       BL_Target_file = /etc/default/bootserver/pci258;
#       BL_mode = p;

# Unix Host  #1

[BL_Host_id=$unix_hosta;BL_method=Quasi];
        BL_Target_file = /unix;

# Real-time Host  A

[BL_Host_id=$rmk_hosta;BL_second_stage = $rmk_stage2];
        BL_Target_file = /usr/donna/rmk_app/abs/realtime.abs;
        BL_debug_on_boot = 1M;
        CC_console = ccrci;

# Real-time Host  B

[BL_Host_id=$rmk_hostb;BL_second_stage = $rmk_stage2];
        BL_Target_file =
                /usr/intel/rmk/examples/ic386/bist.sml/abs/testss.abs;
        BL_debug_on_boot = on;
        CC_console = cc354a;
```

Figure 5-2. Edited Bootstrap Configuration File

Rebooting Individual Boards

After you make the appropriate changes to the bootstrap configuration file, you must reboot the boards to load your application. One way to do this is to reboot the entire system. However, there are commands to selectively reboot individual boards in the system. These commands are **initbp**, **reset**, and **reboot**. You may use any of these commands to reboot one or more boards in the system. The first two commands are part of System V/386, and the last is provided with the Kernel. Two other Kernel commands, **bootstat** and **nmi**, are also useful under certain conditions. Table 5-2 gives a synopsis of these commands, which are described on the following pages.

Before rebooting a real-time host, configure the debugger, as described in Chapter 2.

Table 5-2. Commands for Rebooting Individual Boards

Utility	Path	Action
initbp	/usr/lbin/initbp	Clears all current BPS parameters and resets board Can reset current UNIX host or file server
reset	/usr/lbin/reset	Clears firmware and configuration file BPS parameters (not those entered by bootstrap loader or operator) and resets board Faster than initbp command Can reset current UNIX host or file server
reboot	/usr/intel/rmk/bin/reboot	Clears firmware and configuration file BPS parameters (not those entered by bootstrap loader or operator) and resets board Faster than reset command Clears bootstrap status register from previous boot Cannot reset current UNIX host, but can reset file server
bootstat	/usr/intel/rmk/bin/bootstat	Displays information about boot status, useful if board won't reboot
nmi	/usr/intel/rmk/bin/nmi	Issues non-maskable interrupt to board, useful to stop runaway process Cannot interrupt current UNIX host, but can interrupt file server

CAUTION

Before you reboot the system or any System V host, make sure you have not changed any lines in the bootstrap configuration file that refer to a System V host or to the file server (PCI host). Changes you make to those entries could make it impossible to automatically boot UNIX. Make sure you have a backup copy of the unedited file called *config.orig*. If editing the *config* file causes the system not to boot UNIX, refer to Appendix C.

bootstat

```
bootstat [slot_list]
```

This command is useful if you cannot reboot a board. It displays the contents of the bootstrap status register so that you can get an idea of what may be happening. Specify a slot number or a series of numbers separated with spaces. Bootstrap status information is displayed for each slot specified. If no slots are specified, the boot status is displayed for all boards in the system. Included in the display are:

- the slot number
- whether bootstrap has been enabled or disabled for that slot
- the bootstrap loader stage most recently executed
- the status byte value
- the mnemonic for the status byte value (if known)

If you use the **initbp** or **reset** command to reset a board, the bootstrap status registers are not cleared, and the information displayed by **bootstat** may not be valid for the last boot attempt. If you use the **reboot** command instead, the bootstrap status displayed by **bootstat** is for the current boot attempt.

See also: *Firmware User's Guide for MSA Firmware*

```
initbp [-v] slot_number
```

This command resets the board specified by the slot number on the command line, and causes it to bootstrap load. All BPS parameters from all sources are set to null before the reset. There is no protection from resetting the board where you are working.

-v When this option is specified, diagnostic information is displayed during execution of the command.

nmi

```
nmi [slot_list]
```

This command causes a non-maskable interrupt (NMI) on the board(s) specified in the command line. Specify a slot number or series of numbers separated with spaces. If no slots are specified, the command does nothing. The **nmi** command is useful to stop a runaway process on a board, but it only works if an appropriate NMI handler is present in the IDT of the interrupted board. If you attempt to interrupt the System V host where you are working, the command aborts with an error message. It is not able to detect if you are attempting to interrupt some board that your host is dependent on, such as the PCI file server.

`reboot [slot_list]`

This command resets the board(s) specified on the command line, and causes them to bootstrap load. Current BPS parameters entered by the bootstrap loader or an operator using the MSA Master Test Handler are retained after the reset. During the reset process, BPS parameters in the bootstrap configuration file overwrite those set previously. This command resets the bootstrap status register, clearing error information. If a **bootstat** command is subsequently issued, the bootstrap status displayed is for the current boot attempt.

Specify a slot number or series of numbers separated with spaces. If no slots are specified, the command does nothing. If you attempt to reset the System V host where you are working, the command aborts with an error message. It is not able to detect if you are attempting to reboot some board that your host is dependent on, such as the PCI file server.

reset

