intel®

iRMK™ Kernel Reference Manual

Order Number: 467231-001

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

In the United States, additional copies of this manual or other Intel literature 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. Phar 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

Reader Level

This manual describes the features, concepts, and usage of the iRMK™ I.3 Real-time Kernel. It provides technical information necessary to develop Kernel applications. The manual assumes the reader's familiarity with the following:

- Intel386™ family microprocessors including the 486™ processor and the 376™ embedded processor
- C, ASM386, FORTRAN, and/or PLM-386 programming languages as required for the user's application
- Real-time operating system programming concepts
- Multibus II hardware, if an application uses message passing

Manual Organization

This manual consists of five chapters and four appendices:

- Chapter 1 provides an explanation of the syntax, data types, and description headings used in the Kernel system calls descriptions.
- Chapter 2 provides a detailed, alphabetically arranged, description of each Kernel system call. The chapter includes a cross-reference chart listing the system calls by function.
- Chapter 3 describes the stdio library functions supplied with the Kernel and provides programming models showing how to include the stdio library in various application formats. This chapter provides quick reference material for the stdio functions provided by the Kernel.
- Chapter 4 describes interfaces for handlers that you may supply to the Kernel to perform additional operating system functions.
- Chapter 5 describes configuration for the Kernel.
- Appendix A lists the exception codes that can be received while running a Kernel application.

- Appendix B discusses the stack requirements of application tasks. It also discusses the stack requirements for the Kernel's internal tasks, and how to set up the internal stacks.
- Appendix C provides information on making Kernel system calls using the assembly language interface.
- Appendix D lists the conditions for using the address translation mechanisms, modifying alias selectors for GDT and IDT slots, modifying the I/O permission bit map for a task, and collected notes on using the 82380 and 82370 devices.

Conventions

In this manual, the term "Kernel" refers to the iRMK Kernel.

The KN_ prefix begins all Kernel system calls when called from C or PL/M, except **initialize_stdio**. When the Kernel primitives are called from ASM, the KNA_ prefix must be used. When referring to the system calls in text, this manual uses a shorthand notation and omits the prefix. For example, **KN_create_task** is written as **create_task**. The prefix is included when listing code sequences.

All Kernel system calls referenced within discussions are printed in **bold** type.

Related Publications

The following documents may be of interest to you. They are available through your local Intel sales office:

- *386™ DX Microprocessor Programmer's Reference Manual*
  order number 230985
- *386™ SX Microprocessor Programmer's Reference Manual*
  order number 240331
- *80386 System Software Writer's Guide*
  order number 231499
- *ASM386 Assembler Language Reference Manual*
  order number 480251
- *iC-386 Compiler User's Guide*
  order number 483326
- *C: A Reference Manual*
  order number 555107
- *PL/M Programmer's Guide*
  order number 452161

- *FORTRAN-386 Compiler User's Guide*
  order number 481837
- *Intel386™ Family System Builder User's Guide*
  order number 481342
- *Intel386™ Family Utilities User's Guide*
  order number 481343
- *Microprocessors Handbook*
  order number 230843
- *Peripherals Handbook*
  order number 296467
- *Firmware User's Guide for Multibus II System Architecture (MSA) Firmware*
  order number 506090
- *Multibus II Interconnect Interface Specification*
  order number 149299
- *Multibus II Transport Protocol Specification and Designer's Guide*
  order number 453508
- *IEEE 1296 Specification*
  order number 281065
- *MPC User's Guide*
  order number 176526
- *Debugging System V/iRMK™ Applications*
  order number 467236
- *System V/iRMK Installation and User's Guide*
  order number 467241
- *System V/iRMK™ C Libraries*
  order number 467226
- *Intel® System V/386 Multibus Reference Manual*
  order number 463328
- *Soft-Scope® III Debugger User's Guide*
  Target: 386™ with System V/iRMK™ Kernel
  Host: System V/386
  order number 467246

Additional publications for products that may be used with the Kernel can be obtained from the following companies:

- *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

- 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. Introduction

Chapter 2. Kernel System Calls

Chapter 2. Kernel System Calls (continued)

Chapter 2. Kernel System Calls (continued)

Chapter 3. Standard Input And Output Functions

Chapter 4. Kernel Handlers

Chapter 5. Configuration And Initialization

Appendix A. Exception Codes

Appendix B. Stack Requirements

Appendix C. Assembly Language Interfaces to the Kernel

Appendix D. Application Notes

Index

Service Information ..... Inside Back Cover

Tables

Figures

OVERVIEW 1

This manual provides detailed descriptions of all the Kernel system calls. The system calls are organized alphabetically in Chapter 2. Chapter 3 describes the Kernel standard I/O functions. Chapter 4 describes the handler interfaces you can use to supply additional operating system functions. For background information about these calls, refer to the *Installation and User's Guide*.

Syntax

Throughout this manual, the system calls, data structures, and data types are specified using the C language syntax. If you write your programs in C, you can access the system calls using this syntax.

The Kernel also provides support for PL/M, FORTRAN, and assembly language programs. The PL/M interface requires including a different set of source files in the compilation of your programs and possibly linking to a different interface library. The assembly language interface is a register interface in which you must set up a group of registers with parameter values before calling the system calls.