```
reset [-b] [-v] [-m | -n | -i index] slot_number
```

This command resets the board specified by the slot number on the command line, and causes it to bootstrap load. Current BPS parameters entered by the bootstrap loader or an operator using the MSA Master Test Handler are retained after the reset. During the reset process, BPS parameters in the bootstrap configuration file overwrite those set previously. There is no protection from resetting the board where you are working.

-b When this option is specified, the BIST complete bit is not set. Default is to set the bit.

-v When this option is specified, diagnostic information is displayed during execution of the command.

-m | -n | -i *index*

These options have to do with the program table index register (PTIR), described in the *Firmware User's Guide for MSA Firmware*. The default is to write 0 to the PTIR.

-m Update the contents of the PTIR to invoke the firmware debug monitor (equivalent to specifying `-i 248`)

-n Leave the contents of the PTIR unchanged (allows the user to sequence through the entries in the program table)

-i *index* Write the value of *index* to the PTIR

***

USING BASIC KERNEL SERVICES 6

Overview

The Kernel architecture is defined by specific object types and the unique set of operations that may be performed on each object. The Kernel manipulates these objects to implement a real-time multitasking environment for applications.

The basic features of the Kernel include:

- Object management
- Task management
- Interrupt management
- Time management

The Kernel object management facilities control creating, deleting, and manipulating object types defined by the Kernel. The user provides memory for Kernel objects and may allocate memory beyond the Kernel's needs. The application can use this additional memory to store application-specific state information associated with the object.

The Kernel task management facilities multiplex the CPU among asynchronous, independent tasks. The Kernel uses a pre-emptive, priority-based scheme to schedule the tasks, while maintaining an image of the CPU registers for each task.

The interrupt management facilities enable applications to react to events external to the CPU in a timely fashion. To do this, the Kernel supports interrupt handlers, which are procedures automatically invoked by the hardware when an interrupt occurs.

The Kernel also provides several time management facilities for applications. The Kernel depends upon the application to provide a source of periodic signals to measure the passing of time. This keeps the Kernel from being dependent on specific timer hardware. The Kernel provides modules that manage standard timer devices.

Objects

Objects are Kernel-maintained data structures classified into specific types, each type having its own set of operations and unique attributes. The Kernel defines the following object types:

- Tasks
- Alarms
- Semaphores
- Mailboxes
- Memory Pools

Tasks and alarms are described in this chapter. Semaphores, mailboxes, and memory pools are described in Chapter 7.

Creating Objects

To create and use an object, the application does the following:

- allocates memory, which should be aligned on a four-byte boundary for maximum performance
- invokes the system call that creates the particular type of object
- uses the token returned by the creating system call to access the object

Objects may be created anywhere in the application's memory space. The application must provide sufficient memory to contain the data structures that define the object. Literals declared in the Kernel's include files specify the amount of memory the Kernel needs for each object type (include files are described in Chapter 4). The application may allocate extra memory for the object, to be used by the application.

Using Object Tokens

When an object is created, the Kernel returns a 32-bit token that identifies the object. Thereafter applications access the object by passing the token to the appropriate system call with requests to interrogate, modify, or perform some operation using the object.

Kernel tokens are recognized over the entire processor. This means that even if an application sets up local environments (LDTs), the tokens for objects created within the local environment are still accessible by all tasks running on that processor.

Access to an Object's State

The data associated with an object is called the object's *state*. The **token_to_ptr** system call accepts the token for an object and returns a pointer to the object's state. This call gives the application direct access to this state data. Typically, the application should only modify application-specific state data in any additional object memory allocated by the application. This chapter describes items within a task's state (TSS), that an application may access directly.

CAUTION

The Kernel has no object protection. The application is in complete control of memory management. Since the application supplies the memory the Kernel uses to maintain objects, the application can directly access the state of Kernel objects. In general, however, modifying object memory areas maintained by the Kernel can cause unpredictable results.

Deleting Objects

The Kernel provides system calls to delete objects that are no longer needed by the application. Deleting an object removes the Kernel's association of state data with an object. The specific object to be deleted is indicated by the object token.

CAUTION

The Kernel does not protect itself against unexpected object deletion. The application must ensure that Kernel system calls do not attempt to access objects either while the objects are being deleted or after they have been deleted.

Task Management

A task is an object that executes a sequence of instructions to manipulate data and other objects. Each task is a thread of execution that performs some function of an application. This thread of execution can be shared among multiple tasks. In the course of its execution, a task may call procedures or functions. Code for tasks may be written for one task, or it can be shared among several tasks. In addition, a task can create, delete, suspend, or resume other tasks.

Through task scheduling, the Kernel controls the distribution of CPU execution cycles between tasks. To schedule tasks, the Kernel uses two attributes: task execution states and task priorities. The next sections discuss these attributes in detail.

The Five Task Execution States

A Kernel task always exists in one of five task execution states: ready, running, asleep, suspended, or asleep-suspended, as follows:

Ready State. A task in the ready state is capable of executing, but is not actually running. A ready task is not running, asleep, suspended, or asleep-suspended.

Running State. A task in the running state is executing on the processor. Only one task can be executing on a processor at a time.

Asleep State. A task may allow the Kernel to put it to sleep. A task in the asleep state is voluntarily waiting for something external to wake it. It does not execute while asleep. A task may put itself to sleep with the **sleep** system call, or may be put to sleep while waiting for a result from various other system calls. A task goes to sleep for one of two reasons:

- To wait for an event to occur, such as a request to be granted
- To wait a specific amount of time, as in the case of a task that periodically puts itself to sleep

Suspended State. A task enters the suspended state as a result of a **suspend_task** system call. Execution is postponed when a task is suspended. Multiple suspend operations on a task increase the suspension depth associated with that task. The suspension depth reflects the number of suspend operations outstanding against a task. The suspension depth can range from 0 (the task is not suspended) to 255.

The **resume_task** system call removes suspensions against a task. The task leaves the suspended state when a resume operation is issued for each suspend operation on the task. For example, a task with a suspension depth of five would require five resume operations to bring the suspension depth back to zero (no longer suspended).

There are three ways a task can be suspended:

- by another task suspending it
- by suspending itself
- by being created in the suspended state

Asleep-suspended State. When a sleeping task is suspended by another task, it enters the asleep-suspended state. While in this state, the task may be awakened, leaving it in the suspended state. It also may be resumed, returning it to the asleep state. However, a task cannot go from the suspended state to the asleep-suspended state. Since a suspended task is not executing, it cannot put itself to sleep.

The rules governing suspension depth also apply to asleep-suspended tasks.

NOTE

The previous discussion on task states assumes that scheduling has not been locked. A scheduling lock allows task switches into the ready state but delays any task switch to the running state. See the section titled, "Controlling Task Switches" on page 6-12 for additional information.

Task State Transitions

Tasks make transitions from state to state during the course of their activities. Figure 6-1 illustrates the possible task state transitions, which are described below.

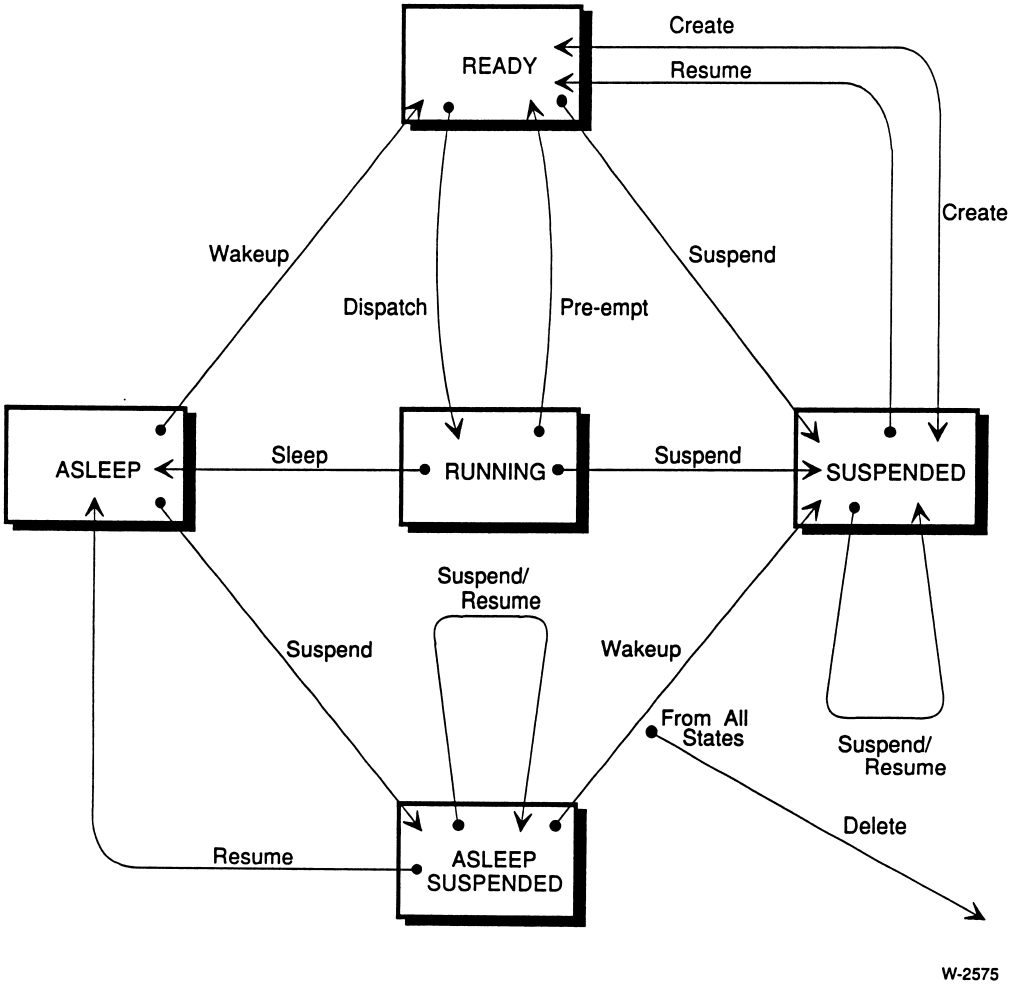


Figure 6-1. Scheduler Task State Transitions

Create. The task is newly created; it is placed into the suspended state or the ready state as specified in the **create_task** system call.

Dispatch. The task goes from the ready state to the running state when one of the following occurs:

- The task has just become ready and has the highest priority of all ready tasks including the currently running task.
- The currently running task gives up the processor because it is suspended, put to sleep, or deleted, and this task has the highest priority of all ready tasks.
- The task has the same priority as the currently running task, and the latter relinquishes the processor because its allotted time slice is up (time-slicing is discussed later in this chapter). No other ready task of the same priority has been ready longer than the task.

Pre-empt. The task goes from the running state to the ready state when one of the following occurs:

- A task with higher priority becomes ready.
- The task has used up its allotted time slice, and a task of equal priority is ready.

Sleep. The task goes from the running state to the asleep state when one of the following occurs:

- The task puts itself to sleep for a specified length of time.
- The task makes a request in a system call that cannot be granted immediately, but the task is willing to wait (such as waiting at a mailbox for a message).

Wakeup. The task goes from asleep to ready or from asleep-suspended to the suspended state when one of the following occurs:

- The time period specified when the task put itself to sleep expires.
- The task's request is granted.

Suspend. The task goes from the running state to the suspended state when it suspends itself. If another task suspends it, one of the following occurs:

- The task goes from ready to suspended, or from asleep to asleep-suspended.
- An already suspended or asleep-suspended task remains in the same state, but the its suspension depth is increased by one.

Resume. The task goes from suspended to ready or from asleep-suspended to asleep when the task has a suspension depth of one, and it is resumed by another task.

The task remains suspended or asleep-suspended when resumed by another task while its suspension depth is greater than one. Its suspension depth is decremented.

Delete. The task goes from any state to nonexistence when it is deleted, either by itself or by another task.

Task Priority

A task's priority is a number representing the task's importance relative to other tasks for scheduling purposes. Task priority ranges from 0 through 511; the lower the number, the higher the task's priority. A task's priority is assigned when the task is created and may be changed during the task's life, either by explicit action or by events associated with a region semaphore (described in Chapter 7).

To schedule tasks, the Kernel selects the running task from the set of ready tasks, based upon the tasks' priorities. The Kernel schedules tasks so that the highest priority ready task is given the processor. Setting task priorities gives applications control over scheduling the processor.

Task Priority Operations

A task's basic priority is called its static priority. The Kernel also maintains another priority for the task: the dynamic priority. The dynamic priority normally equals the static priority but may be raised temporarily when using a region semaphore.

The **get_priority** system call returns the static priority of the specified task. Even if the task's dynamic priority has been temporarily adjusted, **get_priority** returns the task's static priority.

The **set_priority** system call changes a specified task's static priority. If that task's dynamic priority has been temporarily adjusted and the requested change would lower the static priority, the change is delayed until the temporary adjustment is cancelled.

Time Slicing

Time slicing is an optional task scheduling feature of the Kernel. Using time slicing, ready tasks of equal priority are scheduled one after another. Each task of the that priority executes up to a maximum length of time before another task of the same priority has a second chance to run. The amount of time is called a time slice. Without time slicing, the running task would execute until it goes to sleep, suspends itself, or a higher priority task becomes ready. In such a situation, any tasks of equal priority could be delayed indefinitely.

After a task executes for its time slice, the Kernel checks for other ready tasks of equal priority. If there are any, the running task goes to the ready state behind tasks of equal priority, and the next task takes control of the processor in a round-robin fashion. After all tasks of equal priority have run, each task has another opportunity to execute for its full time slice. When a task regains the processor after being pre-empted during its time slice, the task only executes for the remainder of that same time slice. The task does not get a new full time slice.

Two configuration parameters control time slicing. The first sets the number of clock ticks that comprise the time slice. The other parameter, the real-time fence, determines which tasks will be time-sliced. Tasks of higher priority than the real time fence are not time-sliced; tasks of equal or lower priority are. With the real-time fence, high priority tasks are scheduled strictly by priority for greater responsiveness, while less privileged tasks are time-sliced, providing fairer allocation of the CPU.

Changing a Task's Time Slice

The application can change any task's time slice on an individual basis. When a task is created, the Kernel fills in its `task_slice` field with the value configured for time-slicing. To change a task's time slice, change the `task_slice` value in the task's task state. The next time the task gets a new time slice, the time slice will be the new value.

See also: **create_task**, *iRMK™ Kernel Reference Manual*

Task Creation

When creating a task, the application must supply an area in memory that will be used for the task state. This task state corresponds to the processor-defined task state segment (TSS) plus additional fields required by the Kernel. The memory allocated to the task object must begin on a four-byte boundary. The memory can be determined statically, or it can come from a memory pool allocated by the Memory Management facilities. An application calls the **create_task** system call and specifies the following information:

- the memory to be used for the task
- the stack the task will use
- the data segment for the task
- the first instruction the task will execute
- the task's priority
- whether the task will be created in the suspended state or the ready state

Privilege level

Privilege level is a hardware-maintained protection feature of Intel386 family processors, when operating in protected mode. The privilege level of a memory segment is the minimum privilege a task must have to access that segment. There are four privilege levels, numbered 0 to 3. Privilege level 0 (also called ring 0), is the most privileged.

The stack pointer, data segment pointer, and code segment pointer specified when calling the **create_task** system call must all point to segments that have the same privilege level. This privilege level determines the privilege level at which the task initially runs. Usually, Kernel tasks exist totally in ring 0.

See also: *386™ DX Microprocessor Programmer's Reference Manual*

The Ready Queue

The ready queue is a priority-based queue consisting of all tasks that are currently in the ready state. The Task Scheduler selects the next task to run from this queue, using the following rules:

- Tasks are ordered by priority in the ready queue.
- When a task is made ready, it is inserted in the ready queue after all other ready tasks of equal priority.
- If a task is pre-empted during its time-slice, it remains ahead of tasks of equal priority.
- When a task's time-slice expires, it is placed after all other ready tasks of equal priority in the ready queue.

Figure 6-2 illustrates how a typical set of tasks moves through the ready queue. "Pr" stands for priority and "RTF" for the real-time fence. Dashed arrows indicate task movement within the ready queue during Time X. The ordering of the tasks reflects the ready queue at the end of Time X; the task at the top of each ready queue is the running task.

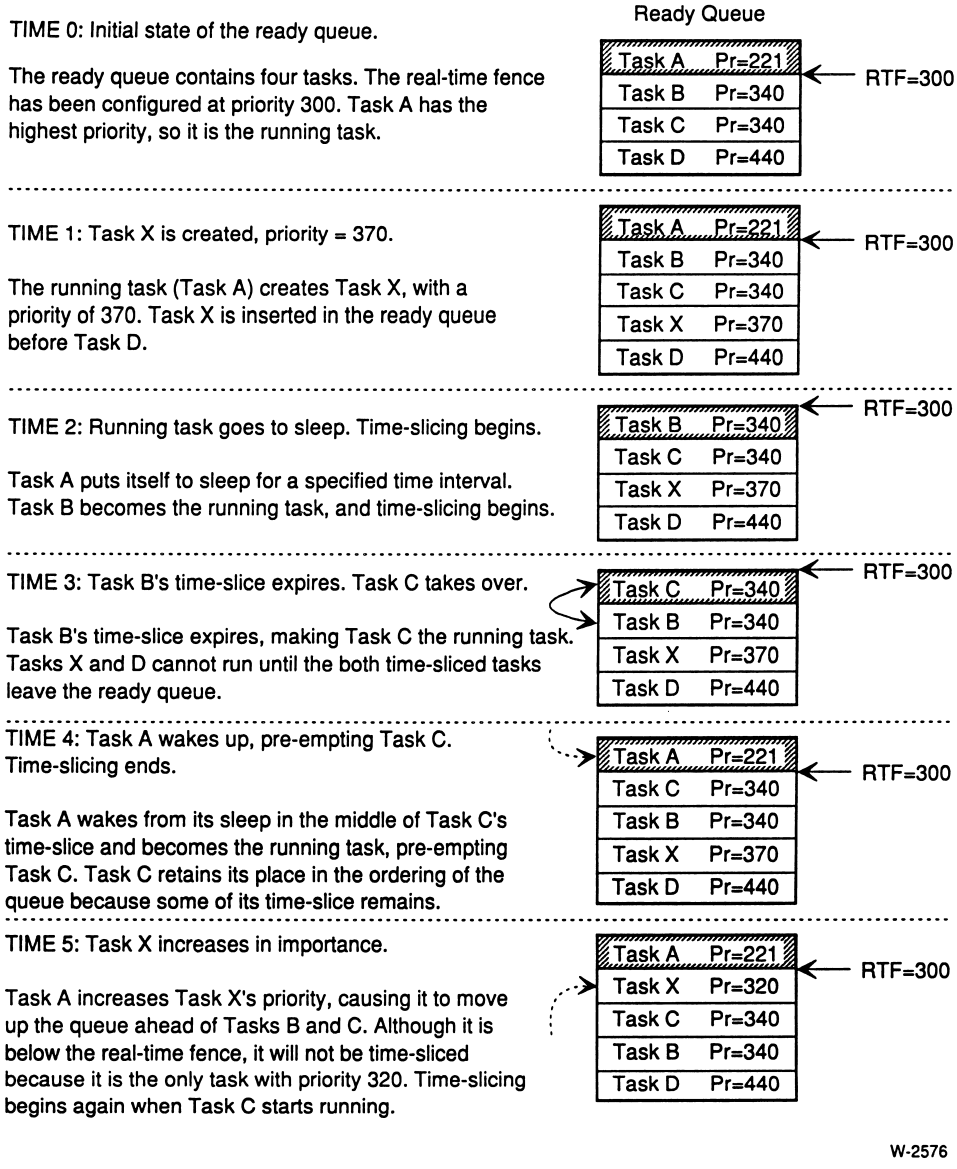


Figure 6-2. Task Scheduling Scenario

Controlling Task Switches

Even though the Kernel determines which task will run at any given moment, it also provides mechanisms for controlling task switches. The following are some of the factors which cause the Kernel to pre-empt the currently running task.

- The executing task may perform some action such as sending a message which causes a higher priority task to become ready.
- The currently running task may use up its allotted time slice.
- An interrupt may occur which affects the state of the currently running task.
- The task may affect its own state through a Kernel system call.

You can protect the currently running task from being pre-empted by using the following techniques.

- disabling interrupts
- performing a scheduling lock

Disabling interrupts prevents a task switch that would result from Kernel calls within an interrupt handler. Disabling interrupts is discussed in the Interrupt Management section later in this chapter.

A scheduling lock also provides protection for the current task. By using a scheduling lock, the running task can ready other tasks without losing control of the processor. The Kernel delays a task switch to the running state until scheduling is resumed. Once it releases the scheduling lock, the running task may be pre-empted.

To lock scheduling, the application calls the **stop_scheduling** system call. Calling **stop_scheduling** multiple times causes multiple scheduling locks to be in effect. To resume scheduling, the application calls **start_scheduling**. The application must call **start_scheduling** once for each lock in effect. In other words, if **stop_scheduling** was called three times, the application must call **start_scheduling** three times to remove all outstanding locks. Task switching resumes after all scheduling locks are removed.

NOTE

A scheduling lock does not prevent task switching in all cases. A system call that causes blocking or rescheduling (see the following section) can initiate a task switch in spite of a scheduling block.

Disabling interrupts and stopping scheduling can cause the system to be less responsive.

Classifications of Task-Switching System Calls

Invoking some Kernel system calls can cause a task to become ready; if the ready task has a higher priority than the running task, a task switch occurs. Before making the system call, the application must determine whether a task switch is appropriate and perform a scheduling lock if necessary. Kernel system calls can be classified into four categories, based on their ability to trigger task scheduling. They are:

- Non-scheduling system calls never cause rescheduling to occur. **Create_semaphore** is an example of a non-scheduling system call.
- Signalling system calls can put tasks into the ready queue and potentially cause rescheduling. **Send_unit** is an example of a signalling system call. If invoking such a system call would cause a higher priority task to become ready, the running task can use a scheduling lock to keep control of the processor.
- Blocking system calls can cause the Kernel to put the running task to sleep (block) and thus initiate a task switch. The developer must be aware of whether the system calls in this category will actually cause the running task to block. **Receive_unit** is an example of a blocking system call. Using **receive_unit**, rescheduling occurs unless the unit is actually available. A scheduling lock does not prevent a system call in this category from causing rescheduling.
- Rescheduling system calls always cause rescheduling or will cause rescheduling when invoked on the running task, regardless of a scheduling lock. For example, **sleep** always causes rescheduling, and **delete_task** causes rescheduling when invoked on the running task.

Table 6-1, on page 6-28, classifies the system calls according to their effect on Kernel scheduling and their appropriateness for use by an interrupt handler.

The Task State Segment

The task state is a Kernel-maintained structure that includes the processor register values kept in the task state segment (TSS). It also includes information such as numeric coprocessor registers, the task token, task time slice, task priority, and task flags.

See also: TSS, *386™ DX Microprocessor Programmer's Reference Manual*

When a task is pre-empted, the Kernel updates the task state structure with the latest information about the task before it lets the second task run. When the first task is ready to run again, the Kernel restores all information from the task state so that the task can continue exactly where it left off.

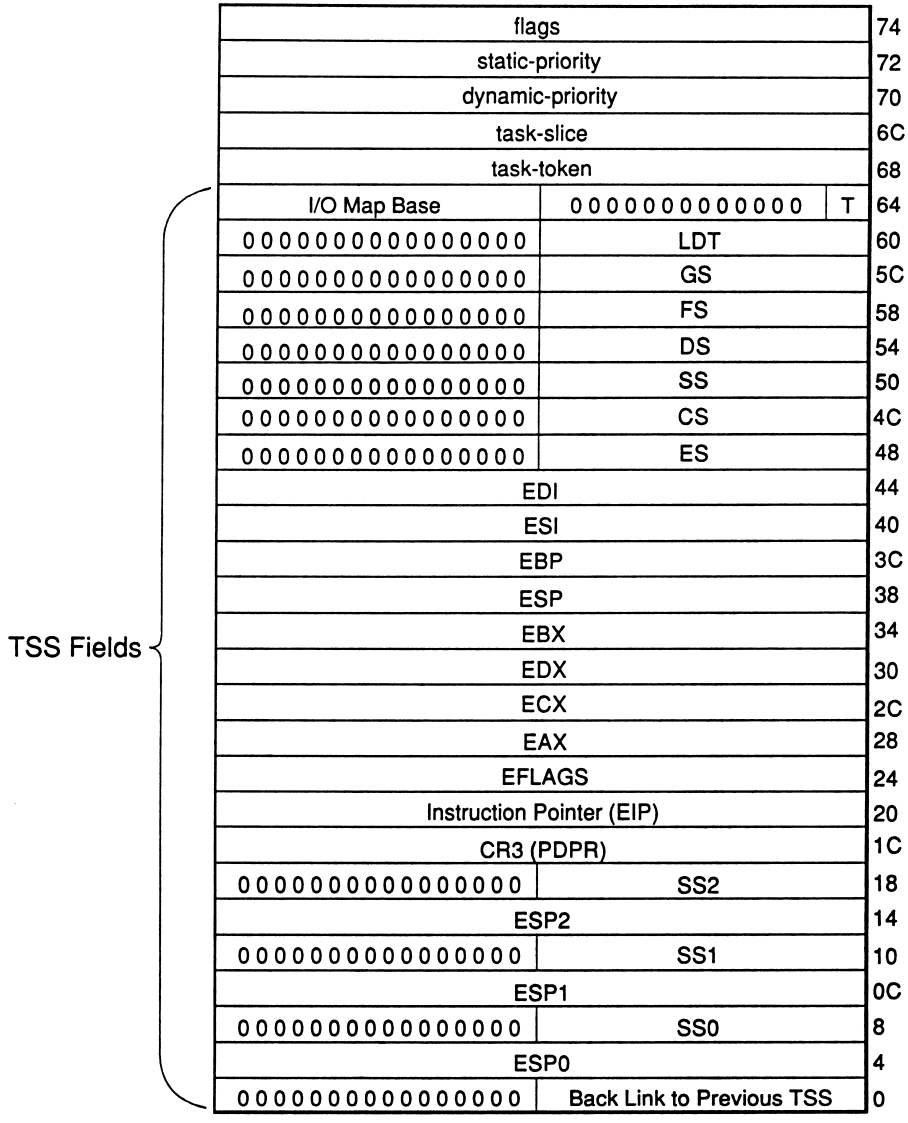
Applications can access a task's task state using the structure KN_TASK_STATE (see the **create_task** system call in the *iRMK™ Kernel Reference Manual*). Applications have access to the TSS portion and to the following additional fields of the KN_TASK_STATE structure:

- the token for the task
- the task's static and dynamic priorities
- the task's time slice
- flags indicating whether or not the task is the idle task (a Kernel-supplied dummy task that exists to ensure that there is always a task ready to run)

Figure 6-3 illustrates a task's task state. The application should modify only the fields designated as safe in the **create_task** description. Changing any of the other fields causes undefined results.

There are several ways to get a pointer to a task's task state:

- The Kernel maintains a pointer to the current running task in a public variable called KN_CURRENT_TASK.
- The **token_to_ptr** system call returns the pointer to the task state when supplied with the task token.
- Various Kernel handler interfaces, such as the **task_switch_handler**, provide pointers to relevant tasks.
- You may use the pointer to the task's task state supplied at task creation.



Note: 0 means Intel reserved. Do not define.

W-2577

Figure 6-3. Task State Elements

The Kernel manages task switching rather than allowing the processor to switch tasks. Figure 6-4 shows how the processor manages a task switch by using multiple TSS descriptors. The processor switches tasks by changing the task register to point to a different descriptor.

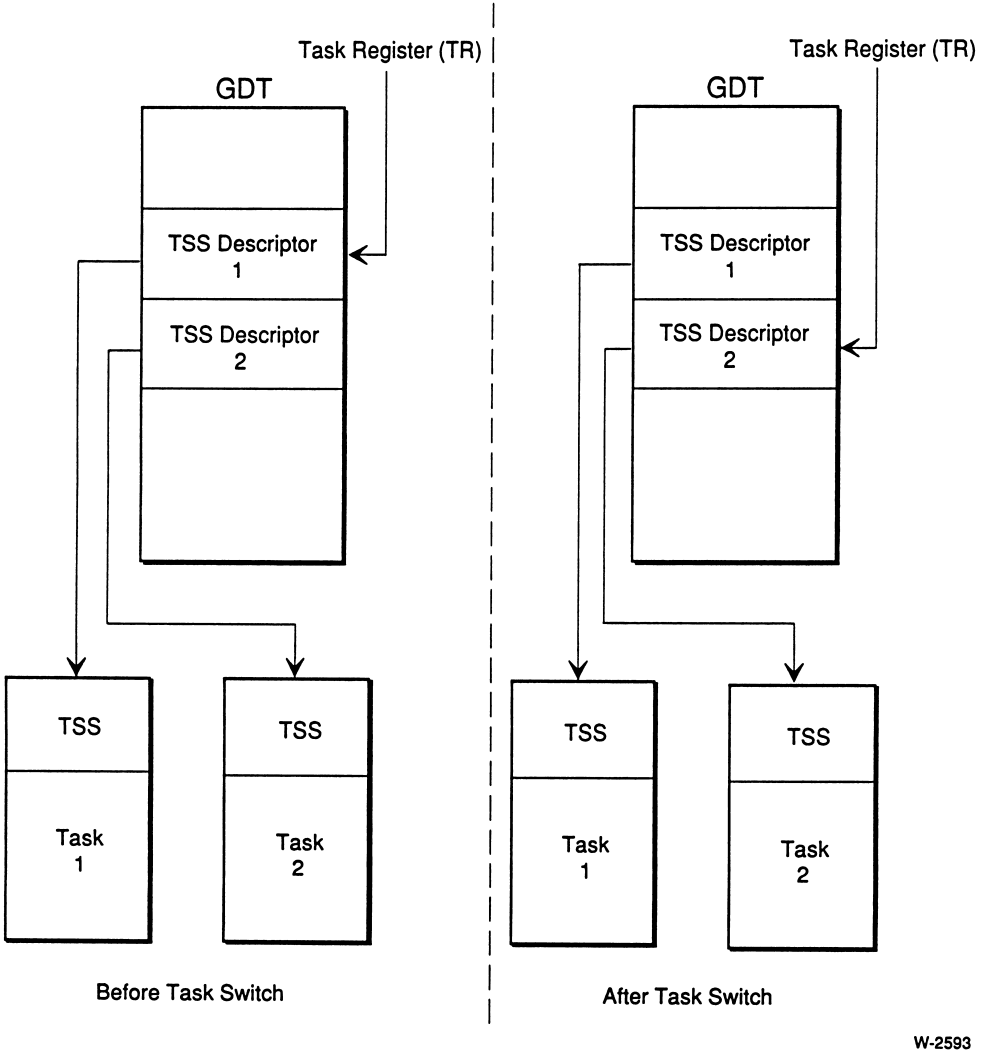


Figure 6-4. Processor Task Switch

Figure 6-5 shows how the Kernel manages a task switch by rewriting a single TSS descriptor. This gives the Kernel more control over the task switching process.

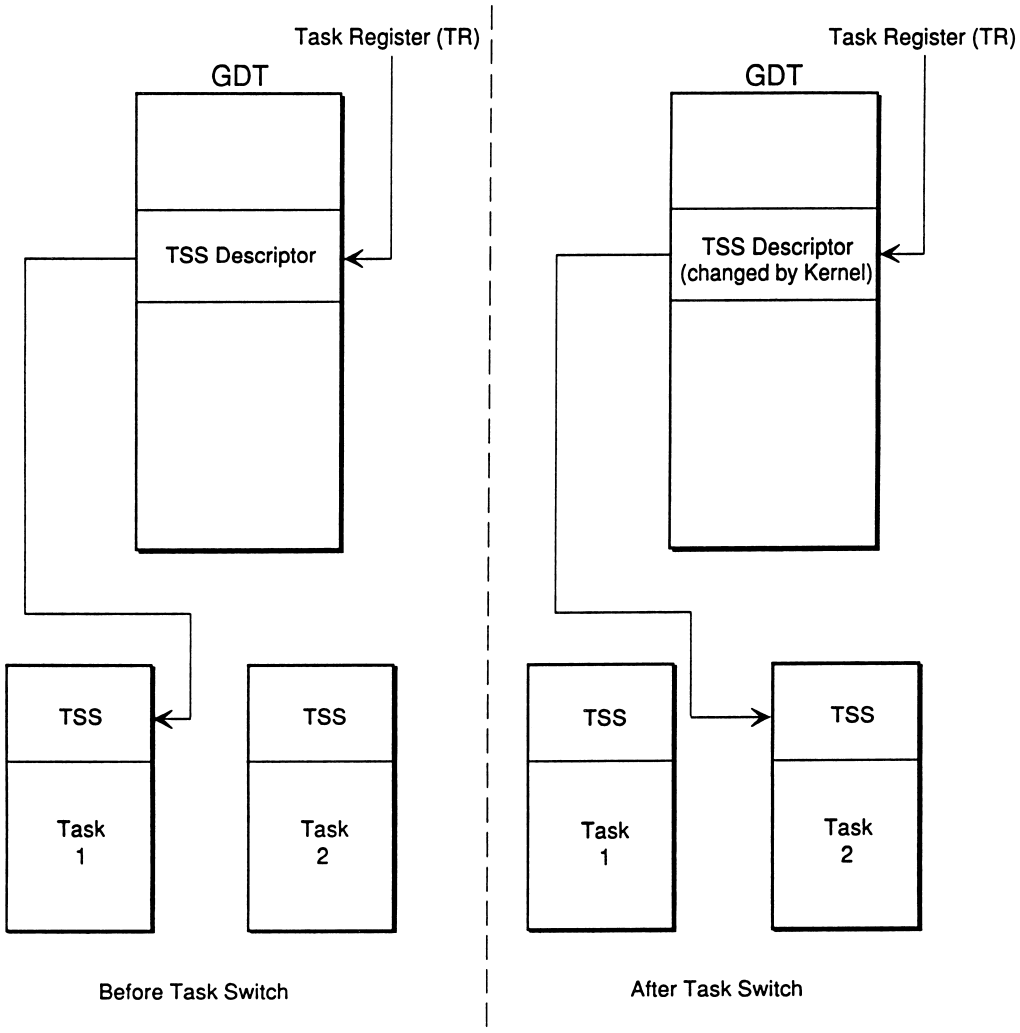


Figure 6-5. Kernel Task Switch

Task Deletion

Using the **delete_task** system call, applications may delete any task, including the calling task (that is, the running task may delete itself). Tasks may be in any task execution state when deleted, but to preserve system consistency, they should be inactive (that is, not executing any Kernel system calls). When **delete_task** returns, all resources of the deleted task are available for reuse. If the running task deletes itself, **delete_task** does not return.

Task Handlers

You may configure the Kernel to invoke application procedures, called task handlers, to perform additional functions during the following situations:

- Task creation
- Task deletion
- Task switching
- Disaster handling
- Certain priority changes

These handlers can be used to enhance the Kernel operations. By providing these procedures, you can add functionality and/or handle error situations. For example, if the application requires a hierarchical structure for tasks, it can use a task handler to implement the setup when the task is created. An application that needs to mask certain interrupts when a task's priority changes can use a priority change handler.

The application may install task handlers at Kernel initialization, or dynamically with the **set_handler** system call. Using **set_handler**, you may install multiple handlers of each type except the disaster handler, which must be installed in the **initialize** system call. The **reset_handler** system call dynamically removes a user-supplied task handler.

See also: Kernel Handlers, *iRMK™ Kernel Reference Manual*

Using Task Handlers for Multiple Privilege Levels

The Kernel assumes a single privilege level for tasks. Task handlers give you the ability to override this assumption, allowing application tasks to run at multiple privilege levels.

All tasks that call the Kernel must have a stack for privilege level 0, even if the task is created with a different initial privilege level. In addition, each task must have a stack for each privilege level in which it executes. However, since the Kernel assumes one privilege level, it only sets up a pointer to one stack in the task's task state.

To run a task at more than one privilege level, you must set up additional stacks for the task by assigning values to the SSi and ESPi fields (where i = 0 through 2) in the TSS. A task creation handler can do this work automatically upon task creation.

At task deletion, a task deletion handler would then undo whatever the task creation handler set up at task creation. For example, memory for additional stacks might be deallocated.

Summary of Task Management System Calls

The following is a list of system calls for Kernel Task Management:

- The **create_task** system call creates a new task using the supplied resources. The Kernel assigns this new task a task state and, by default, gives it the same LDT as the calling task. The caller specifies the memory area to be used for the task object, the stack to be used by the task, the first instruction to be executed by the task, and the priority of the task.
- The **current_task_token** system call returns the token for the currently executing task.
- The **delete_task** system call deletes the specified task regardless of the task's current execution state. All resources dedicated to the deleted task are available for reuse.
- The **get_priority** system call returns the static priority of the specified task.
- The **resume_task** system call cancels one level of suspension for the specified task. If the suspension depth for the task is one when this system call is invoked, the Kernel removes the task from the suspended state and puts it in the ready state.
- The **set_priority** system call changes the static priority of the specified task.
- The **start_scheduling** system call cancels one scheduling lock imposed by **stop_scheduling**. When **start_scheduling** cancels the last outstanding schedule lock, the Kernel carries out all delayed scheduler task state transitions.
- The **stop_scheduling** system call temporarily locks the scheduling mechanism (or places an additional lock on the mechanism) for the running task. Any scheduler task state transitions that would move a task from the running state to the ready state are delayed until scheduling is resumed. **Stop_scheduling** does not prevent a task switch if the running task becomes blocked or calls one of the rescheduling system calls. Refer to Table 6-1 on page 6-28 for a list of blocking and rescheduling system calls.
- The **suspend_task** system call puts the specified task in the suspended state (asleep-suspended if the task is currently asleep). If the task is already suspended, its suspension depth is increased by one.

The following system calls implement task handlers:

- Task handlers may be set up in the **initialize** system call. This is the only system call that can be used to set up a disaster handler.
- The **set_handler** system call dynamically installs a user-supplied task handler. Multiple task handlers of each type may be installed by invoking **set_handler** multiple times.
- The **reset_handler** system call dynamically removes a user-supplied task handler previously set by the **set_handler** system call or during Kernel initialization.

Interrupt Management

Interrupts and interrupt processing are fundamental to a real-time operating system. An interrupt is a signal to the processor that an asynchronous event has occurred. The Kernel allows applications to provide interrupt handlers to respond to interrupts. Interrupt handlers are procedures that execute using the resources of the current running task, rather than requiring a task switch. Since interrupts are typically disabled while a handler executes, interrupt handlers must work quickly and exit. The interrupt processing features of the Kernel enable the user to create applications that respond quickly and flexibly to external interrupts.

The Kernel Interrupt Model

Figure 6-6 illustrates the interrupt model used by the Kernel. Interrupts originate from external sources or devices. These sources are connected to some configuration of Programmable Interrupt Controllers (PICs). A PIC has one output pin and a number of input pins connected to interrupt sources or to the outputs of other PICs.

A typical configuration includes a master PIC with a number of slave PICs connected to it. The master PIC is connected, in turn, to the processor. Each interrupt source connected to the PICs is associated with a specific slot in the Interrupt Descriptor Table (IDT). The gate descriptors in the slots of the IDT point to interrupt handlers. Each handler services the interrupt source with which it is associated.

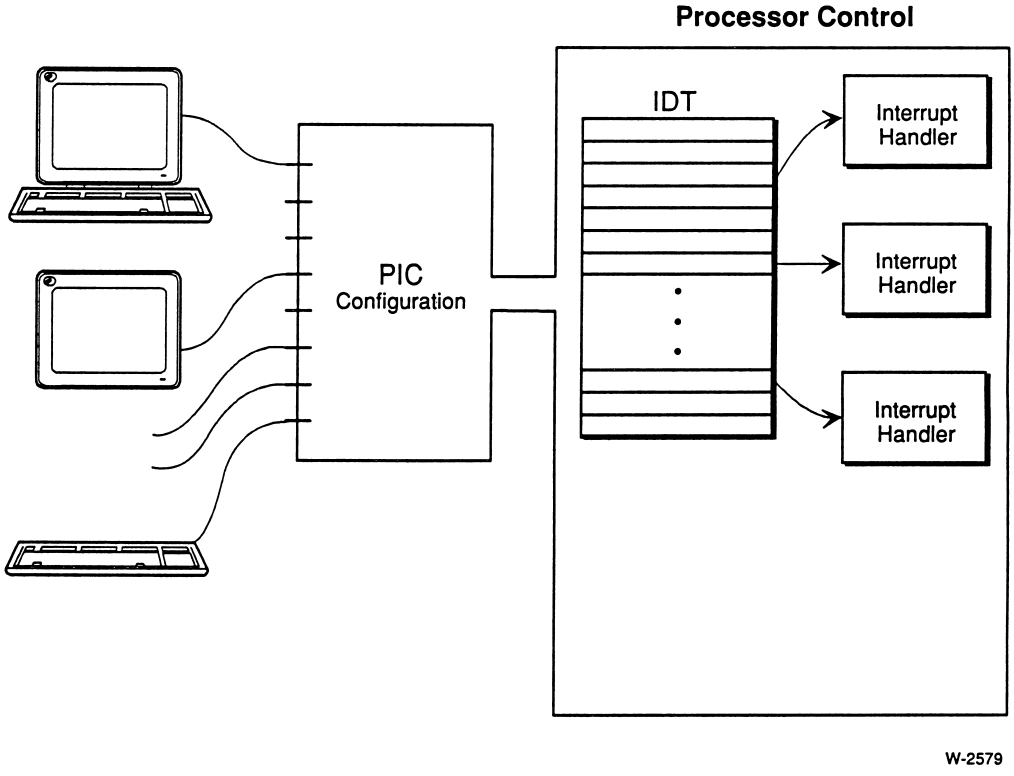


Figure 6-6. Kernel Interrupt Model

The Interrupt Descriptor Table

The Interrupt Descriptor Table (IDT) is a table of up to 256 interrupt gate descriptors. These descriptors point to interrupt service routines (interrupt handlers). One descriptor corresponds to one interrupt source. The Kernel supports only interrupt gates for servicing interrupts.

The definition of the IDT also allows it to contain task gates, which cause task switches when interrupts occur. However, the Kernel does not support using task gates for servicing interrupts.

Interrupt Sources and Slots

Interrupt sources are external devices wired to one or more interrupt controllers in an ordered manner. The interrupt controllers and the way they are connected to one another determine which interrupt will be presented to the processor when multiple interrupts occur at the same time. Thus, ordering the interrupt sources establishes the priority of the sources.

When an interrupt controller detects an interrupt on one of its input lines, it signals the processor. The processor then interrogates the interrupt controller to determine the source. The interrupt controller returns a number, which the processor uses as an offset into the IDT. The IDT entry directs program control to the correct interrupt handler.

To set up an interrupt handler for a particular interrupt, you must set up an interrupt gate for the handler in the appropriate slot of the IDT. How the slots correspond to interrupt sources is determined during Kernel configuration. The gates in the IDT may be installed using the Builder, or they may be set up dynamically, using the **set_interrupt** system call.

Interrupt sources are specified to Kernel system calls using the IDT slot number associated with the source, rather than the location of the interrupt source within the interrupt controller configuration. This allows Kernel applications to remain independent of PIC configuration, so you could change interrupt source configuration without having to change interrupt handler routines.

The Kernel does not force the order of interrupt sources. How you wire the sources into the system and how you configure the PICs controls the ordering. However, Intel recommends that interrupt sources be ordered according to their importance, so that the lowest IDT slot number corresponds to the highest priority interrupt source. By ordering the sources in this way, the more important interrupt is processed first.

If you intend to use the Kernel PIC Management (described in Chapter 7), the system call **new_masks** in that module assumes that higher priority sources are associated with lower IDT slot numbers.

Using Interrupt Handlers

Interrupt handlers are procedures invoked by the hardware to service interrupts. Interrupt handlers use the address space of the current running task. Because there is no task switch, the interrupt handler responds much faster to the interrupt than would a task responding to the interrupt. However, because all interrupts are disabled while the handler executes, the handler must work quickly and exit. Otherwise, other interrupts could be lost or delayed. Figure 6-7 illustrates the flow when an interrupt handler is invoked.

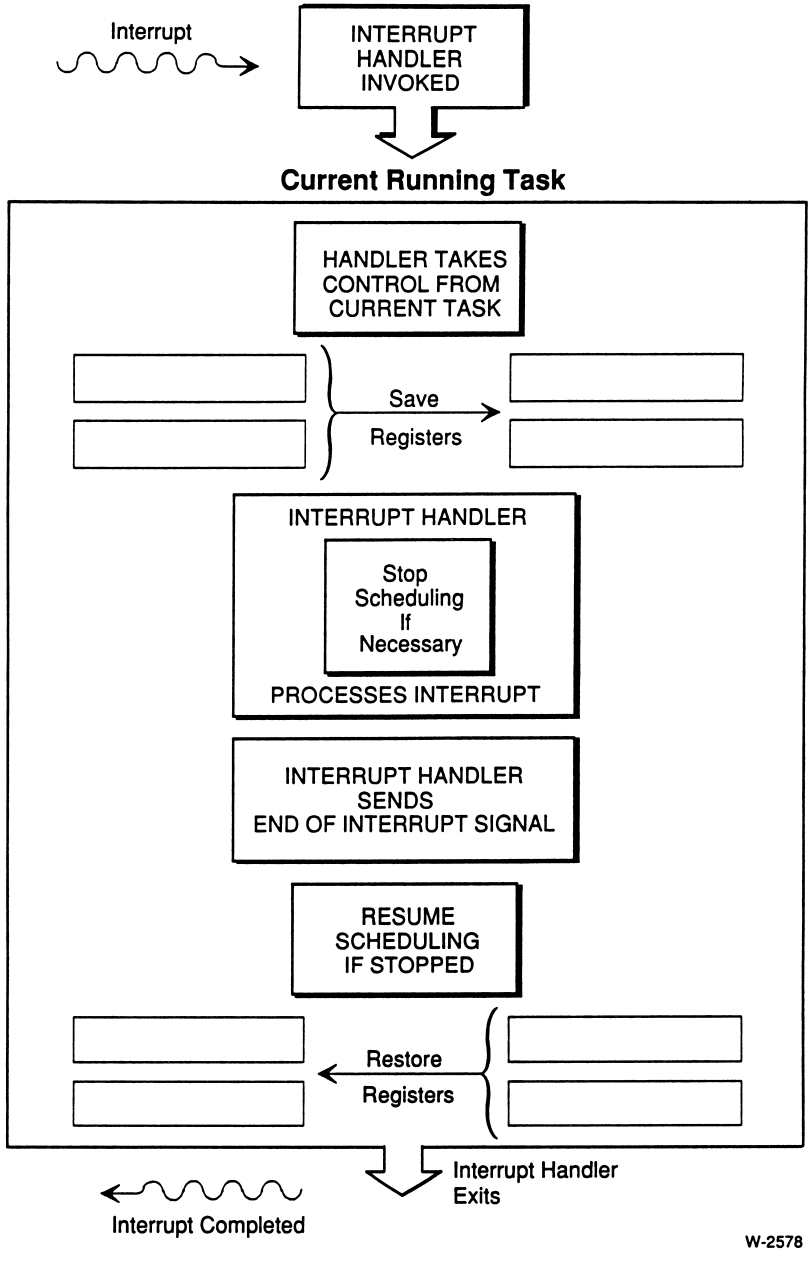


Figure 6-7. Control Flow of Interrupt Handlers

For an interrupt handler to execute, it must save everything the interrupted task needs to resume running. The handler saves the task's registers when it takes control, and restores them when it exits.

When the processor detects an external interrupt, the hardware automatically stops executing the current task and passes control to the interrupt handler. The interrupt handler then services the hardware associated with that interrupt level. This servicing may involve, for example, fetching data associated with the interrupt or sending data to a device.

Interrupts are disabled while an interrupt handler services an interrupt to prevent the handler from losing control of the processor. However, if servicing the interrupt keeps interrupts disabled too long, the handler can have the Kernel enable interrupts but stop scheduling other tasks. This allows interrupts to be serviced but still prevents other tasks from gaining control of the processor. When the servicing is finished, the interrupt handler resumes scheduling. Any scheduler task state changes made by the handler will then take place.

The handler must send an end-of-interrupt signal which allows the interrupt controller to generate another interrupt. If the handler has stopped scheduling, it should send the end of interrupt signal before it resumes scheduling. Otherwise, the interrupt level may be shut off for an indefinite time.

The following sections describe the processes associated with interrupt handlers.

Establishing a Handler in an IDT Slot

There are two ways to establish an interrupt handler in an IDT slot:

- Set up gates and assign IDT entries using the Builder.
- Use the **set_interrupt** system call to establish the interrupt handler's call gate in the IDT slot assigned to the interrupt source. The **set_interrupt** system call overrides any previous gate in that slot.

Saving the Context of a Task

Interrupt handlers must preserve the registers of the tasks they interrupt. Different programming languages vary in their ability to save the context of a task. In assembly language, interrupt handlers must save the register values and then restore those values before exiting. In PL/M, declaring a procedure to be an INTERRUPT procedure automatically causes the procedure to save the registers. Some versions of C that support interrupt handlers also automatically save the registers. If the register values are not saved automatically, you must write an "envelope" procedure in assembly language or PL/M to save them. The envelope could then call a C procedure to process the interrupt.

Figure 6-8 illustrates the stack after an interrupt handler has preserved the context of the interrupted task. The flags in Figure 6-8 mark the top of the interrupted task's information. When the interrupt handler exits, all information including the flags will be popped off the stack and the registers will be restored to their values before the interrupt.

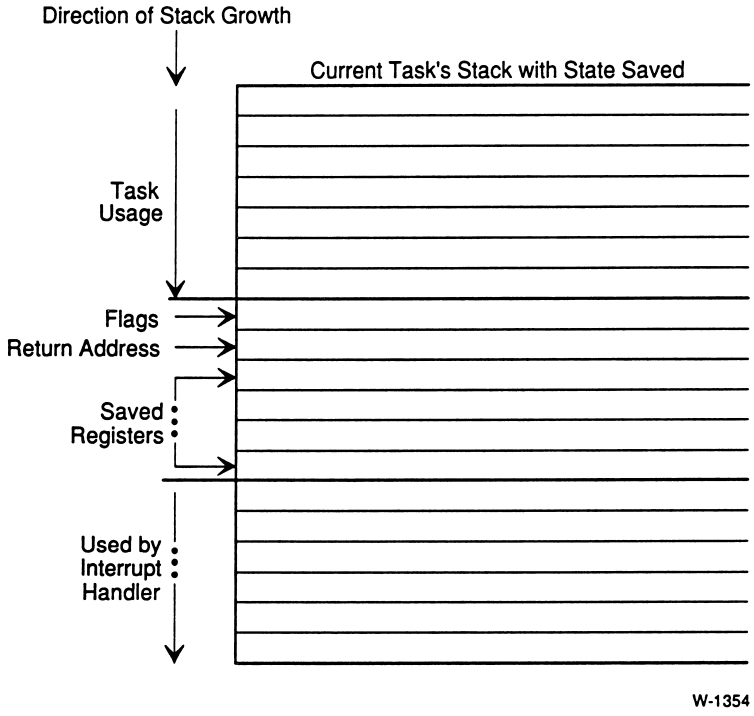


Figure 6-8. Current Task's Stack with State Saved

Interacting with Interrupt Controllers

Because the Kernel is independent of peripheral devices, applications must manage the interrupt controller directly. However, the Kernel provides optional modules for managing 8259A Programmable Interrupt Controllers and the 82380/82370 Integrated System Peripheral. These PIC Manager modules (described in Chapter 7) handle low-level programming details.

Restrictions on System Calls Used by Interrupt Handlers

An interrupt handler frequently interacts with tasks in the system. However, if a task switch occurs and the interrupt handler loses control of the CPU, either deadlock or a loss of interrupts can occur. Table 6-1 lists the Kernel system calls according to their safeness for use in interrupt handlers.

Table 6-1. Classifications of Kernel System Calls

Non-scheduling/Safe		
create_alarm	get_priority	null_descriptor
create_mailbox	get_slot	ptr_to_linear
create_pool	get_time	reset_alarm
create_semaphore	initialize_interconnect	reset_handler
csts	initialize_LDT	send_EOI
current_task_token	initialize_NDP	set_descriptor_attributes
delete_alarm	initialize_PICs	set_handler
delete_pool	initialize_PIT	set_interconnect
delete_task[1]	initialize_stdio	set_interrupt
get_code_selector	initialize_subsystem	set_time
get_data_selector	linear_to_ptr	start_PIT
get_descriptor_attributes	local_host_ID	stop_scheduling
get_interconnect	mask_slot	token_to_ptr
get_PIT_interval	mp_working_storage_size	translate_ptr
get_pool_attributes	new_masks	unmask_slot

Signalling	Blocking	Rescheduling/Unsafe
create_task	attach_protocol_handler	attach_receive_mailbox
delete_mailbox	ci	cancel_dl
delete_semaphore	co	cancel_tp
resume_task	create_area	delete_task[1]
send_data	delete_area	initialize
send_priority_data	receive_data	initialize_console
send_unit	receive_unit	initialize_RDS
set_priority		initialize_message_passing
start_scheduling		send_dl
suspend_task[1]		send_tp
tick		sleep
		suspend_task[1]

[1]suspend task and delete_task cause rescheduling when the system call performs these operations on the calling task.

The categories shown in Table 6-1 are described below. The *iRMK™ Kernel Reference Manual* specifies the category in the description of each system call.

Non-scheduling or Safe System Calls

Non-scheduling system calls are "safe" system calls because they never cause rescheduling. Interrupt handlers may use them without restriction.

Signalling System Calls

Signalling system calls can put tasks into the ready state and thus initiate rescheduling, causing the running task to be pre-empted. Signalling system calls should only be used by interrupt handlers if the rescheduling mechanism is stopped using the **stop_scheduling** system call. Any tasks that the signalling operation awakens cannot pre-empt the running task until rescheduling is enabled using the **start_scheduling** system call. The **send_unit** system call is an example of a signalling system call.

Interrupt handlers may call signalling system calls without scheduling locked if the system calls will not cause scheduler task state transitions. For example, an interrupt handler may invoke **send_data** without stopping scheduling if it knows that no task is waiting at the target mailbox.

The safest way to use signalling system calls, however, is to stop scheduling, invoke the signalling system call, and then later resume scheduling.

Blocking System Calls

Blocking system calls can put the running task into the asleep state and thus initiate a task switch. As soon as a running task is put to sleep the execution of some other task begins. **Receive_unit** and **receive_data** are examples of blocking system calls.

Interrupt handlers may invoke blocking system calls only if the system calls will not put the running task to sleep. For example, an interrupt handler may call the **receive_data** system call to receive a message if it knows there is a message waiting or if the handler will not wait for a message. Otherwise, the results are undefined.

A scheduling lock will not stop a blocking system call from putting the running task to sleep. Even though scheduling is locked, once a task blocks, scheduling is restarted. When the task runs again, the scheduling lock will be restored. It is not recommended to invoke blocking system calls with scheduling locked.

Rescheduling or Unsafe System Calls

Unsafe system calls always cause rescheduling and can never safely be used by interrupt handlers. Examples of such system calls are the **sleep** system call and the **delete_task** system call when **delete_task** is performed on the running task.

Additional Interrupt Servicing

It is sometimes appropriate for tasks to perform additional interrupt servicing beyond that performed by an interrupt handler. In some cases, if an interrupt handler completely serviced an interrupt, interrupts might be disabled for too long. If the algorithm needed to service an interrupt cannot be used within the restrictions on interrupt handlers, then a task must be used.

To share interrupt servicing with a task, the interrupt handler performs the critical services with all interrupts disabled. For example, the handler could stop scheduling; then send a message via a mailbox to direct a task to continue interrupt servicing. When the handler finishes its critical duties, it signals the end-of-interrupt, releases the scheduling lock, and exits. The interrupt task can then receive the message from the mailbox, gain control of the processor, and complete servicing the interrupt. The following sequence of calls shows how to perform the scheduling lock:

```
KN_stop_scheduling()

status = KN_send_data(mailbox,data_ptr,length)

<signal end of interrupt for interrupt slot >

KN_start_scheduling()
```

A screen editor exemplifies how this process works. Each character typed at the keyboard activates an interrupt to the processor. The interrupt handler accepts an input character, decides if the character needs special handling or simply needs to be buffered, and signals the end of the interrupt. When the buffer fills, the handler stops scheduling and sends a mailbox message to an interrupt task indicating that the buffer is filled and needs processing. Then the handler sends the end-of-interrupt signal, and resumes scheduling. With scheduling resumed, the interrupt task gains control of the processor and processes the text. Figure 6-9 illustrates this flow of events.

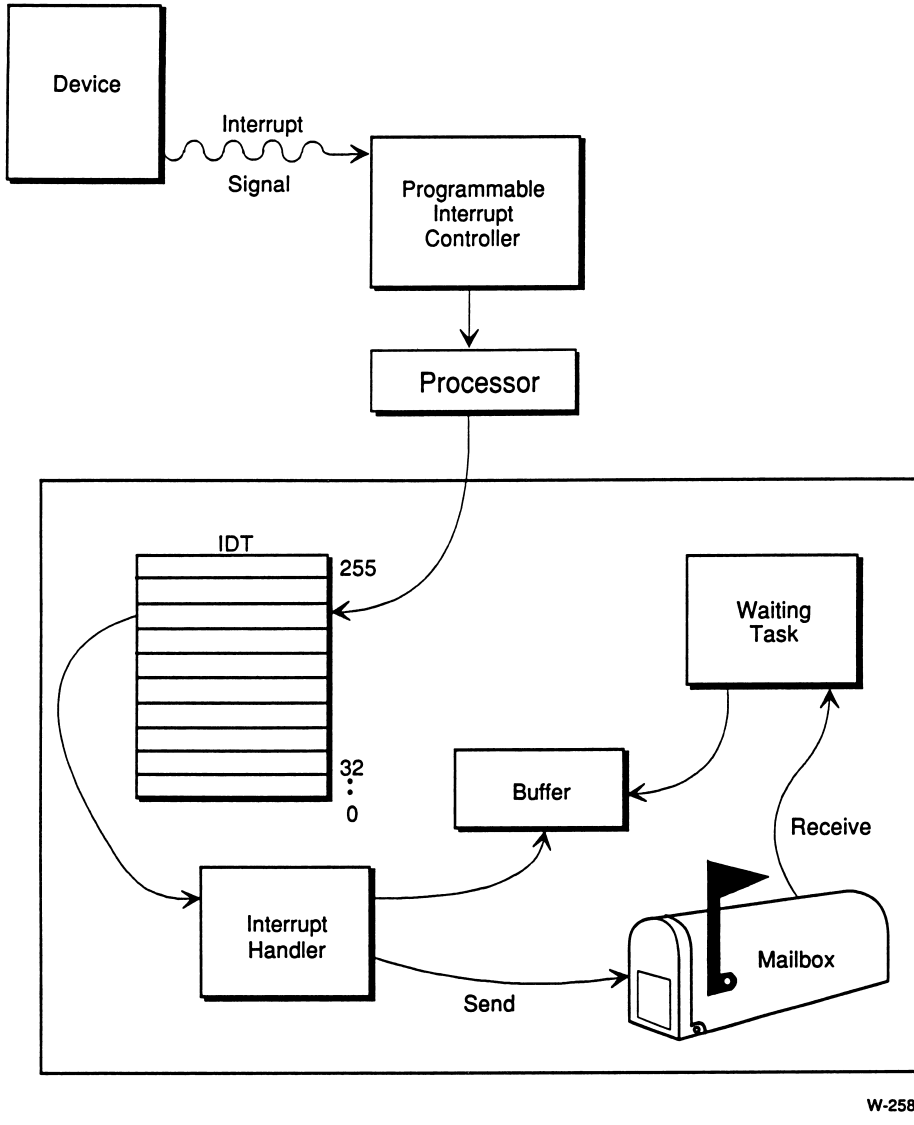


Figure 6-9. Additional Interrupt Servicing

Summary of Interrupt Management System Calls

The **set_interrupt** system call establishes an interrupt handler for the specified hardware interrupt source. It places an interrupt gate referring to that interrupt handler in the specified IDT slot.

The following system calls related to interrupts are discussed in Chapter 7, under Programmable Interrupt Controllers:

- initialize_PICs
- send_EOI
- new_masks
- get_slot
- mask_slot
- unmask_slot

Time Management

The Kernel provides time management facilities that allow tasks to create virtual timers (alarms) and to sleep for a specified amount of time. The Kernel also provides a real-time clock.

Clock Ticks

All time values in the Kernel are specified in units of clock ticks. It is up to the application to determine the frequency and accuracy of the clock ticks.

The Kernel needs an external source of periodic signals to implement its time management facilities because the processor does not generate such signals internally. The Kernel depends on the application to provide these time signals rather than requiring a particular peripheral device in the system. An application can supply these signals automatically by using one of the PIT manager modules (8254 or 82380/82370). If one of these PIT managers is not used, the application supplies the signal directly through the **tick** system call. Each call to **tick** advances the Kernel time by one unit (clock tick). The Kernel has no specific expectations of how clock ticks are generated. Usually a programmable interval timer (PIT) generates periodic interrupts which can be used as a source for clock ticks.

Using the Real-Time Clock

The Kernel provides a real-time clock by counting clock ticks. The **get_time** system call returns the current value of the counter. The value of the real time clock is set to zero at initialization. The application may set the count to any value by using the **set_time** system call.

Measuring Elapsed Time

The application can measure elapsed time by reading the real-time clock with the **get_time** system call at the beginning and end of the interval to be measured. By subtracting, the application determines the elapsed time.

Alarms

The Kernel provides the ability for tasks to create alarms, or virtual timers, to simulate timer interrupts. Alarms invoke user-supplied alarm handlers. Alarm handlers are automatically called with scheduling stopped and interrupts disabled.

There are two types of alarms: single-shot alarms and repetitive alarms. A single-shot alarm invokes its alarm handler once when its time interval elapses. The alarm then becomes inactive and its memory can be re-used. A repetitive alarm, as the name implies, invokes its alarm handler after its time interval elapses and then resets itself for the same time interval. It continues to invoke its handler until the alarm is explicitly deleted.

Creating and Deleting Alarms

An alarm is created with the **create_alarm** system call and deleted with the **delete_alarm** system call. When you create an alarm, specify the following:

- the area in which the alarm object will exist
- whether it is a single shot or a repetitive alarm
- the time interval for which the alarm is set
- a pointer to the application handler that the alarm invokes when the time period elapses

After a task calls the **delete_alarm** system call, the handler associated with that alarm will no longer be invoked and the memory that the alarm occupies can be reused. Alarms may be deleted whether or not they have gone off (that is, whether or not they have invoked the associated alarm handler). This means that deleting an alarm does not have to be synchronized with the expiration of the alarm.

Resetting Alarms

The **reset_alarm** system call resets an alarm, returning it to its initial state. **Reset_alarm** uses the alarm's token; the alarm parameters are not required. Both single-shot and repetitive alarms can be reset.

Resetting an alarm is functionally equivalent to a **delete_alarm** followed by a **create_alarm** with the original parameters. Regardless of whether the alarm's time limit has expired, resetting an alarm returns it to its creation state and starts it running as if it were just set. Resetting a single-shot alarm after it has gone off is equivalent to setting the alarm again.

Sleep

The Kernel provides the capability for tasks to sleep for a specified time, using the **sleep** system call. The amount of time the task will be in the asleep state can vary from no time (KN_DON'T_WAIT) to forever (KN_WAIT_FOREVER). If the specified time is KN_DONT_WAIT, the task will not go to sleep. A KN_DONT_WAIT time limit gives the processor to another task of equal priority (assuming one exists).

Designating a KN_WAIT_FOREVER time limit, on the other hand, means the task will never wake up. When a task sleeps forever, it is effectively deleted, but its memory is not released. The KN_WAIT_FOREVER literal is more appropriately used with blocking system calls, indicating that the task will wait until an event occurs to wake it. For example, a task might "wait forever" until a message arrives at a mailbox.

Summary of Time Management System Calls

The following is a list of Kernel Time Management system calls:

- The **create_alarm** system call creates and starts a virtual alarm clock. Use it to specify a time limit and a task handler to be invoked when the time limit elapses.
- The **delete_alarm** system call deletes an existing alarm. As a result of this call, the handler associated with the alarm will not be invoked. The memory area occupied by the alarm is available for reuse.
- The **get_time** system call returns the current value of the counter that the Kernel uses to keep track of the number of clock ticks that have occurred.
- The **reset_alarm** system call returns a existing alarm to its creation state. In all cases, this operation is equivalent to invoking the **delete_alarm** system call followed by invoking the **create_alarm** system call.
- The **set_time** system call sets the value of the counter that the Kernel uses to keep track of the number of clock ticks that have occurred. At Kernel initialization, the Kernel sets count to zero.
- The **sleep** system call puts the calling task in the asleep state for the specified number of clock ticks.
- The **tick** system call notifies the Kernel that another clock tick has occurred. Normally, users use the **tick** system call in applications that have their own PIT handlers and omit the Kernel's PIT Management module.

***

USING OPTIONAL KERNEL SERVICES 7

Including Optional Modules

The Kernel's basic modules are described in Chapter 6. Support for those modules is always present, regardless of the calls made by the application. This chapter describes the Kernel's optional features. Optional features are modules that are only built into the Kernel if the application calls them. Although many of these "optional" modules may be mandatory for your application, this approach allows a more compact Kernel to be built by applications that do not require some features.

To include an optional module in your system, you simply call one of that module's system calls in your application. Some modules require an initialization call to set configuration values. When you bind your application to the Kernel, the appropriate modules are automatically bound into the system. If an optional module is not referenced by the application, that module is not bound with the system.

If more than one module provides the same type of service, for instance the module for the 8259A and the module for the 82380, the particular device is chosen by a configuration value. The *iRMK™ Kernel Reference Manual* contains configuration information for optional modules.

The Kernel's optional features include:

- Semaphores
- Mailboxes
- Memory Management
- Device Management, including support for programmable interrupt controllers (PICs), programmable interval timers (PITs), numeric coprocessors, and serial I/O
- Descriptor Table Management
- Interconnect Space Management (a Multibus II feature)
- Message Passing control between Multibus II hosts

Device Manager and Interface Support Files

The device managers described in this chapter may be modified to produce custom device managers. Table 7-1 lists general and specific files used to produce the various device managers. Files with the extension *.a38* are ASM386 source; files with the extension *.p38* are PL/M-386 source. To modify these files you must use the appropriate tool. The files shown in Table 7-1 are in directory */usr/intel/rmk/system/src*.

The files in the Interface Library and General Include lists are used to produce the interface libraries *c_call.lib* and *plm_call.lib*. The *makefile* produces object files and interface libraries.

Table 7-1. Device Manager and Interface Support Files

makefile	**General Include**	**General PIC Support**	**82258 ADMA Specific**
	exports.equ	asm_8259.a38	dma82258.p38
Interface Library	exports.ext	asm_8259.inc	dm.inc
interfac.mac	exports.lit	pic_sppt.a38	
xfc_387.a38	kernel.mac	pic_sppt.inc	**380 PIT Specific**
xfc_8254.a38	schedulr.edf	pic_sppt.equ	pit82380.p38
xfc_8259.a38	schedulr.equ	pic_sppt.ext	pit_380.inc
xfc_addr.a38	semaphor.lit	pic_sppt.lit	
xfc_base.a38	taskman.equ		**380 PIC Specific**
xfc_intc.a38	stdtypes.lit	**8259 Specific**	pic82380.p38
xfc_mail.a38		plm_8259.inc	pic_380.inc
xfc_mem.a38	**General PIT Support**	plm_8259.p38	
xfc_msgs.a38	pit_sppt.a38		**380 DMA Specific**
xfc_sem3.a38	pit_sppt.inc	**82530 USART Specific**	dma82380.p38
xfc_sems.a38		usart.lit	dma_380.inc
xfc_usrt.a38	**8254 Specific**	sm_354a.inc	
	plm_8254.inc	sm354a.p38	
	plm_8254.p38	std354.lit	

Task Communication and Synchronization

Tasks frequently need to communicate, share resources, and synchronize their activities. The Kernel employs mailboxes and semaphores to implement inter-task communication, mutual exclusion, and task synchronization. Mailboxes are used primarily for communication and semaphores are used primarily for synchronization. Mailboxes and semaphores establish task queues where tasks wait to communicate or exchange data with other tasks.

Mailboxes are used by tasks to exchange messages. A task that wants to send information to another task sends a message to a mailbox. Messages accumulate in a mailbox message queue until they are retrieved by a task, or until the queue is full. When a task wants to receive a message, the task waits at the mailbox. If there are no messages, the task may sleep (be blocked) in the task queue.

Semaphores are used to synchronize activities or protect resources shared by tasks. Semaphores use abstract "units" to perform synchronization or mutual exclusion. Tasks increment and decrement the semaphore's number of units. If a task requests a unit and one is available, the task proceeds. If no units are available, the task sleeps in the task queue.

Mailbox and Semaphore Task Queues

Mailboxes and semaphores contain queues where tasks wait to exchange messages or to synchronize with other tasks. When a requested operation cannot be immediately performed, the Kernel places tasks in these queues in one of the following orders:

- FIFO order
- Priority-based order

FIFO order places tasks in the task queue on a first-in-first-out basis. The first task placed in the queue is the first task serviced from the queue.

Priority-based order places tasks in the task queue with the highest priority task at the head of the queue. Tasks of equal priority are queued in FIFO order.

If a task is willing to wait at the mailbox or semaphore when its request cannot be filled, the task must specify the maximum time it will wait. The task then goes to sleep in the task queue until one of four events occurs:

- the task is at the head of the task queue and the request can be filled
- the specified number of clock ticks expires
- the task is deleted
- the mailbox or semaphore is deleted

Semaphores

A semaphore is an object at which tasks exchange abstract units either to synchronize execution or to provide mutual exclusion in their use of shared resources. In either case a task requests a unit from a semaphore. If the semaphore has a unit available, the task can proceed. If the semaphore has no units and the task is willing to wait, the task goes to sleep until a wakeup event occurs.

You can create three kinds of semaphores. The first two are general-purpose semaphores that can have up to 65,535 units:

- FIFO Semaphores: Tasks queue at these semaphores in First-In-First-Out order.
- Priority Semaphores: Tasks queue at these semaphores in priority order.
- Region Semaphores: These are a special type of semaphore with priority adjustment capabilities, making them useful for mutual exclusion in critical areas. Tasks queue at region semaphores according to priority.

Creating and Deleting Semaphores

Semaphores are created with the **create_semaphore** system call and deleted with the **delete_semaphore** system call. When you create a semaphore, specify the following:

- The memory area for the semaphore object
- The kind of semaphore (priority-based or FIFO task queue, or region semaphore)
- The initial number of units in the semaphore (zero or one)

To provide additional units to a semaphore after creation, use the **send_unit** system call once for each additional unit desired. Region semaphores cannot accept more than one unit. A region semaphore created with zero units is owned by the creating task, and cannot be used by other tasks until the creating task sends a unit.

If a semaphore is deleted, all tasks in the semaphore's task queue are awakened with a status message indicating that the semaphore was deleted while the tasks were waiting.

Sending and Receiving Semaphore Units

A task requests a unit (or units) from a semaphore with the **receive_unit** system call. Typically, the purpose is to gain access to a resource. If a unit is available, the semaphore decreases its unit count by one and the task proceeds.

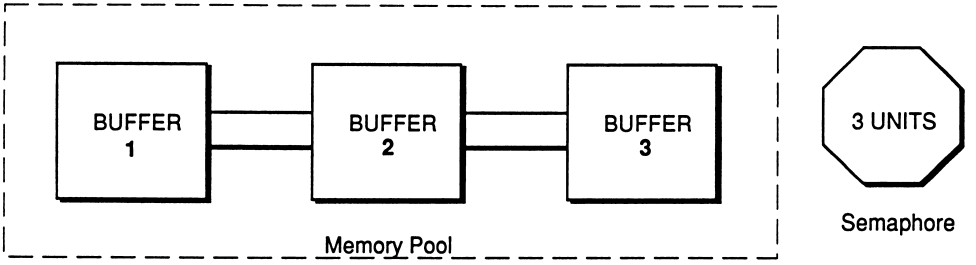
A task sends a unit (or units) to a semaphore with the **send_unit** system call. Typically, the purpose is to release a previously-obtained resource. If tasks are waiting at the semaphore, the task at the head of the queue is awakened to receive the newly-available unit.

Using FIFO and Priority Semaphores

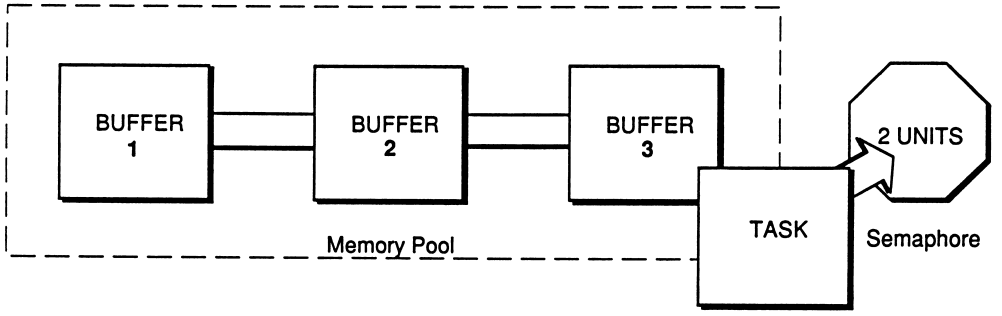
One use of semaphores is to synchronize activities between tasks. For example, assume that task A has obtained a unit from a single unit semaphore, and task B is waiting at the semaphore for the unit to become available. When task A is ready to synchronize with task B, it returns the unit. Task B obtains the unit from the semaphore and, assuming it is higher priority, begins execution.

Tasks also use semaphores to ensure that only one task has access to a resource at any time. This is called mutual exclusion. Semaphores used for mutual exclusion can have one or more units. (A region semaphore, discussed later, has only one unit.) If the resource the semaphore is guarding is available, a unit is available. If the resource is being used by a task, the semaphore will not have a unit. Using semaphores to control access to resources such as buffers avoids deadlock or other undefined events that result when two tasks attempt to access the same resource simultaneously.

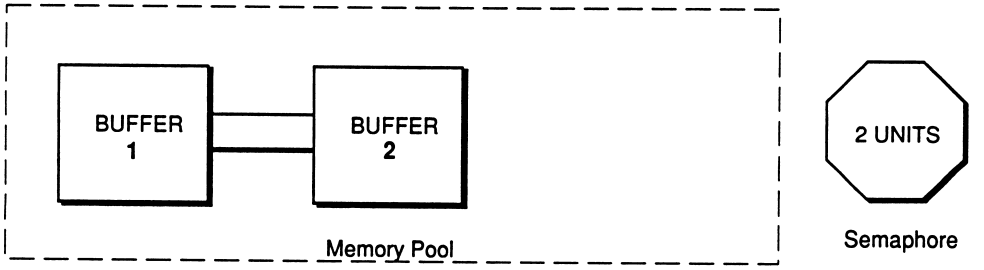
Figure 7-1, on the following page, illustrates an example of tasks using a semaphore to regulate use of a shared resource. In this case, the number of units in the semaphore correspond to the number of units available in the resource. The example shows a memory pool consisting of three buffers, guarded by a three-unit semaphore. A task that wants a buffer first requests a unit from the semaphore; then proceeds to use a buffer. If no units are available at the semaphore, no buffers are available. After releasing a buffer, a task sends a unit back to the semaphore. Another task waiting for a unit may then be awakened and serviced.



A memory pool has three user-created buffers available. The associated semaphore has three units.



A task that wants a buffer requests a unit from the semaphore. Units are available, so the semaphore decreases its units to two.



The task removes a buffer from the memory pool. After it returns the buffer, it will return a unit to the semaphore.

W-2581

Figure 7-1. An Example Using a Semaphore

Using Region Semaphores

A region semaphore can be used for mutual exclusion to protect a critical section of code or memory, or a critical hardware device. If a task must get a unit from a region before entering a critical section, and if it returns the unit when leaving the area, only one task will ever execute in the critical section at a time. The critical section is "owned" by the task with the unit, and guarded, in effect, by the region. Region semaphores may also be used for synchronization. Region semaphores contain a maximum of one unit and support priority adjustment.

The Benefits of Priority Adjustment

Problems may arise in certain situations when using non-region semaphores for mutual exclusion. Consider the case where a task with a low priority accesses a resource being guarded by a non-region semaphore. The low priority task is given exclusive use of the resource. If a task with a high priority attempts to access the same resource through the semaphore, and if the high priority task is willing to wait, it is put to sleep until the unit is available from the semaphore. Once the unit is available, it is passed to the high priority task and the task is permitted to access the resource.

However, suppose instead that a medium priority task becomes ready for execution while the high priority task is sleeping. It immediately gains control of the processor and pre-empts the low priority task that has access to the resource. The suspension of the low priority task's execution continues to tie up the resource and further delays the awakening and execution of the high priority task. In effect, the medium priority task has pre-empted the high priority task waiting at the semaphore.

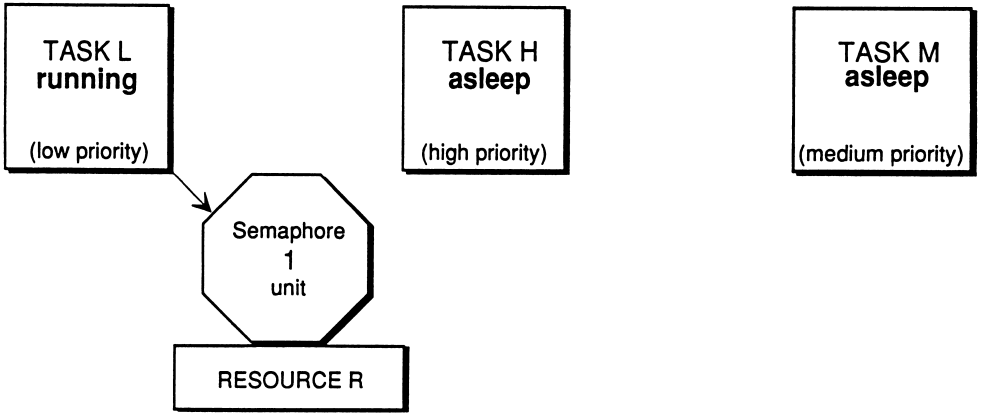
Figure 7-2 on page 7-8 illustrates this example, showing how the high priority task is delayed by using an ordinary semaphore.

Region semaphores provide priority adjustment to solve the problem of blocking a high priority task. Priority adjustment means that the region temporarily raises the priority of the task controlling the region to that of any higher priority task waiting for the region. The lower priority task's new priority remains in effect only while the task continues to hold the region. Thus, when a region semaphore is used, a medium priority task cannot pre-empt a high priority task waiting for the resource.

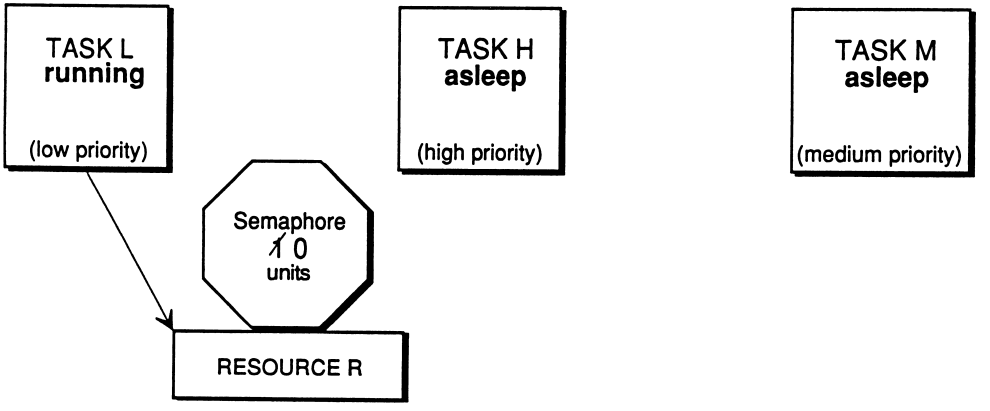
This temporary priority is called the task's dynamic priority. The task's static priority is not affected by this change. When a task gives up control of the region(s) it owns, the task's dynamic priority is readjusted to its static priority. The changing value of a task's dynamic priority has no effect on the static priority value returned by the **get_priority** system call.

Figure 7-3 on page 7-10 shows the effects of priority adjustment when using a region semaphore.

Figure 7-2 illustrates what happens when an ordinary single-unit semaphore is used for mutual exclusion



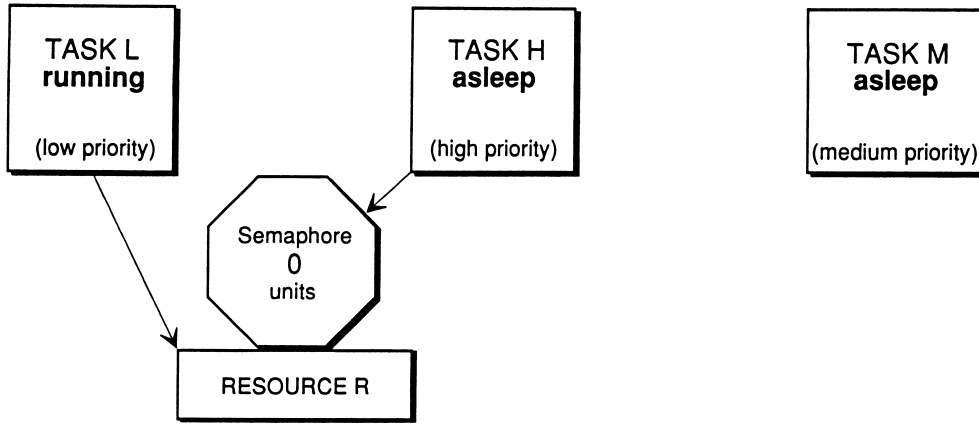
(a) Task L gets unit from the semaphore guarding access to Resource R



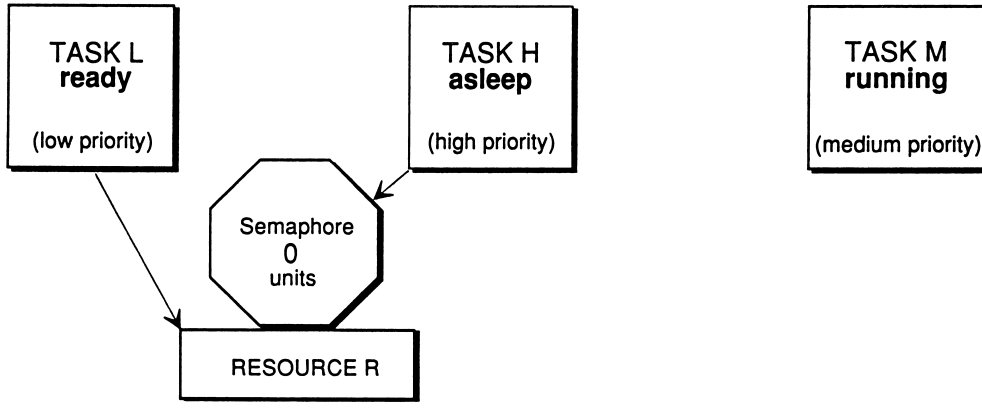
(b) Task L has exclusive access to Resource R

W-2582

Figure 7-2. Critical Resource Guarded by a Single Unit Semaphore (Part 1 of 2)



(c) Task H awakens as a result of an external event. Task H pre-empts Task L and attempts to gain access to Resource R, but the semaphore has no units. Task H goes back to sleep, waiting for the unit.

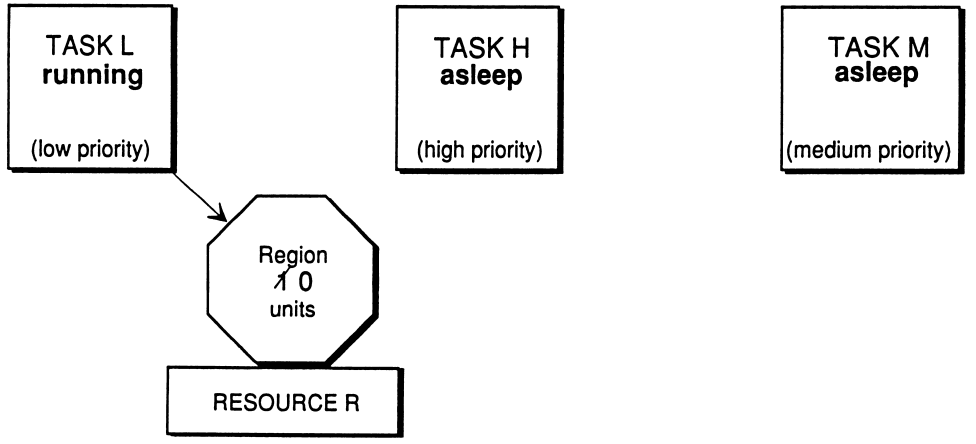


(d) Task M awakens as a result of an external event and pre-empts Task L. Note that a higher priority task, Task H, is effectively blocked by a lower priority task, Task M.

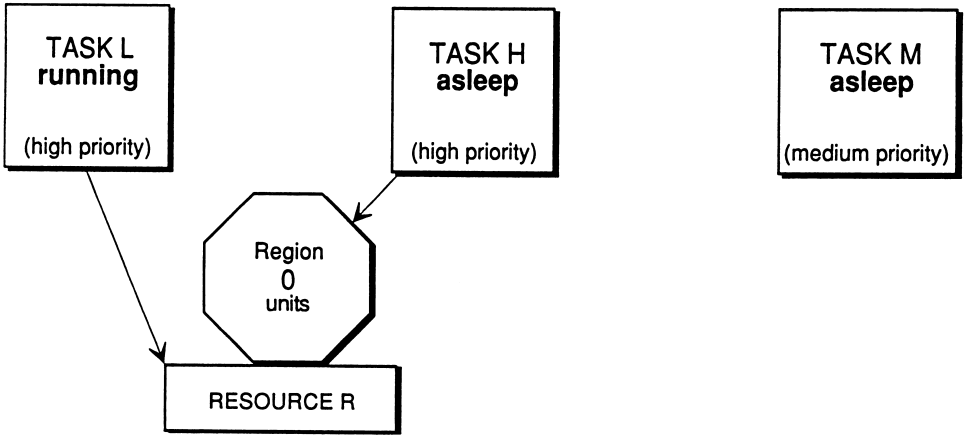
W-2583

Figure 7-2. Critical Resource Guarded by a Single Unit Semaphore (Part 2 of 2)

Figure 7-3 illustrates the same example as Figure 7-2, but using a region. The results are different because regions incorporate priority adjustment.



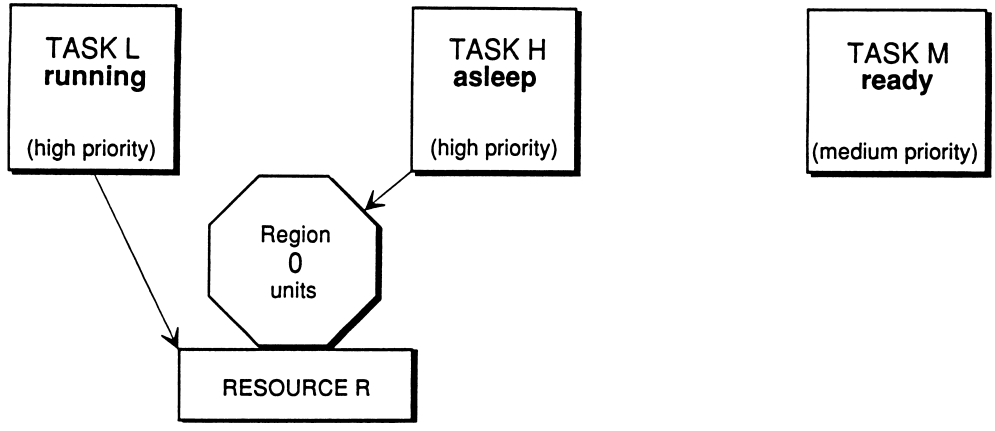
(a) Task L gets unit from the region guarding access to Resource R.



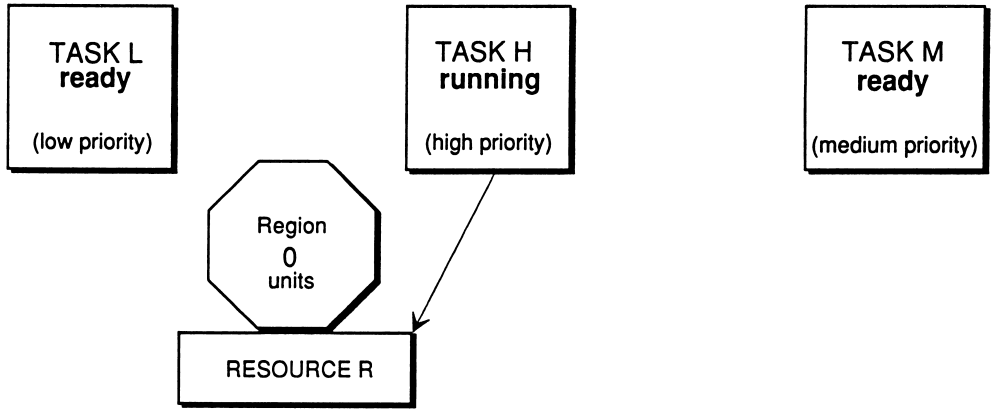
(b) Task H awakens as a result of an external event. Task H pre-empts Task L and attempts to gain exclusive access to Resource R, but there are no units available in the region, so L proceeds. The priority of L is raised to that of H while it holds Resource R.

W-2584

Figure 7-3. Critical Resource Guarded by a Region (Part 1 of 2)



(c) Task M awakens as a result of an external event. Task M makes a transition into the ready state because Task L is higher priority.



(d) When L releases Resource R, returning a unit to the region, L has its priority reduced to its original value. Task H is awakened, receiving the unit from the region. Task H now has control of Resource R.

W-2585

Figure 7-3. Critical Resource Guarded by a Region (Part 2 of 2))

NOTE

You must ensure that the task owning a region is always the task to return the unit to the region semaphore. Doing so prompts the Kernel to reset a task's dynamic priority back to its static priority.

Nesting Regions and Deadlock

A task may have control over more than one region semaphore at a time. This is called nesting regions, even though the individual regions controlled by the task are not otherwise connected.

Nesting regions can lead to deadlock, in which two or more tasks control resources needed by the other tasks. Deadlock can occur when tasks try to control more than one region simultaneously. Applications must keep track of nested regions and make sure to release regions in a last-obtained/first-released order. Each task must release the most recently obtained region first.

The following technique avoids deadlock in cases where nested regions are used:

- Apply a strict order to the regions in the system. For example, if the system uses 10 regions to guard shared resources, list the 10 regions on a sheet of paper and number them from 1 to 10.
- Have all tasks access the regions in numerical order. For instance, if a task needs to nest regions 3, 6, and 8, it must ask for control of these regions in numerical order and then give up control of the regions in the reverse numerical order. The order of regions is not important as long as all tasks follow the same numbering scheme.

When a task nests regions and its priority is adjusted, its original priority will not be restored until it has given up all the regions it controls.

Summary of Semaphore System Calls

The following is a list of the Kernel semaphore system calls:

- The **create_semaphore** system call creates a semaphore of the specified type with zero or one initial units. If a region is created with zero initial units, the creating task is made the owner of the region.
- The **delete_semaphore** system call deletes the specified semaphore. All tasks waiting at the semaphore are awakened and given the E_NONEXIST status code.
- The **receive_unit** system call requests a unit from the specified semaphore. If the semaphore contains units, the count of units is decremented by one and the task proceeds. If the semaphore has no units and the task is willing to wait, the task is put to sleep and placed into the semaphore's task queue.
- The **send_unit** system call adds a unit to the specified semaphore. If tasks are waiting at the semaphore, the task at the head of the queue is awakened.

Mailboxes

A mailbox is a Kernel object used for passing messages between tasks. Mailboxes are useful for both synchronization and communication. Mailboxes have task queues and message queues. When a task sends a message to a mailbox and there are tasks waiting in the queue, the message is copied to the message area of the task at the head of the queue. If no task is waiting, the message is stored in the message queue until a task requests a message from the mailbox. At that point, the message is removed from the message queue and given to the task.

If a task requests a message from the mailbox when there is no message, the task may wait in the task queue for a message. The Kernel puts the waiting task to sleep until one of the following occurs:

- A message arrives.
- The amount of time the task is willing to wait expires.
- The mailbox or the task is deleted.

An application can create two kinds of mailboxes, FIFO- or priority-queuing, which determine how waiting tasks queue at the mailbox. If tasks queue in FIFO order, the task that has waited the longest is awakened when a message arrives. If tasks queue according to their priority, the highest-priority waiting task is awakened when a message arrives. In priority-based queues, tasks of equal priority are queued in FIFO order.

Mailbox messages normally queue in FIFO order. If a mailbox contains more than one message when a task requests a message, the task receives the oldest message. However, high priority messages can be sent to a mailbox. The mailbox places them at the front of the message queue. If high priority messages are continually sent, causing the mailbox to put them at the head of the queue, the queue becomes LIFO (last-in-first-out).

When a mailbox is created, the application may reserve one of the slots in the mailbox message queue for a high priority message. This enables the mailbox to accommodate at least one high priority message even if the queue is full.

Figure 7-4 shows an example of using a mailbox, including the following steps:

1. An interrupt handler responds to an interrupt.
2. The handler fills a buffer with data.
3. When finished, the handler sends the buffer as a message to a mailbox where a task is waiting.
4. The waiting task is awakened and receives the buffer.

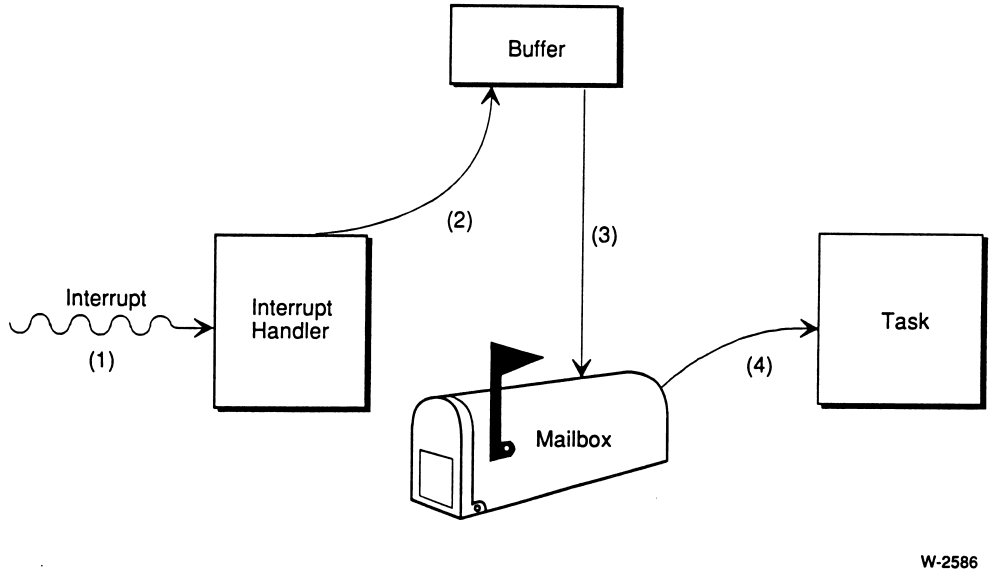


Figure 7-4. Example Task Interaction Using a Mailbox

Creating and Deleting Mailboxes

Mailboxes are created with the **create_mailbox** system call and deleted with the **delete_mailbox** system call. When you create a mailbox, specify the following:

- the memory area for the mailbox
- the message size for the mailbox
- the maximum number of messages the mailbox can hold
- whether the task queue will be priority-based or FIFO
- whether the mailbox will reserve a slot in its message queue for a high priority message

If a mailbox is deleted and there are tasks waiting for messages, all tasks are awakened with a status indicating that the mailbox no longer exists. Any messages queued at the mailbox are lost.

Sending and Receiving Mailbox Messages

A task sends ordinary messages to a mailbox with the **send_data** system call. High priority messages are sent with the **send_priority_data** system call. The task specifies the actual message size, which must be less than or equal to the maximum message size for the mailbox. The maximum message size is the size specified when creating the mailbox. The **send_data** and **send_priority_data** system calls return a status value indicating either that the message was accepted or the mailbox was full.

A task receives data from a mailbox with the **receive_data** system call. The task must provide a message area equal to the maximum message size of the mailbox. The call returns the actual size of the received message and a status value indicating one of the following:

- The task received a message.
- The time limit expired while the task was waiting.
- The mailbox was deleted while the task was waiting.

Handling Mailbox Overflow

If a mailbox contains its maximum number of messages when a message is sent, the Kernel returns an exception stating that the mailbox limit was exceeded. Thus, a mailbox enforces flow control by rejecting messages when the queue is full.

Depending on the application, there are several ways to handle mailbox overflow:

- Design the application so mailboxes never overflow.
- Consider mailbox overflow a fatal system error.
- Abort the activity causing the overflow.
- Send the message again, if it is known that a task received a message, creating room in the message queue.

If a slot is reserved for a high priority message, mailbox overflow may be indicated when sending an ordinary message, even though the mailbox can still accept a high priority message.

Summary of Mailbox System Calls

The following is a list of the mailbox system calls:

- The **create_mailbox** system call creates a mailbox in the specified area. The calling task specifies the maximum size of messages, the maximum number of messages that may be queued, and whether the task queue is FIFO or priority-based.
- The **delete_mailbox** system call deletes the specified mailbox. All tasks waiting at the mailbox are awakened and given an E_NONEXIST status, and all messages queued at the mailbox are lost.
- The **receive_data** system call requests a message from the specified mailbox. If the mailbox contains at least one message, the message at the head of the queue is returned to the caller. This is either the oldest message or the latest high-priority message. If no messages are available and the task is willing to wait, the task is put to sleep in the task queue.
- The **send_data** system call sends a message to the specified mailbox. If a task is waiting at the mailbox, it receives the message. Otherwise, the message is queued. If the mailbox is full, an exception is returned.
- The **send_priority_data** system call sends a high priority message to the specified mailbox. If a task is waiting at the mailbox, it receives the message. Otherwise, the message is placed at the head of the queue. If the mailbox is full, an exception is returned.

Memory Management

The Memory Manager defines and implements memory pools, providing Kernel applications that do not use paging with a physical memory management facility. If you choose to implement paging, you must also provide your own memory management facilities.

A memory pool is a storehouse of memory that tasks can share for dynamic memory allocation. Memory pools allow tasks to minimize their memory usage. Tasks "check out" memory areas from the memory pool, use the memory, and return the memory when finished using it.

A memory pool spans a contiguous range of memory. In addition, it contains a set of data structures for keeping track of which areas in the pool are currently in use and which are available. The Memory Manager allocates and deallocates memory areas in response to application requests.

The Memory Manager does not protect memory areas from unauthorized access. Any task could ignore the "rules" and access memory given to another task, sometimes with disastrous results.

Creating Memory Pools and Areas

An application uses the **create_pool** system call to create a memory pool in a specific range of memory. The calling task specifies where in memory to create the memory pool object and the size of the pool, including overhead. You may determine the memory range in a variety of ways, for example, from configuration parameters or from the application's data segment.

To use the memory in a memory pool, the application invokes the **create_area** system call. The calling task specifies the memory pool's token and the size of the requested area, including area overhead. If the requested space is available in the pool, the Memory Manager returns a pointer to the area. This pointer can be used to access the area, to create a segment descriptor to the area, or to create a memory sub-pool from the area. If the request cannot be filled, **create_area** returns a null pointer.

NOTE

If a memory pool is created on a four-byte boundary, all areas created from that pool will be on a four-byte boundary. To align the memory, the pool can be the start of a Builder-defined segment or a large array of integers defined statically in your application.

Deleting Memory Pools and Areas

When the application is through using an area, it calls **delete_area**, specifying the area to be released and the pool from which the area came. The **delete_area** system call returns the memory to the memory pool, making it available for reuse.

When an application no longer needs a memory pool, it calls the **delete_pool** system call. The **delete_pool** call does not require all of the areas to be returned in order to delete a pool. However, if an area is still in use when the pool is deleted, there is a chance that the same memory could be used simultaneously for two purposes, with undefined results.

NOTE

When using memory pools, applications should not access memory within the pool except for areas allocated by the **create_area** system call. Applications should not invoke memory pool system calls upon a memory pool after invoking the **delete_pool** system call on it.

Pool and Area Overhead

A memory pool occupies exactly the size specified when it is created. There is a minimum size that can be requested, represented by the literal KN_MINIMUM_POOL_SIZE. This size is the minimum number of bytes that the Kernel requires for a memory pool. It includes overhead data structures whose memory cannot be allocated from the pool. The usable space for a pool is actually the requested size minus the pool overhead. The literal KN_POOL_OVERHEAD defines the number of bytes in the overhead. To create a pool of size X, the total number of bytes required would be X + KN_POOL_OVERHEAD.

The literal KN_MINIMUM_AREA_SIZE designates the smallest area that can be allocated from a memory pool. If an application requests an area smaller than the minimum size, the memory manager rounds the requested size up to the minimum size. There is also an overhead associated with each area created from a memory pool. The literal KN_AREA_OVERHEAD defines this amount. Thus, if an area of size X is desired, X + KN_AREA_OVERHEAD bytes are required.

Performance Issues

The application gains the highest level of performance from a memory pool if all allocated memory areas are of the same size. In addition to minimizing wasted space, the times to allocate and deallocate fixed-size areas are less.

Allocating memory areas on four-byte boundaries enables Kernel system calls to execute faster because the objects created in the areas are also aligned on four-byte boundaries. Memory pool properties provide that, if a memory pool is created aligned on a four-byte boundary, all areas allocated from within that pool are also aligned on four-byte boundaries.

To create a pool that can allocate exactly n areas all of size m, the area required is:

```
n * (m + KN_AREA_OVERHEAD) + KN_POOL_OVERHEAD
```

If m is less than KN_MINIMUM_AREA_SIZE, replace m with KN_MINIMUM_AREA_SIZE in the expression.

Figure 7-5 illustrates the relationship between a memory pool and three memory areas. Although areas may be different sizes, as shown in this figure, access to the areas is more efficient if all areas in a pool are the same size.

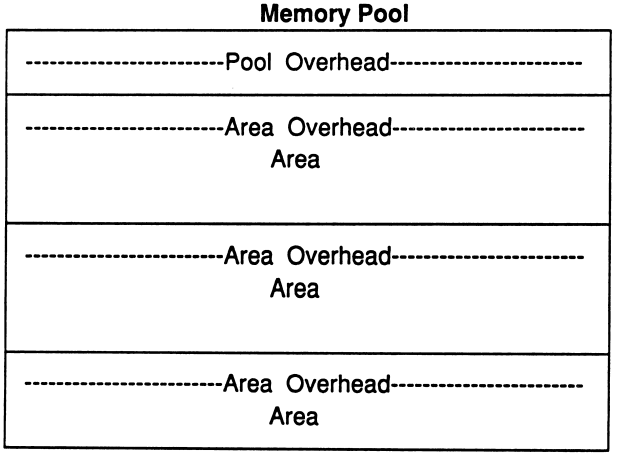


Figure 7-5. Memory Pools and Areas

Getting Information about a Pool

Using the **get_pool_attributes** system call, a task can get the following information about a specific memory pool:

- the size of the pool
- the total available space in the pool
- the largest contiguous available area in the pool

Allocating Memory in an Interrupt Handler

The **create_area** and **delete_area** system calls use an internal semaphore for mutual exclusion and may cause tasks to go to sleep. Interrupt handlers may safely use these system calls on a pool only if all operations on the memory pool (by either the interrupt handler or any other procedure) are performed with interrupts disabled. This ensures that the memory pool will always be accessible when the interrupt handler invokes a system call on it.

Summary of Memory Management System Calls

The following is a list of the Memory Management system calls:

- The **create_area** system call allocates an area of memory of specified size from a specified memory pool. If there is insufficient memory in the pool to satisfy the request, a null pointer is returned.
- The **create_pool** system call creates a memory pool in a specified range of memory.
- The **delete_area** system call returns an area to the memory pool it was allocated from. The area becomes a part of the available space in the pool.
- The **delete_pool** system call deletes a memory pool. The pool may be deleted even if it has areas allocated from it. This system call makes the entire address range of the memory pool available for reuse.
- The **get_pool_attributes** system call returns the size of the pool, the total available space in the pool, and the largest contiguous available area in the pool.

Device Management

In general, real-time systems are required to respond to and control real-time devices such as robotic assembly arms, pressure values, and conveyor belts. Peripheral devices such as Programmable Interrupt Controllers (PICs) and Programmable Interval Timers (PITs), assist real-time systems in receiving, interpreting, and processing the interrupts or inputs from these real-time devices.

The Kernel provides optional modules that manage the 8259A PIC, the 82380/82370 Integrated System Peripheral, the 8254 PIT, the 387™, 387SX, and 486 numeric coprocessors, and the 82530 SCC device.

Programmable Interrupt Controller (PIC) Management

Proper management of PICs ensures that a PIC is informed when the system has finished processing its interrupt. In addition, if several interrupt sources exist in a system, specific sources may need to be masked while others are enabled. For example, if too many interrupts from one source are queued for processing, that source may need to be masked. If the running task is important enough that it should not be pre-empted by certain interrupts, then the sources for those interrupts should be masked.

To ensure device independence, the Kernel never directly manipulates the PICs in the system. The user is free to manipulate the PICs according to the needs of the application. The source code for the PIC Manager is provided with the Kernel (see Table 7-1 on page 2). Source code is supplied so you can modify the default manager or support other types of PICs.

Using the 8259A PIC

Each 8259A PIC provides eight input pins for external interrupts. The PIC has a single output pin that connects to the INTR pin of the processor or to the input of another PIC. When the PIC detects an interrupt on one of its input lines, it signals the processor. The processor then interrogates the PIC to determine the interrupt source.

With a single 8259A PIC, the master PIC, there are eight available interrupt sources. PICs can also be cascaded, with up to eight of the input pins on the master PIC connected to output pins of slave PICs. With a cascaded environment, there are 64 possible interrupt sources (8 slaves * 8 inputs/slave). An input line on the master PIC can be connected to either a slave PIC or an external interrupt source. Figure 7-6 shows the relationship between slave PICs, the master PIC, and the CPU.

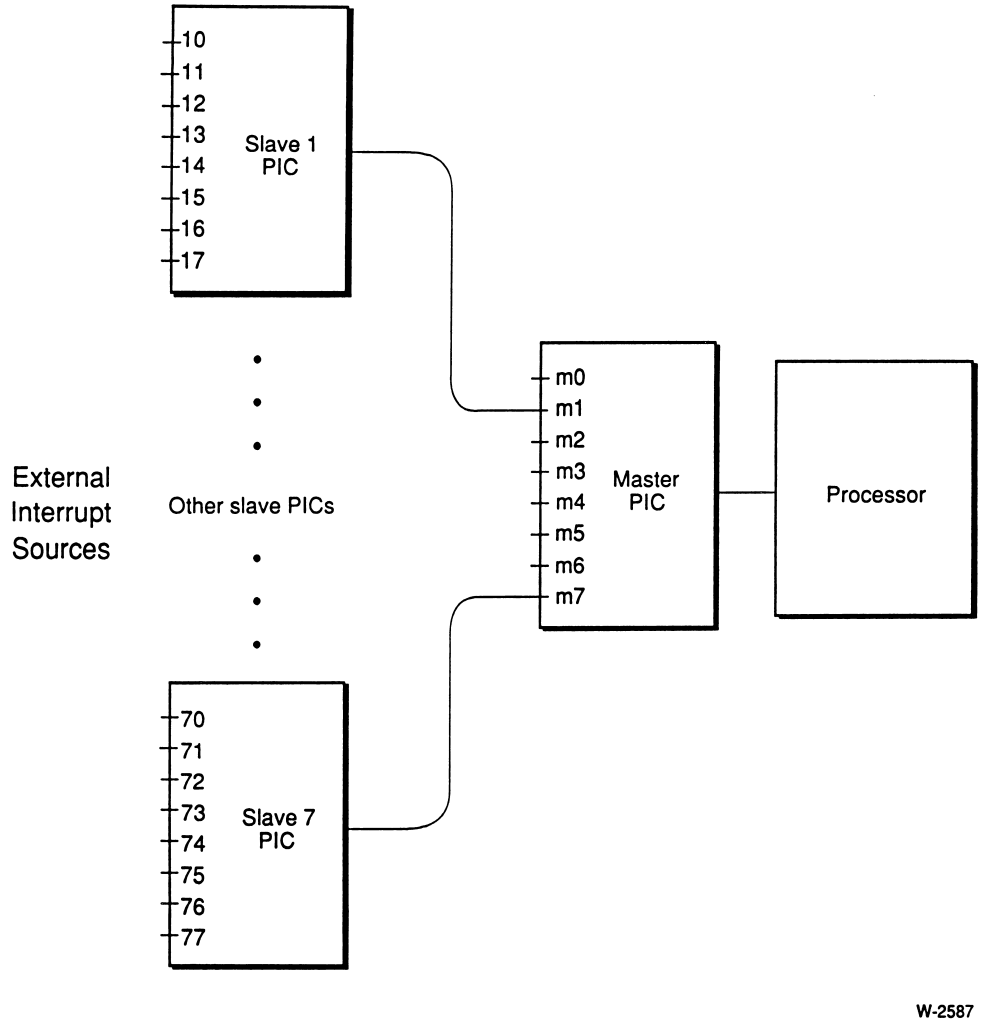


Figure 7-6. External Interrupt Sources

The pins on the master PIC are numbered m0 - m7, with Pin 0 having the highest priority. If the PIC detects more than one interrupt on its pins at the same time, the highest priority interrupt is reported to the processor. Pins on the slave PICs are numbered from x0 to x7, where x is equal to the master pin to which the slave is attached. The higher the slave pin number, the lower the interrupt priority. Master Pin 0 can be connected to a slave PIC only if all the master pins are connected to slaves.

The Kernel PIC Manager only supports the 8259A when programmed in special fully-nested mode (SFNM). The user can choose edge or level sensitivity for interrupt detection. However, if the optional PIT Manager module is used as the system clock, the PIC to which the PIT is connected must be set for edge-sensitive detection. Applications must always use an end-of-interrupt signal to restore the PIC after servicing an interrupt.

Using the 82380 or 82370 Device as a PIC

The 82380/82370 Integrated System Peripheral has three interrupt controllers. They are called Bank A, Bank B, and Bank C. Bank C is cascaded to Bank B, which is cascaded to Bank A. Unlike the 8259A, vectors for all interrupt levels can be individually programmed. In addition, Bank A provides an interrupt level 1.5, and the handler attached to that level will be invoked whenever the Initialization Command Word 2 (ICW2) of any of the banks is modified. (ICW2 provides compatibility with code written for the 8259A.)

The Kernel supports only Bank A and Bank B interrupt controllers. Bank A is supported as a PIC master, and Bank B is supported as a slave to Bank A. The Kernel does not currently support individual vector assignments. The user must supply the first slot for each device, and the Kernel programs the vectors in ascending order starting with first slot as the first vector. The Kernel provides a handler for level 1.5, and it should be set before the PIC is initialized (use the **level_M15_handler**, described under level_x7 handlers in the *iRMK™ Kernel Reference Manual*).

Because the 82380/82370 is highly integrated, some of its interrupt request lines are internally tied to timer and DMA functions. Bank B has five externally available interrupt pins.

The 82380/823870 has a single output pin that connects to the INTR pin of the processor. When the 82380/82370 detects an interrupt on one of its input lines, it signals the processor. The processor then interrogates the 82380/82370 to determine the interrupt source.

The Kernel 82380/82370 Manager only supports programming the 82380/82370 in special fully-nested mode (SFNM). The user can choose edge or level sensitivity for interrupt detection. However, if the optional 82380/82370 PIT Manager module is used as the system clock, the PIC to which the PIT is connected must be set for edge-sensitive detection. Applications should always use an end-of-interrupt signal to restore the PIC after servicing an interrupt.

See also: Appendix D, *iRMK™ Kernel Reference Manual*

Initializing PICs

The **initialize_PICs** system call configures the 8259A or the 82380/83370 PICs in the system. It also initializes the PIC(s) so that interrupt processing can begin. Before making this call you set up a configuration structure that specifies which PIC device you use, along with device-specific information. The support module for the chosen device handles the low-level programming details involved with managing the PIC.

See also: *Peripherals Handbook*

Interrupt Slots and Interrupt Sources

An interrupt slot is the IDT entry associated with a given interrupt source. Whenever an interrupt occurs, the gate at that entry is called to service the interrupt. The PIC Manager uses interrupt slots to specify interrupt sources in its interfaces. However, the **initialize_PICs** system call specifies interrupt sources by the pins on the master and slave PICs, rather than by slot. Using this method, the PIC Manager conceals the actual configuration of the PICs and interrupt sources in the system. Thus, changes to the PIC configuration cause minimal changes to the application. Further, this makes the PIC Manager interfaces generic, in the sense that they can be implemented for other types of PICs with different configurations.

Ordering Interrupt Sources

The physical and programmatic configuration of PICs determines the ordering of interrupt sources. This ordering establishes the priority of the interrupt sources, that is, it determines which interrupt will be presented to the processor should multiple interrupts occur simultaneously. Interrupt sources are mapped to the correct interrupt slots when the PIC manager is initialized. The interrupt slots should be configured in numeric order so that more important interrupt sources are given lower interrupt slot numbers.

See also: Configuration, *iRMK™ Kernel Reference Manual*

Masking Interrupt Sources

When an interrupt occurs, the processor automatically disables all hardware interrupts. However, interrupts can also be disabled individually at the PIC.

There are PIC management system calls that can mask and unmask specific interrupt sources. The **mask_slot** system call causes the PIC(s) to mask out any interrupts on the specified slot. The **unmask_slot** system call overrides **mask_slot**, causing the PIC(s) to unmask interrupts on the specified slot.

The **new_masks** system call changes the masking of interrupt sources relative to a specific slot. All slots less important than the specified slot are masked, while the specified slot and all slots of higher priority are unmasked. Specific slots may be masked using **mask_slot** after **new_masks** has been called. However, **unmask_slot** does not override the effects of **new_masks**. Further, **new_masks** does not override the effects of **mask_slot**.

Figure 7-7 illustrates the interaction between the masking system calls. In effect, there are two independent sets of interrupt masks, one controlled by the **mask_slot** and **unmask_slot** system calls, and the other controlled by **new_masks**. A given interrupt slot is enabled if and only if it is enabled in both masks.

For some applications it is necessary to change interrupt masks according to the priority of the current running task. A task switch handler and a priority change handler may be provided which invoke the **new_masks** system call according to the current running task's priority.

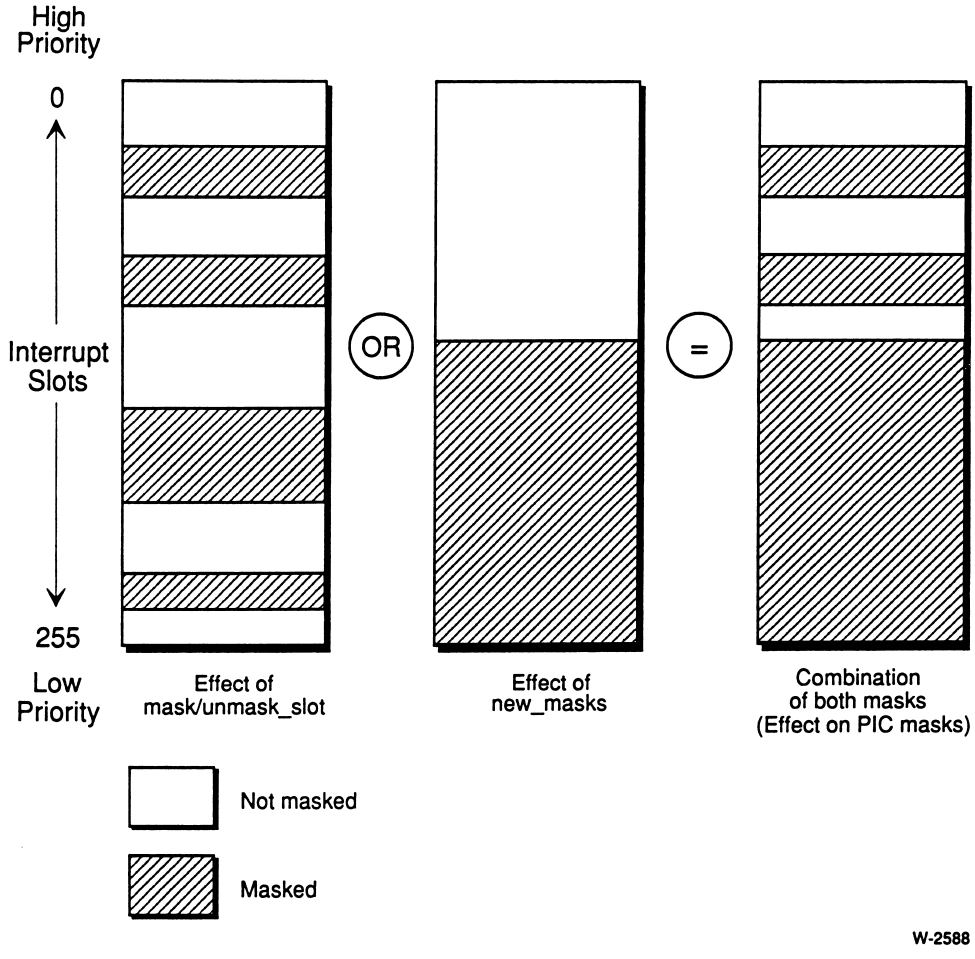


Figure 7-7. Effects of new_masks, mask_slot, and unmask_slot System Calls

Handling Spurious Interrupts

A spurious interrupt is a false signal received by the processor, often an electrical pulse that does not remain asserted long enough for the interrupt controller to respond to it. 8259A PICs or 82380/82370 Integrated System Peripherals report spurious interrupts on Line 7 of the PIC involved.

The PIC Manager provides default interrupt handlers, called level_x7 handlers, to respond to spurious interrupts. The Kernel-supplied level_x7 handlers assume true interrupt sources are not connected to Line 7 inputs. If an application does not connect Line 7 inputs to real interrupt sources, the level_x7 handlers can be used to handle spurious interrupts. Otherwise, the user must supply interrupt handlers that determine if a signal is a true or a spurious interrupt. To accomplish this, the user interrupt handler invokes the **get_slot** system call, which returns the highest priority interrupt currently in service. If **get_slot** returns the expected slot, the interrupt is real; otherwise, the interrupt is spurious.

See also: Kernel Handlers, *iRMK™ Kernel Reference Manual*

Sending End of Interrupt

The PIC Manager must give an end-of-interrupt indication to the PIC which caused the interrupt when interrupt processing is complete. The **send_EOI** system call indicates that the interrupt on the specified slot has been processed and that another interrupt on the same slot can be serviced.

Summary of PIC Manager System Calls

The following is a list of the PIC system calls provided by the Kernel:

- The **get_slot** system call returns the most important interrupt slot in service.
- The **initialize_PICs** system call configures the 8259A or the 82380/83370 PICs in the system. It also initializes the PIC(s) so that interrupt processing can begin.
- The **mask_slot** system call prompts the PIC to mask any interrupts on the specified slot until a corresponding **unmask_slot** system call is invoked on the slot.
- The **new_masks** system call changes the masking of the interrupt sources based upon a specified slot. All slots of lower priority than the specified slot are masked. If a slot is currently masked due to the effects of the **new_mask** system call, then the slot remains masked until it is unmasked by **new_masks**.
- The **send_EOI** system call informs the PIC that the interrupt on the specified slot has been processed and another interrupt on the same slot can be serviced.
- The **unmask_slot** system call unmasks interrupts on the specified slot that were masked by the **mask_slot** system call. **Unmask_slot** has no effect on slots that are masked by the **new_masks** system call.

Programmable Interval Timer (PIT) Management

The Kernel requires an external source of timing signals to implement its time management facilities. Rather than requiring a particular peripheral device, the Kernel depends upon the application to provide these timing signals.

Frequently, applications use a Programmable Interval Timer (PIT) to generate timing signals. If an 8254 PIT or an 82370/82380 Integrated System Peripheral device is used for this purpose, the Kernel provides an optional module to relieve the user of the low-level programming details involved with managing the PIT. In addition, the PIT Manager frees the application from having to call the **tick** system call, which generates software timing interrupts for the Kernel.

The PIT Manager programs the 8254 or the 82380/82370, accepts the interrupts from the chip, and invokes the **tick** system call.

The source code for the PIT Manager is provided with the Kernel (see Table 7-1 on page 2). This source code may be used to determine how the Kernel implements the PIT management support, or may be modified to support other devices.

Using the PIT Manager

The **initialize_PIT** system call is the first PIT Manager call the application makes. Either the 8254 or the 82380/82370 PIT Manager is chosen in this call. The following information must also be supplied:

- the frequency of the clock driving the PIT
- the interrupt slot assigned to the PIT
- the I/O addresses used to access the PIT
- the number of the selected timer on the PIT

The 8254 device has three timers. The timer port separation value is 2. The default system timer for the 8254 is timer 0.

The 82370/82380 Integrated System Peripheral device has four timers; the Kernel supports all four timers. The timer port separation value is 1. The default system timer is timer 3.

The **initialize_PIT** system call initializes the PIT Manager module but does not start the PIT counting. The **start_PIT** system call starts the PIT counting, using a time interval specified in the call.

The **get_PIT_interval** system call returns the value of the time interval currently in use and may be used to translate clock ticks into absolute time values.

Summary of PIT Manager System Calls

The following lists the PIT Manager system calls:

- The **get_PIT_interval** system call returns the value of the time interval generated by the PIT. This value can be used to translate clock ticks into absolute time values.
- The **initialize_PIT** system call initializes the 8254 timer or a timer on the 82380/82370 device. The initialization specifies the timer configuration but does not actually start the PIT.
- The **start_PIT** system call starts the PIT counting for the specified interval. The PIT must be initialized before the **start_PIT** system call is invoked.
- The **tick** system call sends a signal to the Kernel to increment its internal timer used for all time management facilities. The application only needs to make this call if a PIT Manager module is not used.

Coprocessor Management

A numeric coprocessor provides floating point operations for your application. The Kernel does not require a coprocessor in the system but can support coprocessor use by an application. The Kernel's Numeric Coprocessor Manager handles floating point operations for a 387 or 387SX coprocessor, or for the coprocessor built into the 486 processor.

The Numeric Coprocessor Manager frees the application from many of the low-level programming details of managing the numeric coprocessor. Coprocessor states must be multiplexed among the tasks in the system, just like the state of the central processor. Each task using a coprocessor must have its own copy of the coprocessor registers. When a task uses the coprocessor, the Kernel loads the coprocessor register values for that task from the coprocessor state information stored with the task state.

Using the Numeric Coprocessor Manager

The Numeric Coprocessor Manager saves and restores the coprocessor state only when a task accesses the coprocessor and a different task's state is already loaded. It does this by determining whether the context in the coprocessor's registers is the context of the task accessing the coprocessor. The Numeric Coprocessor Manager uses slot 7 in the IDT.

To use the Numeric Coprocessor Manager, invoke the **initialize_NDP** system call before any code that accesses the coprocessor. If the call to **initialize_NDP** is present in your code, the coprocessor manager is automatically included during the binding process.

When creating tasks that use the coprocessor, you must reserve an area to hold the coprocessor state and initialize that area with appropriate values. The coprocessor manager can automatically initialize the save area for each task created after you call **initialize_NDP**, by using the default task initialization handler. You choose whether to use this default handler with a flag set in the **initialize_NDP** call.

Manually Setting Coprocessor Save Areas

If you use the Numeric Coprocessor Manager, but do not choose to use the default task initialization handler, you must initialize the coprocessor registers save area (at the end of the task state) for each task that uses the coprocessor. To determine these initialization values, perform the following steps:

1. Disable interrupts.
2. Call **initialize_NDP**.
3. Execute an FSAVE assembly language instruction.
4. Set the task switch flag in register CR0.
5. Enable interrupts.

The location specified in the FSAVE operation will contain the values you must use to initialize the coprocessor save areas in tasks' task state areas.

Numeric Coprocessor System Call

The **initialize_NDP** system call initializes the Kernel's optional Numeric Coprocessor module.

Serial Communication Controller Support

Character I/O communication between a terminal and the application is provided by the Kernel through an optional 82530 Serial Communication Controller (SCC) Manager module. This manager provides a standard interface which supports low-level access to character I/O and enables the Kernel standard I/O library functions. The SCC Manager initializes and manages an 82530 device. This device might be on an iSBX 354 module attached to the base computer board or might exist on the base board itself. Source code for the manager (see Table 7-1 on page 2) allows you to adapt the manager for other serial devices.

To use the SCC Manager, first invoke the **initialize_console** system call before starting any character I/O. In this call you specify configuration values for the device (which can vary from board to board). Then use the **ci** or **csts** system call to read ASCII characters from the console input device. Use the **co** system call to transfer ASCII characters to the console output device.

Summary of 82530 SCC System Calls

The 82530 system calls are as follows:

- The **ci** system call reads an ASCII character from the console input device. This system call waits for console input if a character is not immediately available.
- The **co** system call transfers an ASCII character to the console output device. Before this system call transfers the character, it checks to see if a CONTROL-S was entered at the console. If so, **co** waits for the operator to enter CONTROL-Q before transmitting the character.
- The **csts** system call reads an ASCII character from the console input device. It is similar to the **ci** system call but does not wait if a character is not immediately available. If no console input character is available, it returns an ASCII null character (value 0).
- The **initialize_console** system call initializes the 82530 SCC device. Invoke this system call before starting character I/O.

Kernel kstdio Library Calls

The SCC character calls described above are invoked by the standard I/O functions provided in the *kstdioc.lib* and *kstdios.lib* libraries: **printf, scanf, getchar,** and **putchar**. To use the *kstdio* functions, you must first call **initialize_console**, and then call **initialize_stdio**.

See also: Chapter 8

Descriptor Table Management

Descriptor tables are part of the address translation tables supported by Intel386 family processors. The descriptor tables define all of the segments used in the system and implement the gate mechanism for controlling procedure calls and task management. The Kernel Descriptor Table Manager manipulates the descriptor tables. (The other address translation tables are page tables, which can be implemented, but do not have Kernel support.)

Every application that uses the Kernel runs in protected mode. Therefore, access to code and data is controlled by descriptor tables. There are three types of descriptor tables: Global, Local, and Interrupt (GDT, LDT, and IDT respectively). These three types of tables serve to translate virtual addresses into linear addresses, act as gate mechanisms for controlling procedure calls (including interrupt handling), and serve as a protection mechanism for isolating a task's code and data.

The Global Descriptor Table (GDT) contains:

- descriptors for code and data segments
- gates accessible to all tasks in the system
- descriptors for all descriptor tables in the system (the Kernel expects a descriptor for the GDT in slot 1 of the GDT)

Local Descriptor Tables (LDTs) are optional with the Kernel. If used, an LDT contains all the descriptors private to the tasks that share the LDT. Hence, a task's logical address space is defined by segment descriptors in the GDT and its LDT.

The Interrupt Descriptor Table (IDT) contains descriptor gates pointing to the location of up to 256 interrupt handlers and tasks.

See also: *386™ DX Microprocessor Programmer's Reference Manual*

Figure 7-8 shows the relationships between virtual, linear, and physical addresses and the descriptor tables.

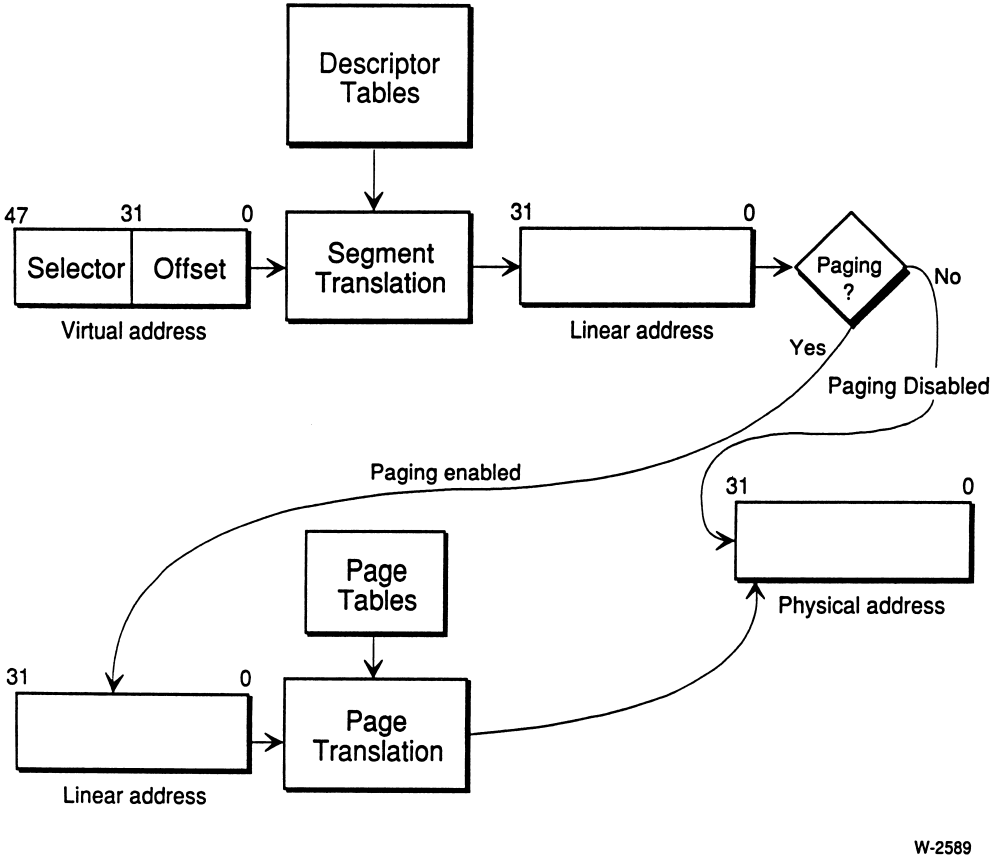


Figure 7-8. Addressing and Address Translation Tables

Building the Initial Descriptor Tables

The Builder utility (described in Chapter 4) constructs the GDT, the IDT, and any initial LDTs. When initially specifying the size of the GDT, include space for extra descriptors if you intend to allocate new descriptors at runtime.

By default, the Builder establishes a segment descriptor in slot 1 of the GDT, allowing the GDT to reference itself. Slot 2 has a descriptor to the IDT.

Descriptor Table Manager

The Kernel's Descriptor Table Manager allows the application to interrogate and change descriptors and descriptor fields, to initialize LDTs, and to convert between pointers and linear addresses.

There is no support for descriptor allocation in the Kernel. This support must come through the Builder or the application code. The application must keep track of which descriptor entries are in use and which are available.

Reading and Writing Descriptors

Selectors are 16-bit structures recognized by the processor, which specify a GDT or LDT and contain an index of a descriptor within the table. The Kernel requires every descriptor table in the system to have a descriptor in the GDT, including the GDT itself, which is referenced in slot 1. Thus a GDT selector can be used to specify any descriptor table in the system.

To read a descriptor, use the **get_descriptor_attributes** system call. To write a descriptor, use the **set_descriptor_attributes** system call. In either call, you specify a pair of selectors. One selector references a descriptor table and the other references a specific descriptor within the descriptor table.

When reading or writing a descriptor, you give a pointer to a structure where the attributes of the descriptor are specified (for writing to the descriptor table) or will be returned (for reading from the table). The first byte of the attributes structure determines whether the descriptor is a segment descriptor or a gate. All other information in the attribute's structure depends on this value.

Creating New LDTs

The Kernel supports creation of new LDTs at run-time. For an application to create an LDT, the application must supply memory for the table. Calculate the size of the memory required by multiplying the descriptor size (8 bytes) by the number of descriptors wanted. Then call the Kernel system call **initialize_LDT**, supplying a pointer to this memory.

Initialize_LDT initializes all the descriptors in the table to a null value, and initializes a descriptor in the GDT for the new LDT.

Returning a Descriptor to the Null State

The **null_descriptor** system call overwrites a specified descriptor with a null descriptor. The null descriptor indicates a one-byte, read-only segment with a DPL of 3, which is an unused location within the Kernel.

Converting Addresses and Pointers

There are two system calls that allow conversions between pointers and linear addresses. Given a linear address, the **linear_to_ptr** system call generates a pointer that references the linear address. Given a pointer, the **ptr_to_linear** system call generates a linear address equivalent to the pointer.

Translating Pointers

A pointer (a selector:offset pair) resolves to a particular location in memory. Given a new selector, the new offset that identifies the same memory location can be obtained using the **translate_ptr** system call.

Pointers in the Descriptor Table Manager System Calls

All pointers in the Descriptor Table Manager system calls are a full 48 bits, even those for the small model interface. Two system calls, **get_data_selector** and **get_code_selector**, allow small model programs to generate full pointers.

An Example of Creating a Segment

The following example describes the steps necessary for an application to create a segment.

1. While building the system, reserve the number of slots desired in the GDT, using the Builder utility. This ensures that those slots will be available for application use.
2. In the application, define an area that will be managed as free space. The segment will eventually be allocated from this area.
3. Call **create_pool** to create a memory pool from the memory space.
4. Call **create_area** to get the needed memory for the segment.
5. Call **set_descriptor_attributes**, specifying:
   - the GDT as the table to be used
   - one of the previously-reserved descriptors
   - a segment attributes structure containing a pointer to the area as the segment base, and the size of the area as the segment size

Summary of Descriptor Table Management System Calls

The following are the Descriptor Table Manager system calls.

- The **get_descriptor_attributes** system call returns a structure containing the attributes of a specified descriptor.
- The **initialize_LDT** system call sets up a specified descriptor in the GDT as a descriptor for an LDT. The previous contents of the GDT descriptor are overwritten. The call initializes all descriptors in the new LDT with null descriptors.
- The **linear_to_ptr** system call generates a pointer that refers to the same location as a given linear address. This system call is handy for referencing memory or for passing configuration parameters to Kernel interfaces.
- The **null_descriptor** system call overwrites a specified descriptor with a null descriptor. The null descriptor indicates a one-byte, read-only segment with a DPL of 3, which is an unused location within the Kernel.
- The **ptr_to_linear** system call returns the linear address referred to by a given pointer. The caller specifies the descriptor table in which the pointer is valid.
- The **set_descriptor_attributes** system call sets the attribute values of a specified descriptor.
- The **translate_ptr** system call returns a new pointer based on a selector provided in the call.

The following system calls are only available in the small model interface library (*c_call.lib*):

- The **get_data_selector** system call returns a selector for the current data segment.
- The **get_code_selector** system call returns a selector for the current code segment.

Interconnect Space

Interconnect space is a collection of up to 512 registers on every board in a Multibus II system. Each register is one byte long. The registers contain information about the board: its manufacturer, model number, memory configuration, and other board-specific information. The first 32 interconnect registers have a format specified by Intel and are collectively known as the header record. The format of the rest of the interconnect registers is defined by the hardware specifications of the manufacturer. Interconnect registers are often grouped into records dedicated to specific purposes.

Interconnect space is used to automate the identification of boards at system start-up. Using interconnect space, system software can determine the resources at its disposal and load system utilities as necessary. Interconnect space for each board is referenced using the board's cardslot ID in the backplane. Software can access bytes in interconnect space by referencing the cardslot number of the board and the address of the desired byte.

See also: *Multibus II Interconnect Interfaces Specification*

Using Interconnect Space

The **initialize_interconnect** system call initializes the Kernel's Interconnect Space Manager. This call must be invoked before any access of interconnect space is attempted. The system call takes a pointer to a structure which specifies the configuration information needed for the interconnect space.

Two system calls are used to manipulate interconnect register values. The **get_interconnect** system call retrieves the value of the specified interconnect register. The **set_interconnect** system call sets the contents of the specified interconnect register.

Applications reference a board in interconnect space by card slot number (host ID number). Host ID 31 always refers to the local board, allowing access to the local interconnect registers without knowing what slot the board is installed in.

Each register in an interconnect space record must be accessed individually. The application is responsible for ensuring consistent access to multiple interconnect registers.

NOTE

Interconnect space access is primarily intended for initialization and error recovery situations. Using the interconnect registers during normal system operations may have a severe impact on system responsiveness.

The **local_host_ID** system call returns the ID of the local host (the ID of the host on which the application is executing). This system call can be invoked without severely impacting system responsiveness.

Summary of Interconnect Space System calls

The following is a list of the interconnect space system calls:

- The **initialize_interconnect** system call initializes the interconnect space interface module, providing the configuration information necessary for tasks to access interconnect space. This system call must be invoked after the **initialize** system call but before accessing interconnect space.
- The **get_interconnect** system call retrieves the value of the specified interconnect register.
- The **set_interconnect** system call sets the contents of the specified interconnect register to a specified value.
- The **local_host_ID** system call returns the host ID of the local host; that is, the host ID of the host on which the application is executing.

Message Passing

Message passing refers to communication across the Multibus II parallel system bus (PSB). Because the PSB is separate from the local busses of the individual hosts, messages can move across the PSB while other processing occurs on individual hosts. The low-level protocols are implemented in an 82389 Message Passing Coprocessor and a direct memory access (DMA) device. The MPC is a custom VLSI device that provides an intelligent, programmable interface to the PSB. The host processors are free to perform other processing tasks while the MPCs transmit and receive messages. For details about programming the MPC and DMA devices, refer to the hardware reference manual for the board(s) you use.

In the optional Message Passing module, the Kernel provides services to support communication between two or more boards, or between tasks on a single board. The message passing module requires two hardware components, the MPC and one of the following, used for DMA:

- an 82258 ADMA device
- an 82380 or 82370 Integrated System Peripheral device
- one of the Intel boards designed for burst mode transfer, including the iSBC 386/133 or 486/125 board, or a MIX board

NOTE

The Kernel-supplied 82380/82370 DMA manager only supports Multibus II message-passing (that is, the data must go through the MPC).

With the 82380/82370 Integrated System Peripheral, the Kernel only supports aligned DMA transfers and does not support data chaining.

Kernel applications can communicate with any application that uses the Multibus II Transport Protocol. Communication might be between different boards using the Kernel or between a Kernel application and the System V host.

In addition to supporting the Multibus II standard message passing protocol, this module also allows the application to provide custom protocol handlers for message passing.

The message passing module consists of two levels of service: a data link layer and a transport layer. The data link layer provides a basic level of message passing support. It manages the MPC and the DMA controller and provides a means of supporting multiple protocols. The transport layer provides more sophisticated services, supporting the Multibus II Transport Protocol. The transport protocol is implemented on top of the data link layer. Either use the transport protocol or use the data link layer and provide your own protocol.

The Multibus II transport protocol specifies host IDs to uniquely identify source and destination hosts. The Multibus II hardware message passing protocol requires identification of the source and destination hosts in a message field called the message ID. The Kernel internally translates host IDs to message IDs.

See also: *Multibus II Transport Protocol Specification and Designer's Guide*

NOTE

The Multibus II Transport Protocol is not a network transport protocol; it is specific to the bus.

Performance Considerations for Message Passing

The following information describes the message passing modes available when using Intel boards and DMA devices. Other boards may not support some or all of these data transfer modes.

Data Transfer Requirements

The Kernel supports three modes of data transfer: slow (two-cycle), fast (one-cycle, or flyby), and burst mode. Fast mode and burst mode require both the data address and the data length to be aligned. When using data chains, all chain elements and the chain block (and their lengths) must be aligned. Unaligned transfers are done in the slow mode.

Support for these modes depends upon the DMA devices or Intel boards being used. Burst mode transfer is only supported by the iSBC 386/133 and 486/125 boards and the iSBC 386/020 MIX board. To use burst mode with one of these boards, the Kernel message passing structure must be configured properly. Table 7-2 shows the requirements for various Intel boards and data transfer modes. As indicated in Table 7-2, messages must be located in a specific address range.

Table 7-2. Data Transfer Mode Requirements

Message Passing Mode		Boards	
		iSBC 386/116 iSBC 386/120 iSBC 386/258	iSBC 386/133 iSBC 486/125 MIX 386/020
Slow Mode (2 cycle)	Data Alignment Address Range	Unaligned 0 to 16 MB	Unaligned 0 to 16 MB
Fast Mode (1 cycle)	Data Alignment Address Range	4-byte (address and length) 0 to 16 MB	4-byte (address and length) 0 to 32 MB
Burst Mode*	Data Alignment Address Range	Not Supported	16-byte (address and length) 0 to 128 MB

* When configuring message passing (the **initialize_message_passing** system call), auxiliary_DMA_support must be set to true.

Solicited and Unsolicited Messages

There are two kinds of messages on Multibus II, unsolicited messages and solicited messages.

Unsolicited messages are short, fixed-length messages that can be transmitted without explicitly allocating buffer resources and without synchronization between sender and receiver. The application sends unsolicited messages to a specific host or broadcasts the same message across the bus to all hosts.

A solicited message is a longer message outside the data structure used by the Kernel for message passing (its length is limited by the range of memory specified for particular boards in Table 7-2). A solicited message requires the receiving host to allocate a buffer to complete the transfer. A buffer request message (which is a specific kind of unsolicited message) initiates a solicited message transmission. It requests that the receiving host allocate a buffer large enough to hold the incoming message data. The receiver must reply with a buffer grant to the sender before the actual solicited message is transmitted.

A buffer grant allows the solicited message data transfer to begin. It informs the sending host that a buffer has been allocated and the receiving host is ready to accept the message. At the end of the transfer, the MPC generates a solicited completion interrupt to both sender and receiver. A buffer reject message sent by the receiver causes the sender's MPC to generate a solicited completion message with an error status.

A solicited message transfer is accomplished as follows:

- The sender sends a buffer request message, indicating that a buffer of a certain size is desired for data transfer.
- On receiving the buffer request, the receiver sends a buffer grant to allow data transmission to take place.
- The sending and receiving MPCs (using pre-programmed DMA controllers), transfer the data to and from the local memory of the sending and receiving hosts.
- Both the sender and receiver are notified when the transmission is complete.

Figure 7-9 illustrates a solicited message transfer.

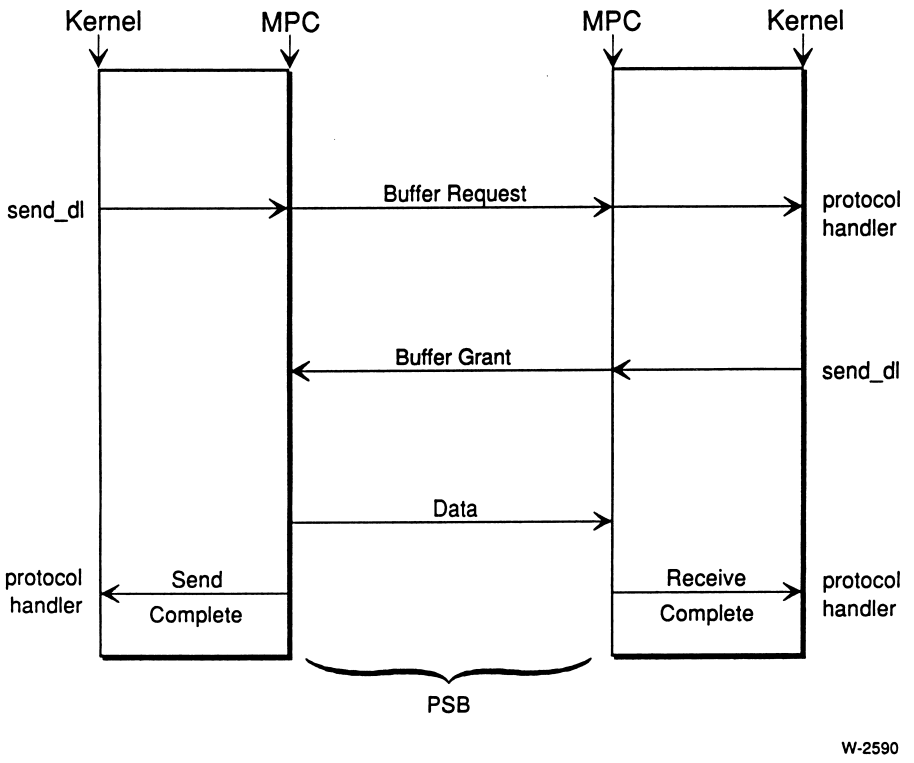


Figure 7-9. Solicited Message with Buffer Request and Buffer Grant

Sometimes the receiver will not have enough memory to receive the message. In this case, the following occurs:

- The sender sends a buffer request message indicating a desire to transfer data.
- The receiver sends a buffer reject rather than a buffer grant.
- The sender is notified of the reject as part of the solicited completion interrupt.

Figure 7-10 illustrates the buffer reject scenario.

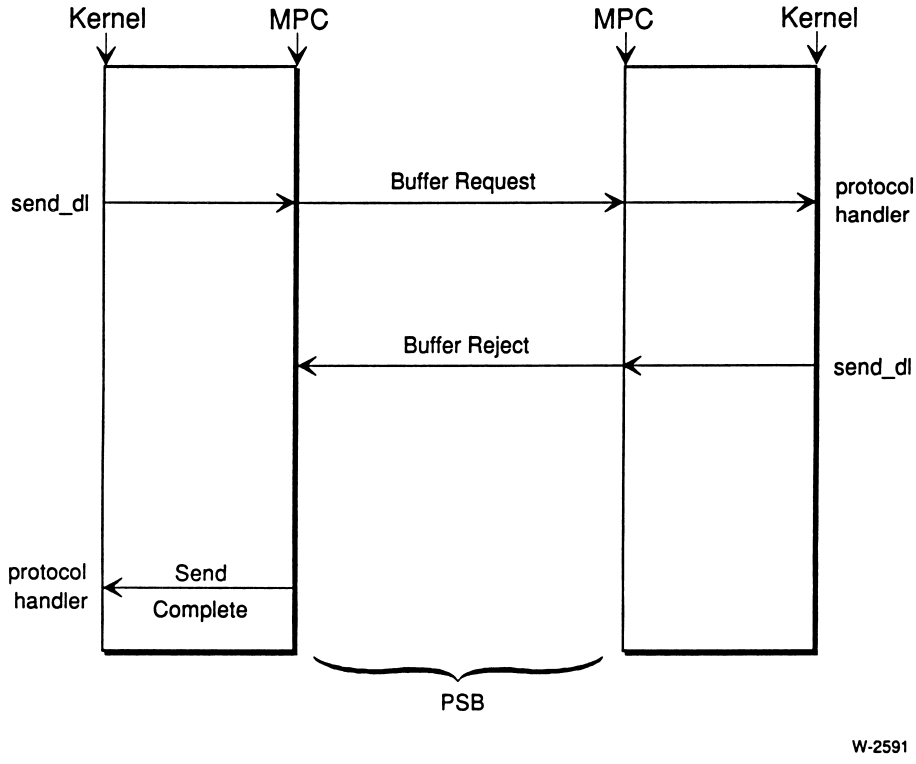


Figure 7-10. Solicited Message with Buffer Reject Response

Data Chaining

It may be impractical for an application to use one contiguous piece of memory to hold a message. If there is no contiguous memory available to send or receive a message, the application can build a chain of buffers called a data chain. A data chain is a list of noncontiguous buffers used to specify a single message. A structure called a data chain block, containing buffer pointers and byte counts for those buffers, specifies a data chain.

Figure 7-11 illustrates a data chain block and data chain. These conditions apply to data chains:

- The length of an individual data element of a chain cannot exceed 64K bytes.
- It is the application's responsibility to preserve the data chain block and chain elements until the message transmission is complete.
- Aligned data chains (addresses and lengths a multiple of 4 or 16 bytes) can take advantage of the fast or burst mode data transfer methods.
- The chain block and the chain elements should have the same scope: they must all be based on the GDT or on a single LDT.

The `data_type` field in either a data link or transport message specifies whether the data to be transmitted is in a contiguous block of memory or in a data chain. If the data is in a data chain, the `data_ptr` field of the message references the data chain block structure.

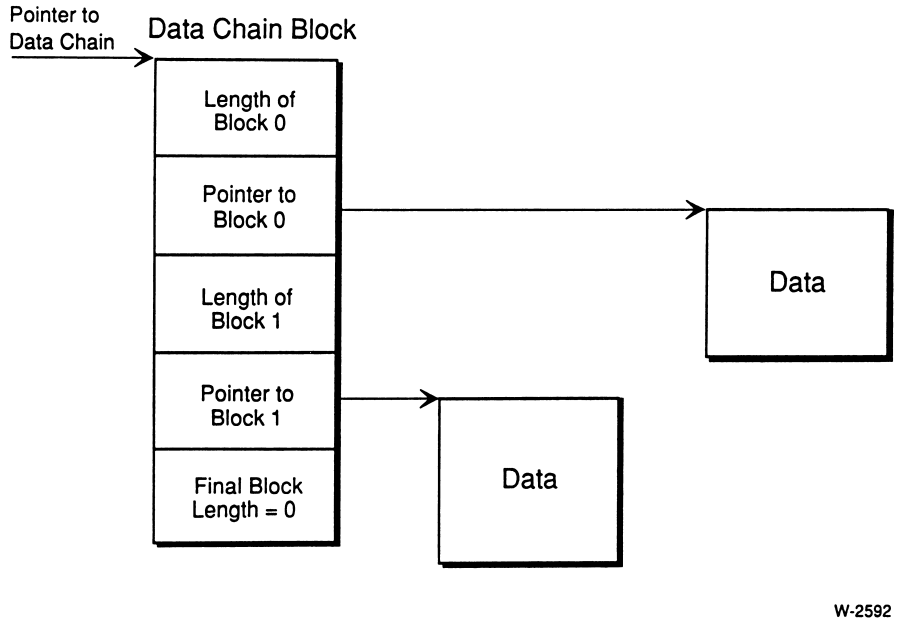


Figure 7-11. A Data Chain

Using the Data Link Layer

The data link layer provides low level support for managing the MPC and the ADMA device. This layer supports multiple protocols. The application provides a protocol handler for each protocol to be supported on the board. The application's protocol handler(s) are used instead of the Kernel's transport protocol handler.

Protocol IDs

Every Multibus II message contains a one-byte, software-defined protocol ID. This ID defines the protocol that must be used to interpret the message. Each independent user of the data link layer selects a protocol ID and an associated procedure for message passing on a given host. The data link layer routes the messages within the hosts according to that ID. When a message arrives at a host, the handler associated with the protocol ID is invoked. Each application protocol must have a unique protocol ID. All handlers in the system that handle a particular protocol use the same protocol ID. Protocol IDs have the following ranges:

Protocol ID	Protocol
0-7Fh	Intel reserved
80-0FFh	Available for applications.

Protocol Handlers

A protocol handler is a procedure associated with a particular protocol ID. When the MPC receives a message, it interrupts the processor and causes the protocol handler associated with the message to be invoked. The application establishes protocol handlers using the **attach_protocol_handler** system call.

NOTE

Scheduling is stopped during the execution of a protocol handler and interrupts below the level configured for message passing are disabled. Thus, blocking and unsafe system calls should not be used in protocol handlers (see Table 6-1 on page 6-28). The **send_dl** system call should never be invoked from inside a protocol handler.

The following actions occur when an unsolicited message is received by the MPC:

1. The MPC interrupts the CPU, invoking the MPC interrupt handler.
2. The interrupt handler reads the message's protocol ID.
3. The interrupt handler invokes the appropriate protocol handler.
4. The protocol handler executes.

Sending Data Link Messages

The data link layer provides the **send_dl** system call for sending messages. The caller supplies the host ID and protocol ID in every message to specify a destination.

Send_dl can be used to transmit the following types of messages:

- Unsolicited: a message that arrives at a host without a prior request
- Broadcast: a message delivered to the handlers using the specified protocol ID on all hosts in the system
- Buffer request: an unsolicited message used to request a buffer for solicited data transfer
- Buffer grant: an unsolicited message used to grant a buffer request and initiate solicited data transfer
- Buffer reject: an unsolicited message used to reject a buffer request and reject a request for solicited data transfer

Buffer request and buffer grant messages contain a data descriptor to specify solicited data being sent or received.

Sending and Receiving MIC Type Messages

Message Interrupt Controller (MIC) messages, are used by some Multibus II hardware, but typically are not used with the MPC. (The MPC can be set to support message interrupt operation, but this precludes full unsolicited and solicited message support.) MIC messages are only four bytes in length; not the complete message structure typically used with the data link layer. However, an application can use Kernel data link system calls to send and receive MIC messages.

To send a MIC message to a MIC based host, use the **send_dl** system call, specifying a four-byte message (not the complete message structure). The first two bytes of the structure are the remote host ID, the next byte is unused, and the last byte is the type (KN_UNSOL). When the data link layer receives a four-byte message, it automatically routes the message to the protocol handler with ID 0 on the specified host.

To receive a MIC message, the application must attach a protocol handler with a protocol ID of 0. This is the only case where you should use a protocol ID in the Intel-reserved range 0-7Fh. The message remote host ID and type (KN_UNSOL) are the only defined fields of a received MIC message. The application defines the operation of the handler; it is treated like any other protocol handler by the Kernel.

Solicited Channel Usage

The MPC has a limited number of solicited input and output channels. The data link layer hides this fact from the application by queuing buffer requests and buffer grants when channel resources are busy. When an input or output channel becomes available, the data link automatically dequeues and transmits the next buffer request or grant. The data link informs a protocol handler that the solicited message has been transmitted, by invoking the protocol handler with the original buffer request or buffer grant message. The message's type field is changed to indicate that the send or receive is complete.

NOTE

Because protocol handlers participate directly in Multibus II buffer negotiation, errors within one protocol handler can cause unexpected behavior in a second. For example, a protocol handler that does not respond to a buffer request message with a buffer grant or buffer reject message can lock the remote MPC solicited output channel until the MPC timer expires. Applications that add protocol handlers must be aware of the potential conflict in solicited channel usage among multiple protocols.

Example of a Solicited Message Transmission

The following example illustrates the sequence of events for a solicited message transmission at the data link layer:

1. The user sends a buffer request message by calling **send_dl**.
2. **Send_dl** either transmits the buffer request message immediately or queues the message for later transmission.
3. The MPC of the receiver gets the buffer request message, causing an interrupt to the receiver's CPU.
4. The MPC interrupt handler of the receiver CPU invokes the appropriate protocol handler.

If the receiver is able to grant the request:

5. The receiver's protocol handler signals an application task that allocates an area for the solicited message. The task responds by calling **send_dl** with a buffer grant message, including the location of the allocated buffer.
6. **Send_dl** either transmits the buffer grant message immediately or queues the message for later transmission.

7. Once the buffer grant message is transmitted, the sending and receiving MPCs transfer the data.
8. When message transmission is complete, each MPC interrupts its respective CPU to indicate that the message is complete, and the protocol handlers are invoked with message indicating the send or receive process has completed.

If the receiver cannot grant the request:

5. When the receiver's protocol handler signals an application task to allocate buffer space, but there is no space available, the task responds with a buffer reject message in the **send_dl** system call.
6. The sender's MPC generates a completion interrupt.
7. The interrupt handler of the sender's MPC invokes the protocol handler with a message indicating the send is complete.

Figure 7-12 illustrates the relationship between the **send_dl** system call, the protocol handler and the application task.

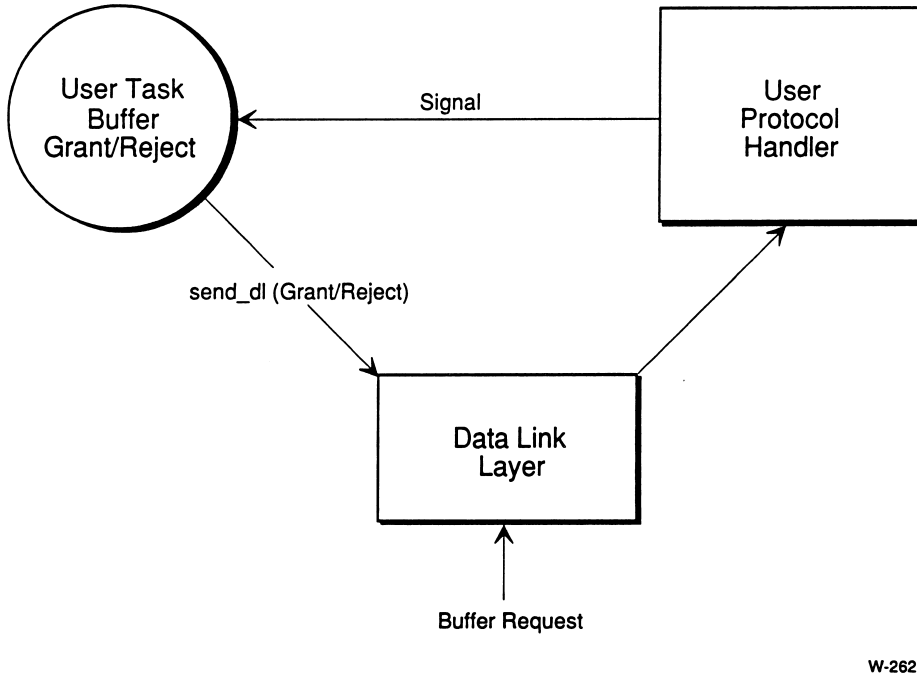


Figure 7-12. Sending a Data Link Message

Cancelling a Message at the Data Link Layer

The **cancel_dl** system call cancels a previously queued buffer request, buffer grant, or ongoing solicited message transfer. All information needed by this system call is embedded in the original message, referenced by the parameter passed to **cancel_dl**.

Summary of Data Link System Calls

The following is a list of the data link layer system calls:

- The **attach_protocol_handler** system call associates a user-written protocol handler with a particular protocol ID. Whenever a message arrives at the host, the data link layer's interrupt handler notes the protocol ID encoded in the message and invokes the corresponding protocol handler to deal with the message.
- The **cancel_dl** system call cancels a previously queued buffer request, buffer grant, or ongoing solicited message transfer.
- The **send_dl** system call sends a data link message to the specified protocol handler on the specified host.

Using the Transport Layer

The transport layer implements the Multibus II Transport Protocol, which supports request-response transactions between a client and a server. This protocol is described in the *Multibus II Transport Protocol Specification and Designer's Guide*

To use the transport layer, a client sends one request message with the **send_tp** system call, then waits for a response. The client does not have to issue a buffer grant to receive a response. The server, if possible, honors the client's request and then sends a response message to the client, also with the **send_tp** system call.

Both clients and servers receive messages by waiting at mailboxes, using the **receive_data** system call.

The main features of the transport layer include:

- support for message passing between Kernel tasks via port IDs
- transactions, which support request-response message passing
- the ability to use Kernel mailboxes as a means of synchronizing with asynchronous message-passing events

Request and response messages can be unsolicited or solicited messages. Solicited messages can be received in fragments.

Host IDs and Port IDs

A host ID is a 16-bit value that logically identifies a single host within the Multibus II system. By convention, the host ID is the number of the card slot in which the board is installed; the board in slot 4 has host ID 4. Typically, the host ID is automatically stored by MSA firmware in an interconnect record on the board at boot time. Host ID 31 always refers to the local host.

A port ID identifies the start point or end point of a message for communication between processors. The port ID is an arbitrary 16-bit number assigned by the application. The transport layer uses port IDs to route transport protocol messages within a host. Multiple tasks can use the same port ID; when a broadcast message is sent, it goes to the same port ID on all boards. The Kernel does not regulate port ID assignment. The application must maintain consistency when creating port IDs so that duplicate ID numbers are not used by tasks with different purposes.

To route a message to a particular port on a particular host within the system, specify both the host ID and the port ID. To send a message to a task on the same board, specify its port ID and host ID 31.

Reserved Port IDs

The Kernel does not specifically use port IDs for itself, so in general, Kernel applications can use any port ID to communicate between hosts managed by the Kernel. However, there are some port IDs the application must not use in certain circumstances:

- The RCI Server (for debugging communications) uses port ID 507H on the Kernel host.
- The C Server (for the C libraries) uses port ID 721 on the Kernel host.
- Intel operating systems running in the chassis reserve port IDs 0-800H for their own use. This includes System V and the PCI file server. When communicating with hosts not managed by the Kernel, the application must not send messages to port IDs in this range. Broadcast messages should not be sent to port IDs in this range.

If there is a chance your application might migrate to a different environment, which could include hosts not managed by the Kernel, do not use port IDs in the range 0-800H.

Unsolicited and Solicited Messages

In the transport layer, an unsolicited message can be up to 20 bytes in length (but is limited to 16 bytes in a buffer request). This message is sent as a *control* message in the message structure. The application defines the meaning of the control message for its own purposes.

A solicited message contains both the control message and a pointer to a separate data message. Typically, the control message defines action to be taken by the recipient, and the data message is data to be manipulated. For example, a client might send a *write* instruction in the control message to a file server, and point to the data to be written. The solicited message consists of a buffer request and buffer grant, as described earlier.

Request-Response Transactions

The Transport Protocol defines the concept of request-response transactions. In a transaction, a client task sends a solicited or unsolicited message called a request message, and supplies a response buffer. When a server task on the destination host receives a request message, it is expected to send a response message back to the client. The response buffer supplied by the client is used to hold the incoming response message. Both client and server use the **send_tp** call.

It is important to understand that a request-response transaction is not the same as a buffer request-buffer grant cycle. A buffer request-buffer grant is part of a *solicited* message, which can occur in both the data link and transport layers. A *transaction* (request-response) is only supported by the transport layer. The transaction may include a solicited message, in either the request or the response part, or in both. Thus a buffer request-buffer grant cycle may occur during a transaction, but does not define the request-response cycle of the transaction.

Transaction IDs

A transaction ID, along with a host ID and port ID, uniquely identifies a particular transaction. A transaction includes all activity beginning with a client request and including the entire server response. A client supplies a transaction ID as part of the request message to the server. When the server responds, the transaction ID is also part of the response message. The transport layer uses this transaction ID to match request and response, and to locate a client-supplied response buffer.

Specifying a transaction ID of zero indicates that the message is not a transaction. The message is a request only and does not require (or expect) a response message.

Transaction IDs can range from 1 through 255. To support more than 255 outstanding requests at a single communication endpoint, use multiple port IDs.

NOTE

It is the responsibility of the application to use transaction IDs consistently. That is, no two requests for the same port should use the same transaction ID. A transaction ID that becomes invalid at one endpoint while remaining outstanding at a second can cause erroneous results.

Mailboxes and Message Passing

Rather than notifying applications through interrupts that messages have been received, the transport layer queues incoming messages in application-supplied mailboxes. The application invokes the **attach_receive_mailbox** system call to associate a mailbox with a specified port ID. This mailbox will then receive incoming messages for the specified port ID. By associating individual mailboxes with specific functions, you can meet the particular needs of your application. In the simplest case, you can use a single mailbox for all types of solicited and unsolicited messages for a given port ID.

When sending a message, you also specify a mailbox to receive response messages and solicited transfer completion messages.

The uses for mailboxes in the transport layer include:

- Receive mailboxes are typically used by a server task to receive messages sent to a specified port ID. Unsolicited messages and buffer request messages are queued at these mailboxes. The mailbox is specified in an **attach_receive_mailbox** system call.
- Completion mailboxes are used by client and server tasks. These mailboxes are used for two purposes: to receive a response message in a request-response transaction and to receive solicited transmission completion messages. The mailbox is specified in a **send_tp** system call.

The Format of Transport Messages

When you use the transport layer, the Kernel sends the message using the data link layer. The Kernel supplies transport handlers as data link protocol handlers. The transport message structure overlays the data link message format. A transport message uses the data link `unsol_data` field for the port ID addressing and transaction control fields, and appends completion mailbox tokens to the end of the data link messages.

When a client sends a request message, initiating a transaction, it supplies a double message structure, containing a request message and a response message part. Each part uses the basic transport message template; both parts together comprise a single request-response message.

Figure 7-13 shows how the basic transport message template is duplicated to create a request-response message.

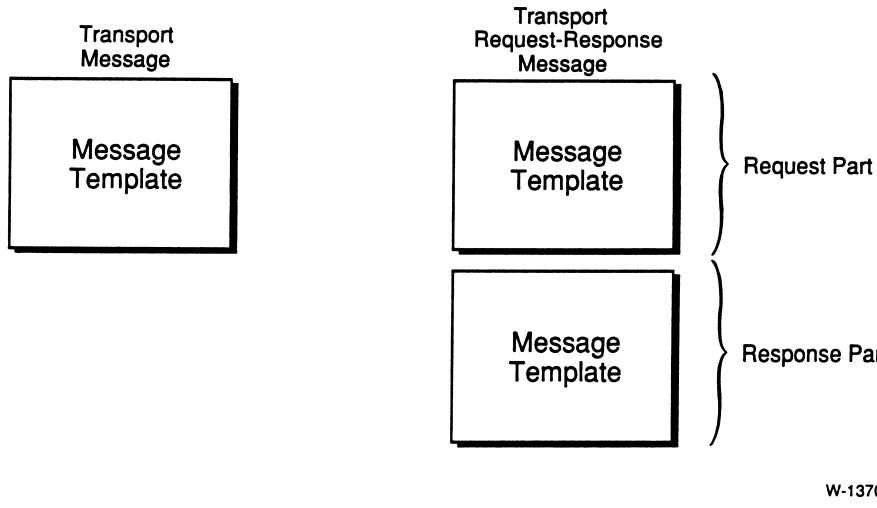


Figure 7-13. Message Templates for Transport Messages

Cancelling Transport Messages

A solicited message can be cancelled any time after the buffer request or buffer grant is queued, until the transfer is complete. For a request-response transaction, a message can be cancelled any time after the request is sent and until the response is complete.

To cancel a transport message, invoke the **cancel_tp** system call with a pointer to the message sent in the **send_tp** system call. **Cancel_tp** returns the status of the operation, indicating whether the message was cancelled or had already been sent and the cancellation was too late.

Fragmentation Support

Message receivers (server tasks) must provide memory to receive solicited and unsolicited messages. Since unsolicited messages are embedded in the transport message structure, there is usually not a buffering problem with them. However, solicited messages are data external to the structure; they may be too large to be sent or received in a single transfer. To address this problem, the transport layer supports message fragmentation.

Only transaction messages may be fragmented. A server task may fragment a request message it receives or a response message it sends. Fragmentation may be used with data chains as well as with contiguous buffers.

Request Message Fragmentation

In the **send_tp** system call, the client specifies in its buffer request message whether the message may be received by the server in fragments. If the client indicates that the message may be fragmented and the server cannot receive the message in a single transfer, the server responds with a buffer reject message. This message indicates to the transport layer that fragmentation is necessary. The transport layer handles the fragmentation. It is the server's responsibility to receive the message in fragments. The client never knows whether the message was received in fragments.

Following its buffer reject message, the server sends a message specifying the size of fragments it can receive, based on the server's ability to allocate buffer space. The server repeatedly requests and receives fragments until no more fragments are left. The transport layer satisfies the requests without the intervention of the client. The server determines how many fragments to receive by the length of the data being sent, which is embedded within the original request message.

Response Message Fragmentation

If the server does not have enough buffer space to return a large response message in a single transfer, the server can send the message in fragments. For example, this might occur if a client makes a large read request from a file server, but the server does not have enough buffer space available to hold all the data at once. The fragmentation is based on the availability of server resources.

The server specifies fragmentation by sending a buffer reject, then sending the response in fragments. When sending message fragments, the server sets the `trans_control` field of the response message set to KN_NOT_EOT. When the server sends the final fragment, it sets the `trans_control` field of the response message to KN_EOT, indicating that this fragment is the end of the transaction.

If a response message is fragmented, the transport layer assembles the fragments of the response in the client's response buffer as the server sends each fragment. When the complete response has been transferred, a reference to the entire response message is queued at the client's response mailbox for that transaction. The client never knows the message has been fragmented. When the client initiates a transaction, it must always allocate a buffer large enough to receive the size response it requests.

Summary of Transport Layer System Calls

The following is a list of the transport layer system calls:

- The **attach_receive_mailbox** system call associates a mailbox with a port ID. Once this association has been made, all incoming messages for the specified port ID are queued at the mailbox.
- The **cancel_tp** system call cancels an ongoing solicited message or request-response transaction. It performs a local operation and does not send a message to the other host.
- The **receive_data** system call, though not specifically part of the transport layer, is used to receive messages sent with the **send_tp** call.
- The **send_tp** system call sends a transport protocol message.

Examples of Transport-Layer Message Passing

There are a variety of message types that can be sent with the transport layer. This section gives several examples of message-passing scenarios, showing the sequence of system calls used to send and receive messages. The examples show a client-server relationship, where the client task initiates the message-passing sequence and the server task responds according to values received in the message. Both client and server may send and receive messages, depending on the message type.

Figure 7-14 shows the basic structure of all messages sent (with the **send_tp** system call) and received (from a mailbox with the **receive_data** system call). When a client initiates a transaction (request-response), it sends two of these KN_TRANSPORT_MSG structures: the first is a request block and the second is a response block. Figure 7-15 shows the structure that defines this double block.

When a task sends a message, it fills in the `remote_host_ID`, `dst_port_ID`, `src_port_ID`, and `trans_ID` (transaction ID) fields shown in Figure 7-14. When a task receives a message, the `remote_host_ID` field is changed by the transport software to indicate the host where the message originated. If the receiving task returns a message, it sends to the `remote_host_ID` and `trans_ID` in the message it received, but exchanges the `dst_port_ID` and `src_port_ID` from those in the message it received. A transaction ID of zero specifies a non-transaction.

The `trans_control` field indicates whether the message is the request or the response part of a transaction. Other values can be ORed with one of these choices to control fragmentation, as indicated at the bottom of Figure 7-14.

The only application data actually embedded in the message block is the `control_message` field. This is a 20-byte field defined by the application for its own purposes (only the first 16 bytes can be used in a buffer request message). In an unsolicited message, this is the only data transmitted. In a solicited message (buffer request-buffer grant), the `control_message` is used as an instruction for dealing with another, larger data block, indicated by the `data_type`, `data_length`, and `data_ptr` fields. For example, a client might send a file server a "write" instruction in `control_message`, with the data to be written indicated by `data_ptr`.

The `data_type` field indicates whether data is in a contiguous block (KN_SEGMENT) or in a data chain (KN_CHAIN), as shown in Figure 7-11 on page 7-49. If the message being sent is a buffer request, the `data_type`, `data_length`, and `data_ptr` fields refer to data being sent. If the message being sent is a buffer grant, these fields indicate where and how data is to be received.

A task specifies the `completion_mbx` (mailbox) when it has requested a return message or to check the status of a message previously sent. In the message returned by a completion mailbox the Kernel sets the `data_status` field. The `reserved`, `dl_part`, and `data_ptr_fill` fields are not used by the client and server tasks.

```
typedef struct {
    UINT_8              reserved1 [16];
    KN_HOST_ID          remote_host_ID;        /*  Ø-2Ø, or 31 for local host */
    UINT_8              reserved2 [4];
    KN_MESSAGE_TYPE     type;                          /* KN_UNSOL, KN_BROADCAST,
                                                KN_BUFFER_REQUEST, KN_BUFFER_GRANT,
                                                 KN_BUFFER_REJECT, KN_SEND_COMPLETE
                                                          KN_RECEIVE_COMPLETE */
    UINT_8              dl_part [7];         /* reserved for data link overlay */
    KN_PORT_ID          dst_port_ID;            /* port ID to receive message */
    KN_PORT_ID          src_port_ID;               /* port ID sending message */
    KN_TRANS_ID         trans_ID;               /* 1-255, or KN_NO_TRANSACTION */
    KN_TRANS_CONTROL    trans_control;                          /* see below */
    UINT_8              control_message [2Ø];          /* application-defined */
    KN_STATUS           data_status;        /* set by Kernel on msg completion */
    KN_DATA_TYPE        data_type;                 /* KN_SEGMENT, KN_CHAIN */
    UINT_8              reserved3 [3];
    UINT_32             data_length;
    UINT_8        far * data_ptr;           /* see address range in Table 7-2 */
    UINT_16             data_ptr_fill;
    UINT_8              reserved4 [2];
    KN_TOKEN            completion_mbx;
    UINT_8              transport_reserved [64];
} KN_TRANSPORT_MSG;


/* Literals used in the trans_control field
KN_REQUEST                                  KN_RESPONSE
  can be ORed with                            can be ORed with
  KN_NO_FRAGMENTATION    (client)             KN_NOT_EOT   (server)
  KN_FRAGMENTATION       (client)             KN_EOT       (server)
  KN_SEND_NEXT_FRAGMENT  (server)             KN_CANCEL    (set by Kernel)
  KN_NEXT_FRAGMENT       (set by Kernel)                              */
```

Figure 7-14. The KN_TRANSPORT_MSG Structure

```
typedef struct {
    KN_TRANSPORT_MSG   request_message;
    KN_TRANSPORT_MSG   response_message;       /* fields remote_host_ID through
                                                  trans_control are not used */
} KN_RSVP_TRANSPORT_MSG;
```

Figure 7-15. The KN_RSVP_TRANSPORT_MSG Structure

Structures Used to Receive Messages

The previous discussion implied that the KN_TRANSPORT_MSG shown in Figure 7-14 is the message received as well as the message sent. This is not entirely true. In some cases, the message received at the mailbox contains the first, significant, parts of the KN_TRANSPORT_MSG structure. In all other cases, the message received at the mailbox contains a pointer to a message structure. The difference is as follows:

- Port mailboxes, specified in an **attach_receive_mailbox** system call, receive the actual message, which includes the fields up through `data_length` shown in Figure 7-14. The receiving task first overlays another structure to identify the message.
- Completion mailboxes, specified in the `completion_mbx` field of a message being sent, receive a pointer to a message. The pointer indicates the original message block sent by the task, with values updated by the Kernel.

To keep track of the kinds of messages being received, it is simpler not to use the same mailbox as both a port and completion mailbox. If the application uses the same mailbox for both purposes, tasks must be able to distinguish between the two types of messages received. The example scenarios in this chapter show when to use a port mailbox or a completion mailbox.

Receiving Messages at a Port Mailbox

Only tasks classified as servers set up port mailboxes. One mailbox can be used to service more than one port. Often many (identical) server tasks wait at a single mailbox; when one receives a message it services the message, leaving the next task in the queue to receive the next message.

Figure 7-16 shows the structure received at a port mailbox. The first two bytes of the structure are set to the literal value KN_REMOTE_MSG. These two bytes overlay the first two bytes of the `reserved1` field in the KN_TRANSPORT_MSG. After the receiving task identifies this as a KN_REMOTE_MSG, it overlays the KN_TRANSPORT_MSG structure to extract the values it needs. Only the fields `reserved1` through `data_length` are received. (The `data_ptr` field is not significant for messages sent from one host to another.)

```
typedef struct{
    UINT_16             mbx_message_type;                    /* KN_REMOTE_MSG */
    UINT_8              mbx_message [66];
} KN_TRANSPORT_MBX_REMOTE_MSG;
```

Figure 7-16. The KN_TRANSPORT_MBX_REMOTE_MSG Structure

The message received at a port mailbox is not as long as a KN_TRANSPORT_MSG. Before calling **receive_data**, the server task should allocate a buffer the size of KN_TRANSPORT_MSG and pass a pointer to this buffer in the **receive_data** call. When it receives a message, the server can treat the message as if it were a complete KN_TRANSPORT_MSG. If the server returns a message, it can change values in the received message and use the same memory as a return message structure.

CAUTION

Other data will be overwritten if the server allocates only enough memory for the KN_TRANSPORT_MBX_REMOTE_MSG structure, and then uses this memory to send a return message.

Receiving Messages at a Completion Mailbox

Both clients and servers use completion mailboxes. The mailbox is specified when a task sends a message; it serves one of the following purposes:

- to receive an indication that data has been transferred from a solicited message or to get the final status of the solicited message
- to get the response from a transaction message
- when a server uses fragmentation, to receive an indication that the current fragment has been transferred

Figure 7-17 shows the structure received at a completion mailbox. The first two bytes of the structure are set to the literal value KN_LOCAL_MSG, to identify the type of message. This is followed by a pointer to a message previously sent by the task. (The `mbx_message_fill` field is a placeholder for small model applications; in compact model this memory holds the selector of the pointer.) When the completion message is received, the original message block has been updated with returned values. The task that specifies a completion mailbox must maintain the original message structure until receiving a completion message. Depending on whether the task originally sent a KN_TRANSPORT_MSG or a KN_RSVP_TRANSPORT_MSG, this is the structure indicated by the pointer.

```
typedef struct {
    UINT_16            mbx_message_type;                    /* KN_LOCAL_MSG */
    UINT_8       far * mbx_message;
    UINT_16            mbx_message_fill;
} KN_TRANSPORT_MBX_LOCAL_MSG;
```

Figure 7-17. The KN_TRANSPORT_MBX_LOCAL_MSG Structure

Example A — Unsolicited, Non-Transaction Message

Figure 7-18 shows the system calls used to send and receive an unsolicited, non-transaction message. The client sends a message with no accompanying data, and does not expect a response. The steps in Figure 7-18 are described below.

1 The server attaches a mailbox to a port. If a client sends a message to a port ID with no attached mailbox, the message is dropped. The **send_tp** call (step 3) always returns E_OK unless the receiving host board is not running, so in this example the client would not know the message wasn't received. Further examples show how to use a completion mailbox to get the final status of **send_tp**.

2 The server waits at the port mailbox until it receives a message.

3 The client sends an unsolicited, non-transaction message, specified by the `type` and `trans_ID` fields. Besides the fields shown in step 3, the client specifies the remote host ID, the source and destination port IDs, and the control message. Instead of KN_UNSOL, the type could be KN_BROADCAST, which is an unsolicited message sent to the same port ID on every host in the system. The remote host ID is ignored for a broadcast message.

4 The server receives the message and evaluates the contents. Because the type is KN_UNSOL, it doesn't need to receive any further data; received data is in the control message. Because the transaction ID is KN_NO_TRANSACTION, it doesn't need to send a response. The server proceeds to act on the message in the `control_message` field, according to the convention set up by the application.

5 The server waits for another message at the port mailbox.

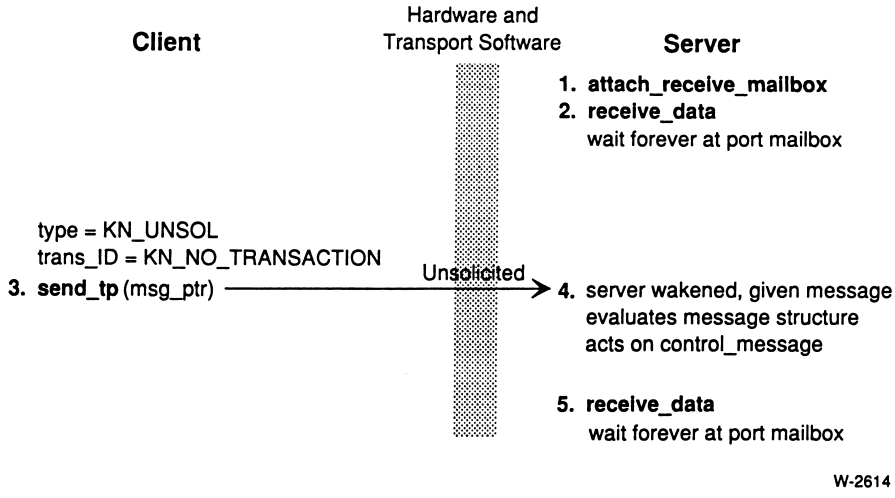


Figure 7-18. An Unsolicited, Non-Transaction Message

Example B — Solicited, Non-Transaction Message

Figure 7-19 shows the system calls used to send and receive a solicited, non-transaction message. The client sends data to the server, but does not expect a response.

1 & 2 The server attaches a mailbox to a port and waits at the mailbox for a message.

3 The client sends a solicited, non-transaction message, specified by the `type` and `trans_ID` fields. The data to be sent with the message is indicated with the `data_type`, `data_length`, and `data_ptr` fields. Besides the fields shown in step 3, the client specifies the remote host ID, the source and destination port IDs, and the control message. The control message tells the server how to act on the data. Because this is a solicited request, the client has the option of specifying a completion mailbox to receive the final status of the **send_tp** call.

3(a) If a completion mailbox is specified, the client waits at the mailbox. As the figure indicates, a message is not received until steps 4 and 5 have completed. The received message contains a pointer to the KN_TRANSPORT_MSG sent in step 3. The client checks the `type` and `data_status` fields to get the final status. The `type` should be KN_SEND_COMPLETE and the `data_status` should be E_OK. If the server board is not running, the error is indicated in the `data_status` field. If the server does not have a mailbox attached to the port, there is a timeout error (but if the failsafe timer is not enabled, no message is received at the completion mailbox).

4 The server receives the message and evaluates the contents. Because the transaction ID is KN_NO_TRANSACTION, it doesn't need to send a response. Because the type is KN_BUFFER_REQUEST, it needs to receive further data. The server checks the `data_length` to see how much data it will receive and allocates a memory buffer.

5 To receive the data, the server sends a buffer grant message. In the `data_type`, `data_length`, and `data_ptr` fields the server indicates the buffer to receive the data. The server specifies a completion mailbox to receive notice that the data has arrived. In addition to the fields shown in step 5, the server specifies the remote host ID and transaction ID received in step 4, and exchanges the source and destination port IDs received in step 4.

The next part of step 5 is handled by the hardware and transport software. The transport software receives the buffer grant and initiates a DMA of the data into the server's buffer. When the data is transferred, the respective MPC devices send a completion interrupt, signaling the software to send a completion message to the respective mailboxes.

6 The server receives a message containing a pointer to the message block it sent in step 5. The `type` should be KN_RECEIVE_COMPLETE and the `data_status` should be E_OK. The server proceeds to act on the data received in step 6 according to the control message received in step 4.

7 The server waits for another message at the port mailbox.

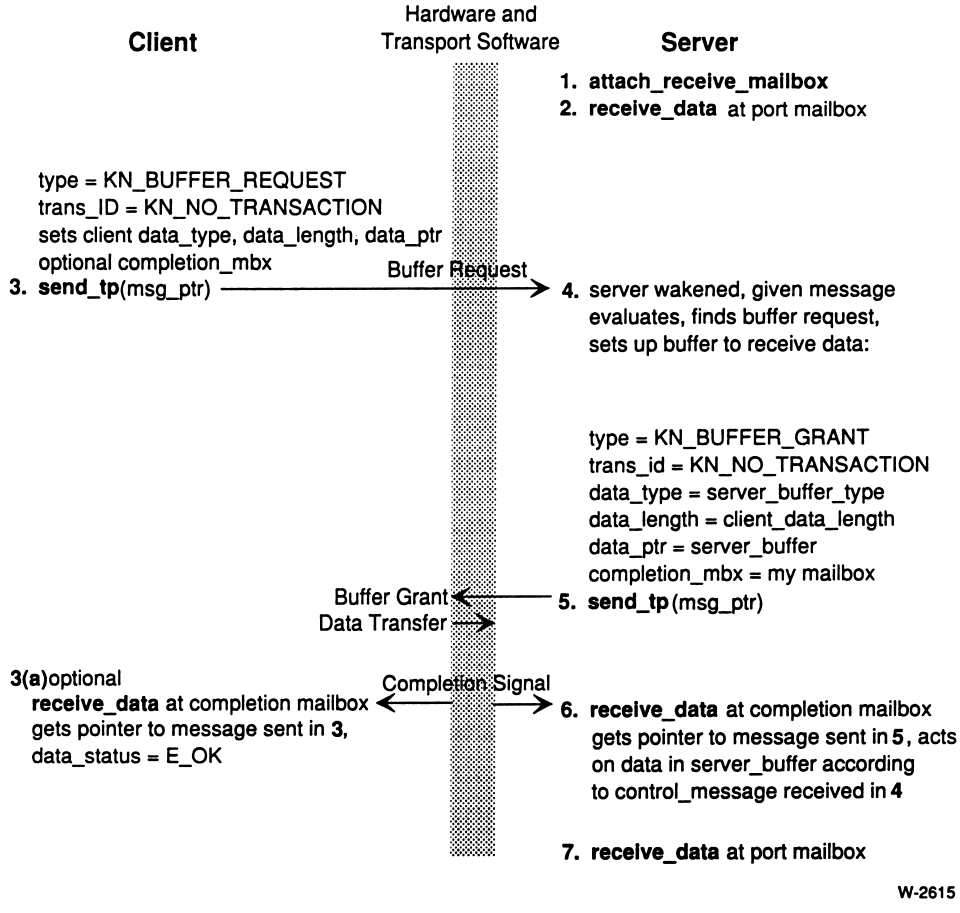


Figure 7-19. A Solicited, Non-Transaction Message

Fragmentation Not Allowed

The server must have enough memory available to receive the solicited data in one transfer, either in a contiguous buffer (`data_type` = KN_SEGMENT) or a data chain (`data_type` = KN_CHAIN). Data fragmentation is not allowed in a non-transaction message.

Example C — Unsolicited Transaction

Figure 7-20 shows the system calls used to send and receive a transaction with both the request and response parts unsolicited. The client requires a response from the server, but does not expect a large amount of data to be returned.

1 & 2 The server attaches a mailbox to a port and waits at the mailbox for a message.

Request Phase

3 The client sends an unsolicited transaction message; a non-zero transaction ID specifies that this is a transaction. The message consists of two KN_TRANSPORT_MSG structures (see Figure 7-15). The first is a request block and the second is a response block. In the request block, the client sets the fields shown in step 3, along with the remote host ID, the source and destination port IDs, and the control message. The control message contains an instruction determined by the application that tells the server to send an unsolicited message (not to send data) in the response phase. In the response block, the only field the client is required to specify is a completion mailbox. However, a good precaution is to also set the `data_length` field to zero. If the server should accidentally respond with a solicited message, the transport software finds the zero-length field and does not attempt to return data to a buffer where the client isn't prepared to receive it.

4 The server receives the request block part of the message and evaluates the contents. Because the type is KN_UNSOL, it doesn't need to receive any further data. Because the transaction ID is not zero (not = KN_NO_TRANSACTION), it needs to send a response. The server understands from the (application-defined) control message that the response is to be unsolicited. The server proceeds to act on the control message, according to the convention set up by the application.

Response Phase

5 The server sends a single KN_TRANSPORT_MSG as a response (only the initiator of a transaction sends a double message block). It uses the message received in step 4 as the template for the response, changing the `trans_control` field to KN_RESPONSE, exchanging the source and destination port IDs, and sending the response data in the control message. Since this is an unsolicited message, the server does not specify a completion mailbox. The server proceeds to step 7.

6 The client waits at the completion mailbox specified in the response block in step 3. The message received at the mailbox contains a pointer to the original (double-block) message structure. In the response block, the client checks the `trans_control` (set to KN_RESPONSE by the Kernel) and gets the control message sent in step 5.

7 The server waits for another message at the port mailbox.

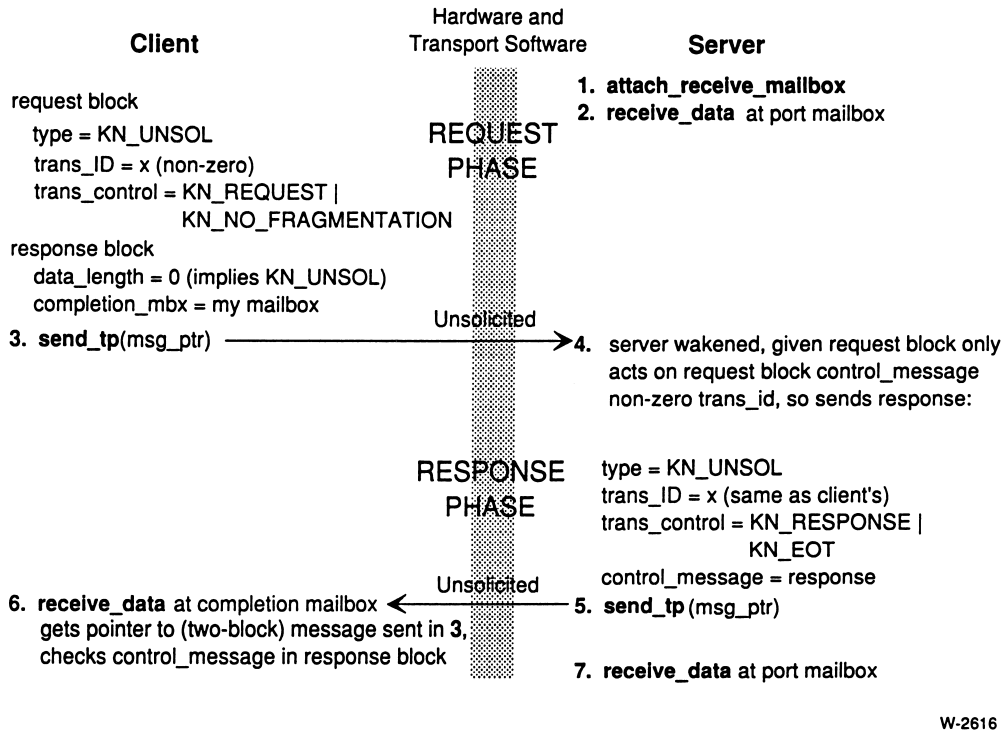


Figure 7-20. An Unsolicited Transaction (Request-Response)

NOTE

The client may send other messages between steps 3 and 6. However, it must not use the same transaction ID specified in step 3 until it receives the completion message in step 6.

Example D — Unsolicited Request and Solicited Response

Figure 7-21 shows the system calls used to send and receive a transaction with an unsolicited request and a solicited response. This would be typical of a client sending a file server a request to read data from disk. The client sends a small request and no data, but expects a large amount of data to be returned.

1 & 2 The server attaches a mailbox to a port and waits at the mailbox for a message.

Request Phase

3 As in Example C, the client sends a message consisting of a request block and a response block. In the request block, the client sets the fields shown in step 3, along with the remote host ID, the source and destination port IDs, and the control message. The control message contains an instruction determined by the application that tells the server to respond with a solicited message, and the maximum length data that can be sent in the response. In the response block, the client specifies where to return data (`data_type` and `data_ptr`) and the size of the buffer allocated (`data_length`). (The server does not receive the information in the response block; it is used by the transport software.) The client specifies a completion mailbox in the response block to receive the signal that data has been returned.

4 The server receives the request block part of the message and evaluates the contents. Because the type is KN_UNSOL, it doesn't need to receive any further data. Because the transaction ID is not zero, it needs to send a response. The server understands from the (application-defined) control message that the response is to be solicited, containing an application-determined amount of data. The server proceeds to act on the control message, according to the convention set up by the application.

Response Phase

5 The server sends a buffer request as a response message, using the message received in step 4 as the template, and changing the fields shown in step 5 (along with the source and destination port IDs). The server indicates the data it is sending with the `data_type`, `data_length`, and `data_ptr` fields. It cannot send more data than the client requested. The server sends a control message appropriate to the application. It specifies a completion mailbox to receive the signal that the transaction is complete, so it can resume waiting for messages at the port mailbox.

The hardware and transport software receive the buffer request, initiate a buffer grant specifying the location of the client's (response block) data buffer, and transfer the data into the client's buffer. When the data is transferred, a completion message is sent to the client and server mailboxes.

6 The client waits at the completion mailbox specified in the response block. The received message points to the original (double-block) message structure. In the response block, the client gets the control message and checks `trans_control` (for KN_RESPONSE), `type` (for KN_RECEIVE_COMPLETE), and `data_length`. The `data_length` indicates the amount of data actually returned.

7 At its completion mailbox, the server receives a pointer to the message sent in step 5. The server checks the `type` (for KN_SEND_COMPLETE) and `data_status` fields.

8 The server waits for another message at the port mailbox.

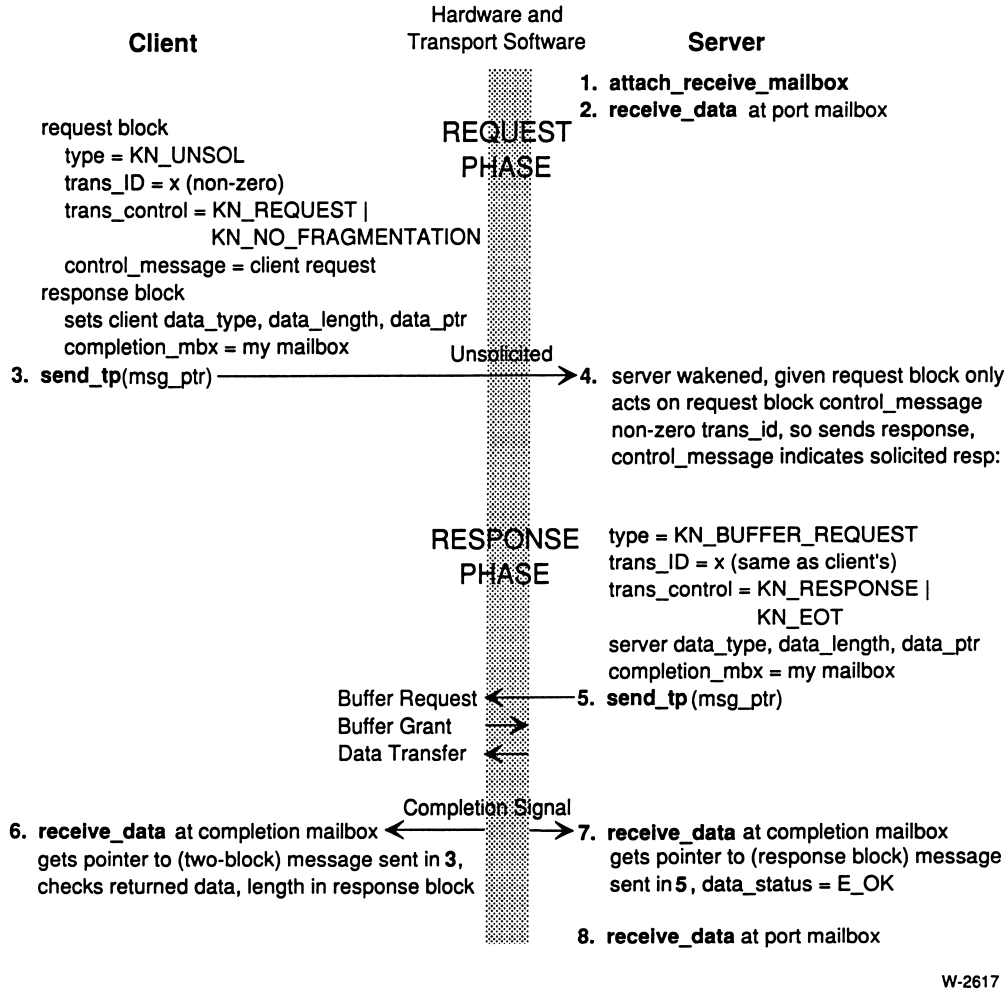


Figure 7-21. An Unsolicited Request and Solicited Response

Fragmenting Data in the Response Message

It may happen that the server in step 5 (Figure 7-21) does not have a large enough buffer to hold all the data requested by the client. If this occurs, the server can fragment the data sent in the response message. This kind of fragmentation is not affected by a KN_FRAGMENTATION or KN_NO_FRAGMENTATION flag set by the client (see `trans_control` in step 3).

To send data in fragments, the server uses a series of **send_tp** (step 5) and **receive_data** (step 7) calls. In step 5, it sets `trans_control` to KN_RESPONSE OR KN_NOT_EOT, with `data_length` set to the length of the fragment. In step 7, it receives a pointer to the message sent in step 5, and it sends another fragment. When sending the last fragment, the server sets `trans_control` to KN_RESPONSE OR KN_EOT. In the final step 7, the server checks the `type` (for KN_SEND_COMPLETE) and `data_status` fields.

Meanwhile, the hardware and transport software transfer each fragment into the client's data buffer. Only after the final fragment is transferred does the client receive the completion message shown in step 6. The client does not receive the message in fragmented form.

Even when using fragmentation to send the solicited data, the server cannot send more data than the client requested. This value must be determined by the application, either by convention or by information in the control message. The total of fragments sent by the server cannot exceed the `data_length` the client sets in its response block in step 3.

Example E — Solicited Request and Unsolicited Response

Figure 7-22 on page 7-76 shows the system calls used to send and receive a transaction with a solicited request and an unsolicited response. This would be typical of a client sending a file server data to be written to disk. The client sends data and wants a response, but does not expect a large amount of data to be returned.

1 & 2 The server attaches a mailbox to a port and waits at the mailbox for a message.

Request Phase

3 The client sends a message consisting of a request block and a response block. In the request block, the client sends a buffer request, initiating a solicited request. The client sets the fields shown in step 3, along with the remote host ID, the source and destination port IDs, and the control message. The control message contains an instruction determined by the application that tells the server to respond with an unsolicited message. The `data_type`, `data_length`, and `data_ptr` fields indicate the data to be transferred with the message. The client may specify a completion mailbox in the request block to receive the status of the solicited transfer.

In the response block, the client must specify a completion mailbox to receive the response message. If a completion mailbox is also specified in the request block, this must be a different mailbox. Both mailboxes receive a pointer to the same message, and the client must keep track of which mailbox is for checking status and which is for receiving the response message. The only other field the client specifies in the response block is to set `data_length` to zero (as in Example C).

3(a) The message received at the optional request-block completion mailbox points to the message sent in step 3. The client must know that this mailbox is for checking status of the solicited transfer. The client checks the `type` (for KN_SEND_COMPLETE) and `data_status` fields.

4 The server receives the request block part of the message and evaluates the contents. Because the type is KN_BUFFER_REQUEST, it needs to receive further data. Because the transaction ID is not zero, it also needs to send a response. The server understands from the control message that the response is to be unsolicited. The server checks the `data_length` field to see how much data it will receive and allocates memory to hold the data.

5 To receive the data, the server sends a buffer grant message, indicating the buffer to receive the data. In addition to the fields shown in step 5, the server specifies the remote host ID and transaction ID received in step 4, and the source and destination port IDs.

The hardware and transport software receive the buffer grant and transfer the data into the server's buffer. When the data is transferred, a completion message is sent to the client (if specified) and server mailboxes.

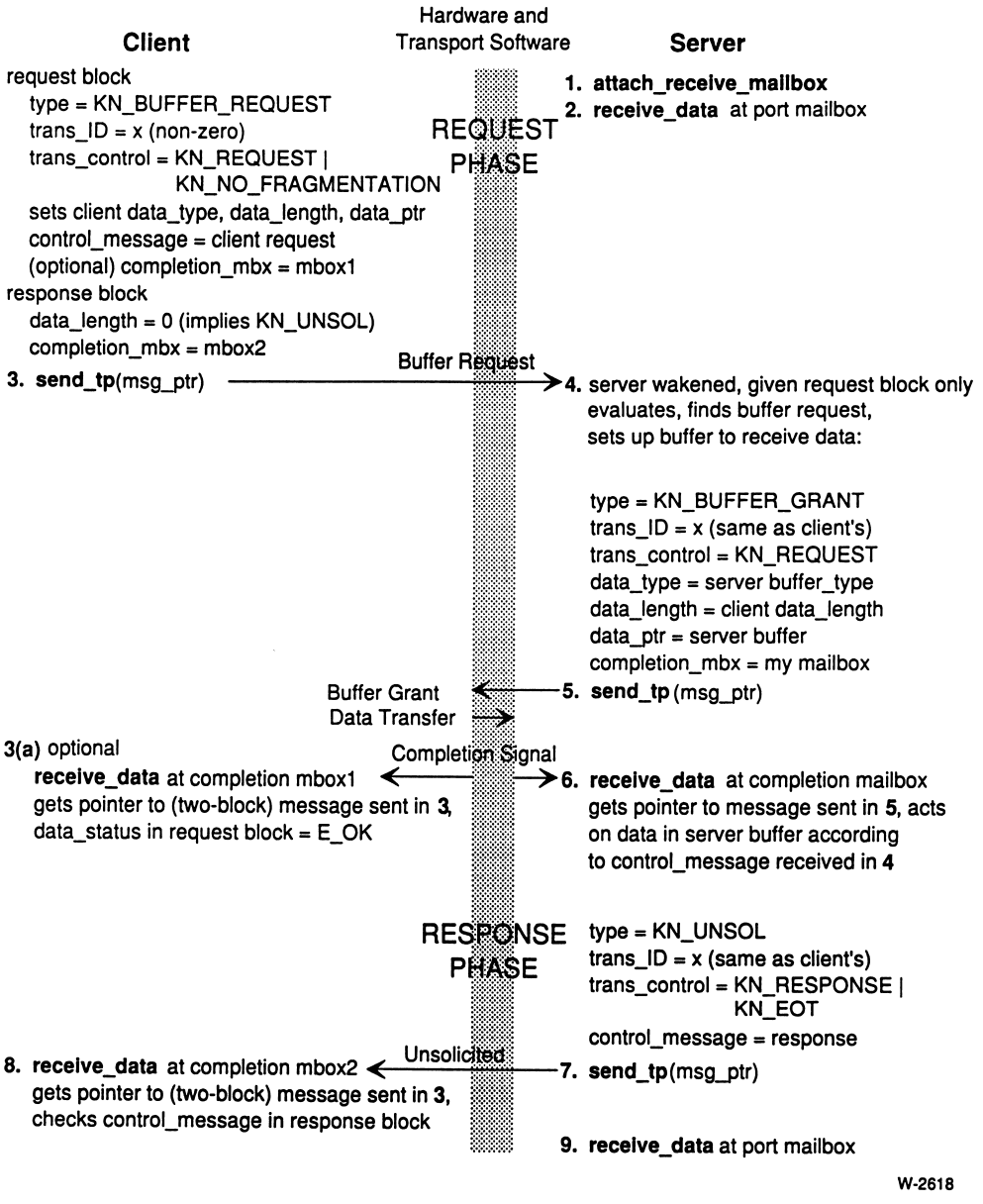


Figure 7-22. A Solicited Request and Unsolicited Response

6 The server receives a message containing a pointer to the message block it sent in step 5. The server proceeds to act on the data received in step 6 according to the control message received in step 4.

Response Phase

7 The server sends an unsolicited response message in the `control_message` field. It could reuse the same message buffer as in step 5, changing the fields shown in step 7. The message does not specify a data buffer or a completion mailbox.

8 The client waits at the completion mailbox for the response block. The received message points to the original (double-block) message structure. In the response block, the client gets the control message sent by the server (`data_status` would be E_CANCELLED if the server didn't attach a mailbox to the port in step 1).

9 The server waits for another message at the port mailbox.

Fragmenting the Received Data

The client in step 3 controls the potential for fragmenting the data received by the server. In the `trans_control` field the client ORs KN_REQUEST with either KN_FRAGMENTATION or KN_NO_FRAGMENTATION. If the server in step 4 does not have enough memory to receive `data_length` bytes, it checks the `trans_control` field to see if fragmentation is allowed. Whether fragmentation is allowed or not, in step 5 the server sends a buffer reject message (`type` = KN_BUFFER_REJECT) rather than a buffer grant, with no completion mailbox. One of the following ensues:

KN_NO_FRAGMENTATION. If fragmentation is not allowed, the server does not expect to receive any data after the buffer reject, and proceeds to step 7. When the transport software receives the buffer reject, it drops the rest of the message. In step 3(a), the client's message block is set to E_SO_CANCEL in the `data_status` field.

KN_FRAGMENTATION. If fragmentation is allowed, the server follows the buffer reject message with another call to **send_tp**. In this call (step 5.5), `type` is KN_UNSOL; `trans_control` is KN_REQUEST OR KN_SEND_NEXT_FRAGMENT. The server sets `data_length` to the amount of data it can receive in one fragment. The transport software sends only that amount of data, with `trans_control` set to KN_REQUEST OR KN_NEXT_FRAGMENT. The server receives the data in step 6 and sends another (step 5.5) message requesting the next fragment. This cycle repeats until the server receives all the data. In each cycle through step 6, the server checks the `type` (for E_OK) and `data_status` (for KN_RECEIVE_COMPLETE) to make sure it has received the fragment. The server must keep track of the amount of data received, and issue no more **send_tp** calls after receiving the last fragment (but it proceeds to invoke the step 7 **send_tp** call). When the last fragment is transferred, step 3(a) occurs, with the client's `type` field set to KN_SEND_COMPLETE. The client gets no indication that the message was fragmented.

Example F — Solicited Request and Solicited Response

Figure 7-23 shows the system calls used to send and receive a transaction with both the request and response solicited.

1 & 2 The server attaches a mailbox to a port and waits at the mailbox for a message.

Request Phase

3 The client sends a message consisting of a request block and a response block. In the request block, the client sends a buffer request, initiating a solicited request. The client sets the fields shown in step 3, along with the remote host ID, the source and destination port IDs, and the control message. The control message contains an instruction determined by the application that tells the server to respond with a solicited message, and the maximum length data that can be sent in the response. The `data_type`, `data_length`, and `data_ptr` fields indicate the data to be transferred with the message. The client may specify a completion mailbox in the request block to receive the status of the solicited transfer.

In the response block, the client specifies where to return data (`data_type` and `data_ptr`) and the maximum amount of data to be returned (`data_length`). The client specifies a completion mailbox in the response block to receive the response message. This must be a different mailbox from the one specified in the request block. The server does not receive the information in the response block; it is used by the transport software.

3(a) The message received at the optional request-block completion mailbox points to the message sent in step 3. The client must know that this mailbox is for checking status of the solicited transfer. The client checks the `type` (for KN_SEND_COMPLETE) and `data_status` fields.

4 The server receives the request block part of the message and evaluates the contents. Because the type is KN_BUFFER_REQUEST, it needs to receive further data. Because the transaction ID is not zero, it also needs to send a response. The server understands from the control message that the response is to be solicited, containing an application-determined amount of data. The server checks the `data_length` field to see how much data it will receive in the request phase and allocates memory to hold the data.

5 To receive the data, the server sends a buffer grant message, indicating the buffer to receive the data. In addition to the fields shown in step 5, the server specifies the remote host ID and transaction ID received in step 4, and the source and destination port IDs.

The hardware and transport software receive the buffer grant and transfer the data into the server's buffer. When the data is transferred, a completion message is sent to the client (if specified) and server mailboxes.

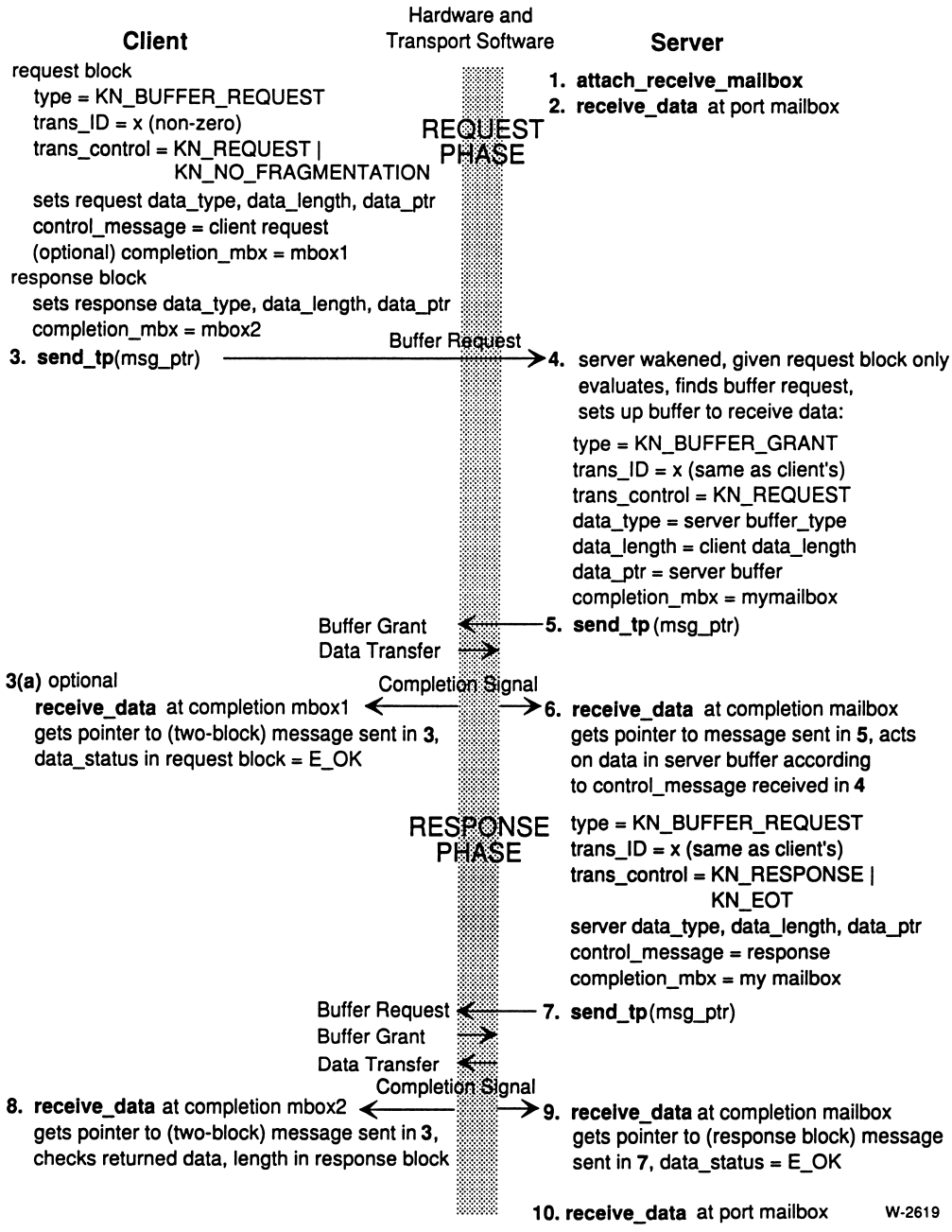


Figure 7-23. A Solicited Request and Solicited Response

6 The server receives a message containing a pointer to the message block it sent in step 5. The server proceeds to act on the data received in step 6 according to the control message received in step 4.

Response Phase

7 The server sends a buffer request as a response message. It could reuse the same message buffer as in step 5, changing the fields shown in step 7. The server indicates the data it is sending with the `data_type`, `data_length`, and `data_ptr` fields. It cannot send more data than the client requests (this value must be determined by the application, either by convention or by information in the control message). The server sends a control message appropriate to the application. It specifies a completion mailbox to receive the signal that the transaction is complete.

The hardware and transport software receive the buffer request, initiate a buffer grant specifying the location of the client's (response block) data buffer, and transfer the data into the client's buffer. When the data is transferred, a completion message is sent to the client and server mailboxes.

8 The client waits at the completion mailbox specified in the response block. The received message points to the original (double-block) message structure. In the response block, the client gets the control message and checks `trans_control` (for KN_RESPONSE), `type` (for KN_RECEIVE_COMPLETE), and `data_length`. The returned `data_length` indicates the actual amount of data returned, which may be less than the `data_length` originally specified in the step 3 response block.

9 At its completion mailbox, the server receives a pointer to the message sent in step 7. The server checks the `type` (for KN_SEND_COMPLETE) and `data_status` fields.

10 The server waits for another message at the port mailbox.

Fragmentation

In steps 5 and 6, the received data could be fragmented as described in Example E.

In steps 7 and 9, the returned data could be fragmented as described in Example D.

Getting the Status of Received Messages

Depending on the circumstances, the status of a message can be indicated by the value returned directly from a **send_tp** system call or in the following fields of a received message structure:

- `type`
- `trans_control`
- `data_status`

In the description of the **send_tp** system call, the *iRMK™ Kernel Reference Manual* lists the values that can be received in these fields.

***

STANDARD I/O AND THE C LIBRARIES 8

Determining Which Library Functions Are Called

The *System V/iRMK C Libraries* manual describes C library functions you can use when programming in C. One of the features of using a C library is that you can perform I/O functions on the System V host from an application on a real-time host. This process involves using the C Server daemon running on the System V host. All C library I/O functions operate as if they were invoked from a program running under System V. For example, a `read` reads from the System V file server, and a `printf` writes to a terminal attached to the System V host. Non-I/O functions, such as `malloc` or `strcmp`, operate on the real-time host, making calls to the Kernel if necessary.

The Kernel also provides the following I/O functions, which read from or write to a terminal attached to the real-time host:

- The **co**, **ci**, and **csts** system calls perform character I/O.
- The *kstdios.lib* and *kstdioc.lib* libraries provide the standard I/O functions `printf`, `scanf`, `putchar`, and `getchar`.

You may want to use the features of the C Libraries, but call functions from a *kstdio* library as well. However, if you have a `printf` in the program, you need to determine whether it is a C library `printf` or a *kstdio* `printf`. In one case it would operate on the System V terminal; in the other, it would operate on the real-time host's terminal. The solution is the order in which you include header files in your code.

Figure 8-1 illustrates three tasks using the `printf` function. The process shown in Figure 8-1 applies for any of the *kstdio* functions. The difference between the tasks is as follows:

- Task A includes only the *kstdio.h* header file, found in directory */usr/intel/rmk/system/inc*. Its `printf` goes to the real-time host's terminal.
- Task B includes only the C library *stdio.h* header file, found in directory */usr/intel/rmk/system/include*. Its `printf` goes to the System V host's terminal.
- Task C includes both the *kstdio.h* and the *stdio.h* header files, with *kstdio.h* first. Its `printf`, and any of the four *kstdio* functions, operate on the real-time host's terminal. All other I/O functions defined in the C libraries operate on the System V host. If the header files were included in the opposite order, the *kstdio* functions would never be called.

The appropriate libraries must be bound with the application, but the order in which they are bound has no effect on which library is called.

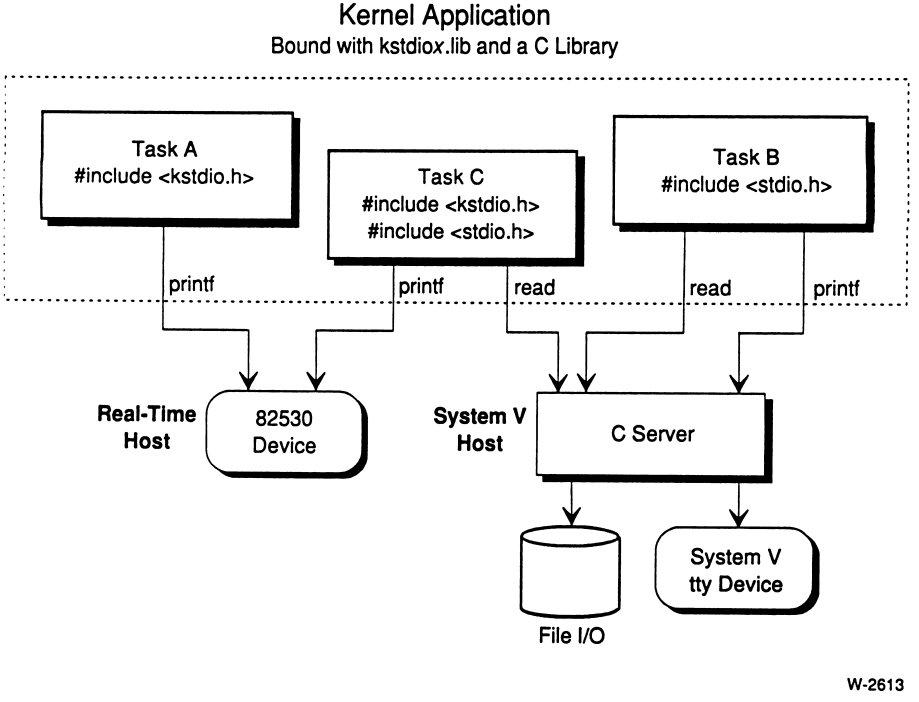


Figure 8-1. Using C Library and kstdio Functions

Establishing a Terminal Device for the C Libraries

Functions such as `printf` operate on whatever terminal device is configured for the C libraries. You configure the device in an **initialize_clib** call, which is described in the *System V/iRMK C Libraries* manual. In the call, you pass a configuration structure that contains, among other items, a name of a System V `tty` port. All `stdin`, `stdout`, and `stderr` streams generated by C library functions go to that port.

Establishing a Terminal Device for Kernel I/O

The Kernel character I/O and *kstdio* functions operate on the 82530 device described in Chapter 7. This might be on an iSBX 354 module attached to the real-time host or a serial device on the base processor board itself. You determine the location of the device in the **initialize_console** system call, described in the *iRMK™ Kernel Reference Manual*. In a structure passed in the call, you specify:

- the I/O port addresses of the data and control ports
- the clock frequency in hertz
- the baud rate

Table 8-1 gives these values for some Intel boards. For other boards, refer to the appropriate hardware reference manual.

Table 8-1. Initialization Values for the 82530 Device

Board	Data Port	Cntrl Port	Frequency (hz)	Baud
iSBC 386/133			9830400 (960000h)	300-38400 in increments
iSBC 486/125			4915200 (4B0000h)	300-38400 in increments
Connector J1	0DEh	0DCh		
Connector J2	0DAh	0D8h		
iSBX 354, *on Intel boards			4915200 (4B0000h)	300-19200 in increments
Connector J1	86h	84h		
Connector J2	82h	80h		

* Intel boards use 80H as the base I/O port address for the iSBX connector. Other boards may use a different address, so that an iSBX 354 module attached to those boards would have different port addresses.

***

DIFFERENCES BETWEEN RELEASE I.2 AND RELEASE I.3 A

Release I.3 of the System V/iRMK Kernel offers the following features not in System V/iRMK Release 1.2:

- OMF-based development, including support for the full spectrum of Intel programming languages
- Support for the Soft-Scope III debugger
- Kernel-aware debugging support
- Gate-based interfaces for segmented applications
- C library support
- Third-party software support
- Additional MSA utilities for bootloading boards
- A wider variety of tested example programs

Also, as described in Appendix B of the *iRMK™ Kernel Reference Manual*, stacks for internal use by the Kernel may now reside in the Kernel's data segment. This allows the application to use the flat programming model as well as other models.

To use this release, an existing application must be moved to a different set of development tools, as described in this appendix.

Benefits to Existing Applications

Three of the major benefits provided in Release I.3 are improvements in:

- Debugger support
- Handler interfaces
- Languages and segmentation models supported

Debugger Support

Release I.2 was supported by an enhanced version of the *sdb* debugger, called *rdb*. This is no longer supported. Instead, Soft-Scope III provides the debugging interface. Table A-1 compares Soft-Scope III with *rdb*.

Table A-1. Comparison of Soft-Scope III Debugger with rdb Debugger

Soft-Scope III debugger	rdb debugger
Kernel object awareness	no
Static mode debugging only	tasking mode
Ability to debug interrupt handlers, alarm handlers, etc.	no
All break points are global	no
Works with all models	no
Initialization code can be debugged	no
Low level machine details (TSS, GDT, memory, etc.) can be examined	no

With the new debugging support, the host-target link can be over Multibus II or a serial link. Previously, only Multibus II message-passing could be used.

Handler Interfaces

Release I.2 required special code by handlers, such as alarm and task handlers. The Kernel was placed in a different segment, and user handlers had to make far returns. Since this is not possible in small model (the only model supported), special code needed to be added to the handlers. Release I.3 does not require such special in-line assembly code in handlers. The interrupt handlers still should be defined as far procedures. Figure A-1 shows an example handler with the in-line assembler code required in Release I.2. Figure A-2 shows how the same handler can be written in Release I.3.

```
alarm_hdlr(alarm_ptr)
int *alarm_ptr;
{
        KN_STATUS status;
            /* alarm handler prologue - since the alarm handler is a far
            procedure for the kernel,
                set up DS
                save kernel DS */
        asm ("push %ds");
        asm ("movw $0x248,%ax");
        asm ("movw %ax,%ds");
            /* alarm handler user code - remember that for some reason the coff
            interface library does not restore DS,  so save and restore DS after
            every kernel call.  Local structures cannot be passed to kernel
            calls since the coff interface library assumes that DS and SS are
            interchangeable, but they are not (inside handlers).  */
        alarm_hdlr_count ++;
        asm ("push %ds");
        KN_send_unit(alarm_sem);
        asm ("pop %ds");
        asm ("push %ds");
        status = KN_send_data(alarm_mbx, snd_data, 8);
        asm ("pop %ds");
        if (status != E_OK)
                alarm_error_count ++;
            /* alarm handler epilogue -
                restore kernel DS
                do a far return */
        asm ("pop %ds");
        asm("leave");
        asm("lret");
}
```

Figure A-1. Alarm Handler for Release I.2

```
alarm_hdlr(alarm_ptr)
int *alarm_ptr;
{
        KN_STATUS status;
        alarm_hdlr_count ++;
        KN_send_unit(alarm_sem);
        status = KN_send_data(alarm_mbx, snd_data, 8);
        if (status != E_OK)
                alarm_error_count ++;
}
```

Figure A-2. Alarm Handler for Release I.3

Language and Model Support

In Release I.2, an application could be compiled only with the UNIX *cc* compiler. Only small model applications were supported.

In Release I.3, the application can be developed in Intel iC-386, PL/M-386, FORTRAN-386, ASM386, MetaWare High C, Phar Lap Assembler or any compatible compiler. All segmentation models are supported. In addition, there is an interface for making intersegment calls through gates. For information about using the gate-based interface, refer to page 4-36 in this manual.

Moving Applications to Release I.3

In Release I.2, applications were built using tools that produced code in Common Object File Format (COFF). The UNIX *cc* compiler was used to compile the application. The output was translated to OMF and built with the Kernel.

Applications for Release I.3 must be OMF386 format. The *cc* compiler cannot be used. This brings the application up-to-date with current Kernel development and positions it for future support.

To make this change, choose one of the OMF-supporting programming languages described in Chapter 4. If the application is written in C or assembler, you have the choice of Intel or third-party tools. For an application in FORTRAN or PL/M, Intel provides compilers. Familiarize yourself with the requirements of the compiler you choose.

Review the programming information in Chapter 4. Changes may be needed in your application, but unless you decide to change the segmentation model, these changes should be minor. Check the list of include files to make sure they're in your application. Remove the assembler interface code required in handlers for Release I.2. One Kernel call must be added for the new debugger support: **initialize_RDS**, which is described in the *iRMK™ Kernel Reference Manual.*

Recompile the application. Depending on the compiler used, build the application using either BND386 and BLD386 or Phar Lap's LinkLoc utility. Use the example bind and build files as a guide. Make sure to build debugger components with the application, as described in Chapter 4.

***

PUTTING CODE IN ROM B

You may want to create a system with an embedded Kernel application that does not use a System V host. Since the Kernel does not include file I/O features, such an application must be loaded from ROM on the individual boards in the system.

The first step is to thoroughly debug the application using the System V host for development. After code is placed into ROM, it cannot be debugged with Soft-Scope III, because the iM III Monitor shipped with the Kernel is bootloaded, not ROM-based. Probably the best debugging system for ROM-based code is an in-circuit emulator.

After the bootloadable application is debugged, the application must be re-compiled to put the program constants in the code segment so that these values are included in ROM. With Intel compilers, this is done by using the `rom` control. An alternative is to program both the code and data segments into ROM and copy the data segment to RAM during initialization.

The application must be rebuilt with additional startup code that initializes the processor, sets up descriptor tables and task state information, and jumps to the initial application task procedure. If the application should run from RAM rather than ROM (for processing speed), this startup code would also move the application code into RAM before jumping to the application. The application is built to place a jump instruction to this startup code at the top of ROM memory. Then the entire application is put into EPROM and installed on the board.

The Kernel software includes PL/M and C examples that are built to run from ROM. To see the characteristics of a ROM-based program, refer to the files in the */usr/intel/rmk/examples/ic386/rom_ex.cpt/src* and */usr/intel/rmk/examples/plm/rom_ex.cpt/src* directories. When these examples are built, a jump from high ROM memory to the startup code is created with the following BLD386 control:

```
BOOTSTRAP (startup)
```

The startup code is an assembly-language routine that is not part of the application until it is put in ROM. It must be written as a 16-bit program because (except for the 376 embedded processor) Intel386 family processors start in real mode rather than protected mode. The startup code performs the following functions:

- sets up a temporary GDT
- puts the processor in protected mode
- copies the Builder-created GDT and IDT from ROM to RAM
- modifies the GDT and IDT aliases (in the GDT) to refer to their new location in RAM (by default, the Kernel expects these aliases to be in slots 1 and 2 of the GDT, but the slots used can be configured with the Kernel)
- loads the GDTR and IDTR registers with the locations of the RAM-based GDT and IDT
- if LDTs are used, moves the LDTs to RAM and adjusts the GDT descriptors for the LDTs
- moves the TSS segment(s) from ROM to RAM and adjusts the GDT descriptor(s) for the TSS(s)
- loads the application code (which includes the Kernel) into RAM (optional)
- adjusts the base of the code descriptor in the GDT to point to the RAM-based code (if the code is put into RAM)
- loads the Task Register (TR) with the location of the initial TSS
- loads the processor registers using values from the initial TSS
- sets up the stack so that the start address and flags are on the stack
- performs an IRET instruction (interrupt return) that jumps to the initial application task, which proceeds to initialize the Kernel

For information about compiling code to be placed in ROM, refer to the compiler documentation. Some C compilers expect their output to be placed into memory by a loader rather than residing permanently in ROM. These compilers assume that initial data resides in the data segment and is initialized by the loader. When using such a compiler, you must add code to an assembler startup module that defines initial values in the code segment and transfers them to the data segment. You need not do this with the iC-386 compiler, because it provides the `rom` control.

See also: *Intel 386 Family System Builder User's Guide*
386™ DX Microprocessor Programmer's Reference Manual

The object code produced with the compiler is OMF format. This must be converted to a format understandable by your EPROM programmer (such as a hex or binary format).

Intel boards typically divide ROM memory into even and odd bytes, using two EPROMs. The ROM-based application must be separated into even and odd bytes when programming the EPROMs.

It is beyond the scope of this manual to describe programming EPROMs; refer to the documentation for your EPROM programmer.

***

BOOTING SYSTEM V WITH A BAD CONFIGURATION FILE C

When editing the bootstrap configuration file to bootload a Kernel application, it is possible to corrupt the file so that System V/386 cannot boot. To recover from this, you can use the MSA Master Test Handler to enter all the System V-related BPS parameters that were in the file to begin with. An alternative method is to use the Master Test Handler to boot from a backup copy of the configuration file that is known to work.

This appendix describes rebooting both with and without a backup copy of the configuration file. Because Release 3.2 of System V/386 uses different pathnames for some files than Release 4.0, instructions are shown for both releases. After rebooting System V using one of the methods shown here, repair the damage to the bootstrap configuration file so you can reboot the real-time host(s).

The instructions assume the following:

- The system previously booted System V correctly, but after editing the *config* file you attempted to boot the system and System V won't boot.
- The system is not operating and no processes on any boards need to be shut down.

For more detailed information about MSA and using the Master Test Handler, refer to the *Firmware User's Guide for MSA Firmware.*

Some of the instructions discuss invoking the *bootserver* daemon. In release 3.2 of System V, the bootserver daemon is in the */etc* directory. In release 4.0, the bootserver daemon is in the */sbin* directory. The bootserver is described in the *Intel® System V/386 Multibus Reference Manual.*

Rebooting with a Backup File

The following instructions assume that you have on the hard disk a *config.orig* file, which is a backup copy of a *config* file that correctly boots the System V host and the file server (PCI) host. The backup file could have any name you choose, but it must be in the root file system. The instructions assume that *config.orig* is in the same directory as the *config* file. The directory is dependent on whether you use System V 3.2 or 4.0. Follow the instructions for the release of System V you use.

Booting System V 3.2 with a Backup File

1. Reset or power-up the system.
2. When you see asterisks or other characters displayed on the screen, press

   <Shift-U>

   repeatedly. The upper-case U will not show on the screen.
3. In a few seconds, the system prompts you for a selection from the following menu:

```
Choose one of the following:

1 Run System Diagnostics
2 Go to Operator Interface
3 Go to Bootphase

Enter number   ?
```

   Select "Run System Diagnostics" by typing 1 at the "Enter number ?" prompt. If you do not make a choice within a few seconds, the system defaults to running the diagnostics (selection 1).
4. The system runs diagnostic tests and displays the results. If some of the test results do not display "PASS", the system may have a hardware problem that is causing it not to boot. Fix the hardware problem before attempting to boot the system.
5. At the end of the diagnostic tests, a prompt similar to the following is displayed:

```
Do you want to enter test commands?  Enter "y" or "n" [n]
```

   Type **Y** within five seconds to invoke the Master Test Handler; otherwise the system enters the boot phase. If the system does enter the boot phase, repeat steps 1 through 3, but in step 3 select **2** (Go to Operator Interface).
6. The Master Test Handler prompt `MTH [0]>` is displayed. The number in the brackets may not be `0`. The number indicates the slot holding the board selected as the master in the system. At the prompt, enter the following series of commands (printed in **bold**). In the **S** *x* command, replace *x* with the slot number of the PCI file server host (typically an iSBC 386/258 board).

```
MTH [Ø]> S x <CR>
MTH [x]> mp <CR>
   Modify Boot Parameters for slot   x:
Modify parameters: "#" deletes, "." quits, <CR> advances
new parameter
BL_config_file = /etc/default/bootserver/config.orig <CR>
new parameter
<CR>
save changes ([y] or n)
<CR>
```

7. Now make the same sort of entries for the System V host. In the **S *z*** command, replace ***z*** with the slot number of the System V host.

```
MTH [x]> S z <CR>
MTH [z]> mp <CR>
   Modify Boot Parameters for slot   z:
Modify parameters: "#" deletes, "." quits, <CR> advances
new parameter
BL_config_file = /etc/default/bootserver/config.orig <CR>
new parameter
<CR>
save changes ([y] or n)
<CR>
```

8. Enter the following command to invoke the boot phase:

```
MTH [z]> B <CR>
```

9. System V should boot. Login to the system as *root*. Enter:

```
uname -a
```

10. If the version of the operating system is shown as

```
3.2 2.1
```

continue with steps 11 and 12. If any other version string is shown, you may ignore steps 11 and 12, and proceed to fix the *config* file.

11. Find the process ID (PID) of the bootserver:

```
ps -ef | grep bootserver
```

12. As the root user, kill and restart the bootserver process.

```
kill -9 bootserver_PID                    (specify PID found in step 11)
bootserver -c /etc/default/bootserver/config.orig -l /tmp/boot.log
```

The resulting */tmp/boot.log* file contains information about the boot process.

Booting System V 4.0 with a Backup File

1. Reset or power-up the system.
2. When you see asterisks or other characters displayed on the screen, press

   <Shift-U>

   repeatedly. The upper-case U will not show on the screen.
3. In a few seconds, the system prompts you for a selection from the following menu:

```
Choose one of the following:

1 Run System Diagnostics
2 Go to Operator Interface
3 Go to Bootphase

Enter number   ?
```

   Select "Run System Diagnostics" by typing 1 at the "Enter number ?" prompt. If you do not make a choice within a few seconds, the system defaults to running the diagnostics (selection 1).
4. The system runs diagnostic tests and displays the results. If some of the test results do not display "PASS", the system may have a hardware problem that is causing it not to boot. Fix the hardware problem before attempting to boot the system.
5. At the end of the diagnostic tests, a prompt similar to the following is displayed:

```
Do you want to enter test commands?  Enter "y" or "n" [n]
```

   Type **Y** within five seconds to invoke the Master Test Handler; otherwise the system enters the boot phase. If the system does enter the boot phase, repeat steps 1 through 3, but in step 3 select **2** (Go to Operator Interface).

6. The Master Test Handler prompt `MTH [Ø]>` is displayed. The number in the brackets may not be Ø. The number indicates the slot holding the board selected as the master in the system. At the prompt, enter the following series of commands (printed in **bold**). In the **S** *x* command, replace *x* with the slot number of the PCI file server host (typically an iSBC 386/258 board).

```
MTH [Ø]> S x <CR>
MTH [x]> mp <CR>
   Modify Boot Parameters for slot   x:
Modify parameters: "#" deletes, "." quits, <CR> advances
new parameter
BL_config_file = /stand/config.orig <CR>
new parameter
<CR>
save changes ([y] or n)
<CR>
```

7. Now make the same sort of entries for the System V host. In the **S** *z* command, replace *z* with the slot number of the System V host.

```
MTH [x]> S z <CR>
MTH [z]> mp <CR>
   Modify Boot Parameters for slot   z:
Modify parameters: "#" deletes, "." quits, <CR> advances
new parameter
BL_config_file = /stand/config.orig <CR>
new parameter
<CR>
save changes ([y] or n)
<CR>
```

8. Enter the following command to invoke the boot phase:

```
MTH [z]> B <CR>
```

9. System V should boot.

Rebooting without a Backup File

If you did not make a backup *config* file, you can boot the system with the Master Test Handler by entering all the BPS parameters required for a given board. Follow the instructions for the release of System V you use.

Booting System V 3.2 without a Backup File

1. Reset or power-up the system.
2. When you see asterisks or other characters displayed on the screen, press

   <Shift-U>

   repeatedly. The upper-case U will not show on the screen.
3. In a few seconds, the system prompts you for a selection from the following menu:

```
Choose one of the following:

1 Run System Diagnostics
2 Go to Operator Interface
3 Go to Bootphase

Enter number   ?
```

   Select "Run System Diagnostics" by typing 1 at the "Enter number ?" prompt. If you do not make a choice within a few seconds, the system defaults to running the diagnostics (selection 1).
4. The system runs diagnostic tests and displays the results. If some of the test results do not display "PASS", the system may have a hardware problem that is causing it not to boot. Fix the hardware problem before attempting to boot the system.
5. At the end of the diagnostic tests, a prompt similar to the following is displayed:

```
Do you want to enter test commands?  Enter "y" or "n" [n]
```

   Type **Y** within five seconds to invoke the Master Test Handler; otherwise the system enters the boot phase. If the system does enter the boot phase, repeat steps 1 through 3, but in step 3 select **2** (Go to Operator Interface).

6. The Master Test Handler prompt `MTH [Ø]>` is displayed. The number in the brackets may not be Ø. The number indicates the slot holding the board selected as the master in the system. At the prompt, enter the following series of commands (printed in **bold**). In the **S** *x* command, replace *x* with the slot number of the PCI file server host. This must be an iSBC 386/258 board to use the `BL_Target_file` parameter shown below.

```
MTH [Ø]> S x <CR>
MTH [x]> mp <CR>
   Modify Boot Parameters for slot   x:
Modify parameters: "#" deletes, "." quits, <CR> advances
new parameter
BL_OI_Master = slot<CR>                    (Specify slot of System V host)
new parameter
BL_Target_file = /etc/default/bootserver/pci258<CR>
new parameter
BL_mode = p<CR>
new parameter
BL_config_file = /junk <CR>                (/junk is any non-existent file)
new parameter
<CR>
save changes ([y] or n)
<CR>
```

7. Now make the same sort of entries for the System V host. In the **S** *z* command, replace *z* with the slot number of the System V host.

```
MTH [x]> S z <CR>
MTH [z]> mp <CR>
   Modify Boot Parameters for slot   z:
Modify parameters: "#" deletes, "." quits, <CR> advances
new parameter
BL_method = Quasi<CR>
new parameter
BL_Target_file = /unix<CR>
new parameter
BL_config_file = /junk <CR>                (/junk is any non-existent file)
new parameter
<CR>
save changes ([y] or n)
<CR>
```

8. Enter the following command to invoke the boot phase:

```
MTH [z]> B <CR>
```

9. System V should boot. Login to the system as *root*. Enter:

   ```
   uname -a
   ```

10. If the version of the operating system is shown as

    ```
    3.2 2.1
    ```

    continue with steps 11 and 12. If any other version string is shown, you may ignore steps 11 and 12, and proceed to fix the *config* file.

11. Find the process ID (PID) of the bootserver:

    ```
    ps -ef | grep bootserver
    ```

12. As the root user, kill and restart the bootserver process.

    ```
    kill -9 bootserver_PID                    (specify PID found in step 11)
    bootserver -c /etc/default/bootserver/config.orig -l /tmp/boot.log
    ```

The resulting */tmp/boot.log* file contains information about the boot process.

Booting System V 4.0 without a Backup File

1. Reset or power-up the system.
2. When you see asterisks or other characters displayed on the screen, press

   <Shift-U>

   repeatedly. The upper-case U will not show on the screen.
3. In a few seconds, the system prompts you for a selection from the following menu:

```
Choose one of the following:

1 Run System Diagnostics
2 Go to Operator Interface
3 Go to Bootphase

Enter number   ?
```

   Select "Run System Diagnostics" by typing 1 at the "Enter number ?" prompt. If you do not make a choice within a few seconds, the system defaults to running the diagnostics (selection 1).
4. The system runs diagnostic tests and displays the results. If some of the test results do not display "PASS", the system may have a hardware problem that is causing it not to boot. Fix the hardware problem before attempting to boot the system.
5. At the end of the diagnostic tests, a prompt similar to the following is displayed:

```
Do you want to enter test commands?  Enter "y" or "n" [n]
```

   Type **Y** within five seconds to invoke the Master Test Handler; otherwise the system enters the boot phase. If the system does enter the boot phase, repeat steps 1 through 3, but in step 3 select **2** (Go to Operator Interface).

6. The Master Test Handler prompt `MTH [Ø]>` is displayed. The number in the brackets may not be Ø. The number indicates the slot holding the board selected as the master in the system. At the prompt, enter the following series of commands (printed in **bold**). In the **S** *x* command, replace *x* with the slot number of the PCI file server host. This must be an iSBC 386/258 board to use the `BL_Target_file` parameter shown below.

```
MTH [Ø]> S x <CR>
MTH [x]> mp <CR>
   Modify Boot Parameters for slot   x:
Modify parameters: "#" deletes, "." quits, <CR> advances
new parameter
BL_QI_Master = slot<CR>                    (Specify slot of System V host)
new parameter
BL_Target_file = /stand/pci258<CR>
new parameter
BL_mode = p<CR>
new parameter
BL_config_file = /junk <CR>                (/junk is any non-existent file)
new parameter
<CR>
save changes ([y] or n)
<CR>
```

7. Now make the same sort of entries for the System V host. In the **S** *z* command, replace *z* with the slot number of the System V host.

```
MTH [x]> S z <CR>
MTH [z]> mp <CR>
   Modify Boot Parameters for slot   z:
Modify parameters: "#" deletes, "." quits, <CR> advances
new parameter
BL_method = Quasi<CR>
new parameter
BL_Target_file = /unix<CR>
new parameter
BL_config_file = /junk <CR>                (/junk is any non-existent file)
new parameter
<CR>
save changes ([y] or n)
<CR>
```

8. Enter the following command to invoke the boot phase:

   MTH [*z*]> **B <CR>**

9. System V should boot.

***

SYNTAX FOR INVOKING INTEL® TOOLS D

Differences in Syntax

The manuals you receive with Intel compilers and utilities show how to invoke these tools in the MS-DOS environment. The invocation syntax is slightly different under System V, where you use the UNXUDI utility as the interface to the tools. The differences in syntax are because:

- System V is case-sensitive, while MS-DOS is not.
- System V interprets parentheses entered on the command line, while MS-DOS does not.
- System V expects command-line switches immediately after the command and before any referenced file(s), while MS-DOS expects command-line switches at the end of the command string. Under System V, command-line switches are preceded with a dash (-).

Case Sensitivity

With one exception, use lower-case for compiler and utility names and controls entered at the command line. The exception is the CONTROLFILE (or CF) control used by BND386 and BLD386. When entered at the command line, the CF control must be upper-case. It is also the only control that is not preceded with a dash when used on the command line. The following section shows an example of using the CF control.

Escaping Parentheses

Some invocations require parentheses on the command line. You must escape the parentheses using quotes (" or ') or a backslash (\). An example is the invocation for the Binder when specifying a control file (note that CF must be upper-case):

```
bnd386  CF '(control.bnd)'
```

Command-Line Switches

When invoking compilers and utilities, controls on the command line are considered command line switches. The controls must be preceded with a dash (-) and come before the file being operated on. For example, the following line invokes the iC-368 compiler on file *test.c* with the small ROM segmentation model, specifying the listing file with the `print` control, and using the `debug` control.

```
ic386 -small -rom -pr myfile.lst -db test.c
```

Contrast this with the MS-DOS invocation listed below:

```
ic386 test.c small rom pr (myfile.lst) db
```

Summary of Invocations

When you invoke one of the Intel utilities under System V, you are actually invoking UNXUDI, which passes the command string to the appropriate executable file. See the following scripts installed in the */usr/intel/bin* directory:

asm386	bld386
ftn386	bnd386
ic386	lib386
plm386	map386

The general syntax for invoking compilers and the assembler is:

```
compiler-name [-controls] filename
```

To invoke LIB386, use the following:

```
lib386 [-batch | -nobatch][-backup | -nobackup] filename
```

To invoke MAP386, BND386 or BLD386, put the controls in a control file and invoke the utility with the general syntax shown below:

```
utility-name CF '(filename)'
```

The controls in control files can be entered exactly as shown in the *Intel386 Family Utilities User's Guide* and the *Intel386 Family System Builder User's Guide.*

***

INDEX

C (continued)

D

H (continued)

I

L (continued)

M

M (continued)

N

O

P

P (continued)

Q

R

R (continued)

S

S (continued)

U

V

W

Request For Reader's Comments

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

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

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

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

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

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

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

Name_____Date_____
Title_____
Company Name/Department_____
Address_____
City_____State_____Zipcode_____
(Country)_____Phone_____

Please check here if you require a written reply ☐

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL
FIRST CLASS MAIL PERMIT NO. 79 HILLSBORO OR

POSTAGE WILL BE PAID BY ADDRESSEE

ICD TECHNICAL PUBLICATIONS HF3-72
INTEL CORPORATION
5200 NE ELAM YOUNG PARKWAY
HILLSBORO OR 97124-9978

Please fold here and close the card with tape. Do not staple.

WE'D LIKE YOUR COMMENTS....

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

If you are in the United States and are sending only this card, postage is prepaid.

If you are sending additional material or if you are outside the United States, please insert this card and any enclosures in an envelope. Send the envelope to the above address, adding "United States of America" if you are outside the United States.

Thanks for your comments.

International Sales Offices

AUSTRALIA
Intel Australia Pty. Ltd.
Unit 13
Allambie Grove Business Park
25 Frenchs Forest Road East
Frenchs Forest, NSW 2086

BRAZIL
Intel Semicondutores do Brazil LTDA
Av. Paulista, 1159-CJS 404/405
01311 - Sao Paulo - S.P.

CANADA
Intel Semiconductor of Canada, Ltd.
4585 Canada Way, Suite 202
Burnaby V5G 4L6
British Columbia

Intel Semiconductor of Canada, Ltd.
2650 Queensview Drive
Suite 250
Ottawa K2B 8H6
Ontario

Intel Semiconductor of Canada, Ltd.
190 Attwell Drive
Suite 500
Rexdale M9W 6H8
Ontario

Intel Semiconductor of Canada, Ltd.
620 St. Jean Boulevard
Pointe Claire H9R 3K2
Quebec

CHINA/HONG KONG
Intel PRC Corporation
15/F, Office 1, Citic Bldg.
Jian Guo Men Wai Street
Beijing, PRC

Intel Semiconductor Ltd.
10/F East Tower
Bond Center
Queensway, Central
Hong Kong

DENMARK
Intel Denmark A/S
Glentevej 61, 3rd Floor
2400 Copenhagen NV

FINLAND
Intel Finland OY
Ruosilantie 2
00390 Helsinki

FRANCE
Intel Corporation S.A.R.L.
1, Rue Edison-BP 303
78054 St. Quentin-en-Yvelines
Cedex

WEST GERMANY
Intel Semiconductor GmbH
Dornacher Strasse 1
8016 Feldkirchen bei Muenchen

Intel Semiconductor GmbH
Hohenzollern Strasse 5
3000 Hannover 1

Intel Semiconductor GmbH
Abraham Lincoln Strasse 16-18
6200 Wiesbaden

Intel Semiconductor GmbH
Zettachring 10A
7000 Stuttgart 80

INDIA
Intel Asia Electronics, Inc.
4/2, Samrah Plaza
St. Mark's Road
Bangalore 560001

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

ITALY
Intel Corporation Italia S.p.A.
Milanofiori Palazzo E
20090 Assago
Milano

JAPAN
Intel Japan K.K.
5-6 Tokodai, Tsukuba-shi
Ibaraki, 300-26

Intel Japan K.K.
Daiichi Mitsugi Bldg.
1-8889 Fuchu-cho
Fuchu-shi, Tokyo 183

Intel Japan K.K.
Bldg. Kumagaya
2-69 Hon-cho
Kumagaya-shi, Saitama 360

Intel Japan K.K.
Kawaasa Bldg., 8-9F
2-11-5, Shinyokohama
Kohoku-ku, Yokohama-shi
Kanagawa, 222

Intel Japan K.K.
Ryokuchi-Eki Bldg.
2-4-1 Terauchi
Toyonaka-shi, Osaka 560

Intel Japan K.K.
Shinmaru Bldg.
1-5-1 Marunouchi
Chiyoda-ku, Tokyo 100

Intel Japan K.K.
Green Bldg.
1-16-20 Nishiki
Naka-ku, Nagoya-shi
Aichi 450

KOREA
Intel Technology Asia, Ltd.
16th Floor, Life Bldg.
61 Yoido-Dong, Youngdeungpo-Ku
Seoul 150-010

NETHERLANDS
Intel Semiconductor B.V.
Postbus 84130
3099 CC Rotterdam

NORWAY
Intel Norway A/S
Hvamveien 4-PO Box 92
2013 Skjetten

SINGAPORE
Intel Singapore Technology, Ltd.
101 Thomson Road #21-05/06
United Square
Singapore 1130

SPAIN
Intel Iberia S.A.
Zurbaran, 28
28010 Madrid

SWEDEN
Intel Sweden A.B.
Dalvagen 24
171 36 Solna

SWITZERLAND
Intel Semiconductor A.G.
Zuerichstrasse
8185 Winkel-Rueti bei Zuerich

TAIWAN
Intel Technology Far East Ltd.
8th Floor, No. 205
Bank Tower Bldg.
Tung Hua N. Road
Taipei

UNITED KINGDOM
Intel Corporation (U.K.) Ltd.
Pipers Way
Swindon, Wiltshire SN3 1RJ