See also: Assembly Language Interfaces, Appendix C
Program Development, *Installation and User's Guide*

Data Types

Throughout this manual a special set of Kernel data types is used to describe parameters and structures. With one exception (POINTER), these data types are defined in the include files that accompany the product. The Kernel data types correspond to PL/M, C, and FORTRAN data types as listed in Table 1-1 below.

The header file *rmk_type.l* contains the definition of data types for C. The header file *rmk_type.lit* contains the definition of data types for PL/M. The header file *rmk_type.par* contains the definition of data types for FORTRAN.

Table 1-1. Kernel Data Types Referenced to Other Languages

Kernel Data Types	PL/M Data Types	C Language Data Types	FORTRAN Data Types
UINT_8	BYTE	char	INT*1
UINT_8	BYTE	boolean	INT*1
UINT_16	HWORD	unsigned short	INT*2
UINT_32	WORD	unsigned int	INT*4
UINT_64	DWORD	unsigned long*	Not Applicable
POINTER	POINTER	See note below	Not Available

* 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
```

NOTE

In small model, each pointer is a 32-bit offset relative to the Kernel's data or code segment. In compact model, each pointer is a 48-bit quantity. A fill parameter is used in small model to ensure structures are the same size in both small and compact models.

There is no generic pointer type in C. Pointers must be typed and they can be cast. In general, when the system calls return pointers, they return pointers to UINT_8 items (UINT_8 *). When you pass a pointer to a system call, pass a reference to the code or data item.

Description headings

When describing the system calls, Chapter 2 provides several standard categories of information that are always listed in the same order. The categories include:

Description

This section describes how the system call works.

Scheduling Category

This category indicates what effect a system call may have on task scheduling and whether a scheduling lock changes that effect. It also indicates whether the system call can be safely used by interrupt handlers, which should not lose control of the CPU when they run. The possible types are:

Non-scheduling (Safe). The system call does not cause rescheduling, and interrupt handlers can safely use it.

Signalling. The system call could put other tasks in the ready state. If those tasks are higher priority, rescheduling would occur, pre-empting the calling task. If this system call is called from an interrupt handler, the handler could lose control. A scheduling lock will prevent rescheduling when using such a system call. Any task state change caused by a signalling system call takes place immediately, but the running task is not switched until scheduling is started again.

Blocking. The system call could put the running task to sleep causing rescheduling. An interrupt handler should not call this system call unless it knows that the running task won't be put to sleep as a result (the system call will complete its operation without blocking the calling task). A scheduling lock does not prevent a blocking system call from causing rescheduling.

Rescheduling (Unsafe). This system call always causes rescheduling. An interrupt handler should never call this system call. A scheduling lock does not prevent rescheduling for this system call.

Return Value

If the system call is a function, this section describes the value returned by the function.

Parameters

This section describes the parameters, if any, that you must specify when calling the system call.

The Kernel declares literals to define many of the data structures and parameter values needed. To use the Kernel-defined values when setting up data structures and calling the system calls, include the appropriate literal files in your programs.

See also: Include Files, *Installation and User's Guide*

All structures in this manual are shown in small model. Fill parameters are used in structures to account for the difference between compact model, which uses 48 bit pointers, and small model, which uses 32 bit pointers.

Flags Parameters

Masks typically refer to a single bit field in the flag. A mask is used to isolate a value in the flags field when you examine a flag. To set a flag, choose one literal value for each mask listed. Then OR the values together to form the flags value.

For example, these are the flags for the **create_semaphore** system call.

flags A KN_FLAGS whose bit structure specifies the following attributes of the semaphore:

Type — Specifies the type of semaphore. The following literals apply to this flag:

Literal	Meaning
KN_EXCH_TYPE_MASK	A mask for this field of the flag.
KN_FIFO_QUEUEING	The semaphore uses FIFO queueing
KN_PRIORITY_QUEUEING	The semaphore uses priority queueing
KN_REGION	The exchange is a region with one unit

Units — Specifies the number of initial units the semaphore receives. The following literals apply to this flag:

Literal	Meaning
KN_INITIAL_SEM_STATE_MASK	A mask for this field of the flag.
KN_ZERO_UNITS	The semaphore is created with no units
KN_ONE_UNIT	The semaphore is created with one unit

To set up a semaphore that uses FIFO queueing and has one unit, the literal values you should choose for the flags are:

```
KN_FIFO_QUEUEING OR KN_ONE_UNIT
```

***

KERNEL SYSTEM CALLS 2

This chapter provides detailed descriptions of all the Kernel system calls, organized alphabetically by system call name. Table 2-1 provides a dictionary of system calls organized by functional groups and is useful for locating the specific system call name and reference desired.

Dictionary of System Calls

Table 2-1 presents the various Kernel system calls by functional groups. Within each group each system call is listed with a description of its basic function and the page reference for the detailed description. Table 2-1 shows the header files to include for C-language programs. Include files for other languages have the same base name with a different filename extension.

See also: Include Files, *Installation and User's Guide*

Table 2-1. Kernel System Calls

System Call Name	System Call Function	Include	Page
	CHARACTER I/O DEVICE SUPPORT		
KN_ci	Read ASCII character from console input device	rmk_dev.h	2-13
KN_co	Transfer a character to console output device	rmk_dev.h	2-14
KN_csts	Read immediate character, if any, from console input device	rmk_dev.h	2-35
KN_initialize_console	Initialize the console device	rmk_dev.h	2-63
initialize_stdio	Initialize stdio library functions	kstdio.h	2-86
stdio functions	See Chapter 3.	kstdio.h	

Table 2-1. Kernel System Calls (Continued)

System Call Name	System Call Function	Include	Page
	COMMUNICATION AND SYNCHRONIZATION		
KN_create_mailbox	Create a mailbox	rmk_base.h	2-20
KN_create_semaphore	Create a semaphore	rmk_base.h	2-25
KN_delete_mailbox	Delete a mailbox	rmk_base.h	2-39
KN_delete_semaphore	Delete a semaphore	rmk_base.h	2-41
KN_receive_data	Request a message from a mailbox	rmk_base.h	2-97
KN_receive_unit	Receive a unit from a semaphore	rmk_base.h	2-99
KN_send_data	Send data to a mailbox	rmk_base.h	2-104
KN_send_priority_data	Place priority message at head of mailbox queue	rmk_base.h	2-114
KN_send_unit	Add a unit to a semaphore	rmk_base.h	2-127
	DEBUGGER INITIALIZATION		
KN_initialize_RDS	Initialize the debugger	rmk_base.h	2-81
	DESCRIPTOR TABLE MANAGEMENT		
KN_get_code_selector	Get the selector for the current code segment	rmk_dt.h	2-43
KN_get_data_selector	Get the selector for the current data segment	rmk_dt.h	2-44
KN_get_descriptor_attributes	Get a descriptor's attributes	rmk_dt.h	2-45
KN_initialize_LDT	Initialize a descriptor to reference an LDT	rmk_dt.h	2-67
KN_linear_to_ptr	Generate a pointer corresponding to a given linear address	rmk_dt.h	2-89
KN_null_descriptor	Overwrite a descriptor with the null descriptor	rmk_dt.h	2-95
KN_ptr_to_linear	Generate a linear address corresponding to a given pointer	rmk_dt.h	2-96
KN_set_descriptor_attributes	Set a descriptor's attributes	rmk_dt.h	2-128
	INTERCONNECT SPACE MANAGEMENT		
KN_get_interconnect	Get the value of an interconnect register	rmk_is.h	2-51
KN_initialize_interconnect	Initialize the interconnect module	rmk_is.h	2-65
KN_local_host_ID	Get the host ID of the local host	rmk_is.h	2-90
KN_set_interconnect	Set the value of an interconnect register	rmk_is.h	2-136

Table 2-1. Kernel System Calls (Continued)

System Call Name	System Call Function	Include	Page
	INTERRUPT AND PIC MANAGEMENT		
KN_get_slot	Return the highest priority active interrupt slot	rmk_dev.h	2-55
KN_initialize_PICs	Initialize a PIC	rmk_dev.h	2-77
KN_mask_slot	Mask out interrupts on a specified slot	rmk_dev.h	2-91
KN_new_masks	Change the masking of interrupt slots	rmk_dev.h	2-94
KN_send_EOI	Signal PIC that interrupt on specified slot has been serviced	rmk_dev.h	2-113
KN_set_interrupt	Establish an interrupt handler for a particular IDT slot	rmk_base.h	2-137
KN_unmask_slot	Unmask interrupts on a specified slot	rmk_dev.h	2-150
	KERNEL INITIALIZATION		
KN_initialize	Initialize the Kernel	rmk_base.h	2-57
	MEMORY MANAGEMENT		
KN_create_area	Create a memory area from a pool (allocate memory)	rmk_base.h	2-18
KN_create_pool	Create a memory pool	rmk_base.h	2-23
KN_delete_area	Return the memory from a memory area to the memory pool	rmk_base.h	2-38
KN_delete_pool	Delete a memory pool	rmk_base.h	2-40
KN_get_pool_attributes	Get a memory pool's attributes	rmk_base.h	2-53

Table 2-1. Kernel System Calls (Continued)

System Call Name	System Call Function	Include	Page
	MESSAGE PASSING MANAGEMENT		
KN_attach_protocol_handler	Establish an interrupt handler for receiving data link messages	rmk_mp.h	2-6
KN_attach_receive_mailbox	Associate a mailbox with a port ID	rmk_mp.h	2-8
KN_cancel_dl	Cancel a data link buffer request	rmk_mp.h	2-10
KN_cancel_tp	Cancel a solicited message or request-response transaction	rmk_mp.h	2-12
KN_initialize_message_passing	Initialize the message passing module	rmk_mp.h	2-69
KN_mp_working_storage_size	Compute work space size needed for message passing	rmk_mp.h	2-92
KN_send_dl	Send a data link message	rmk_mp.h	2-106
KN_send_tp	Send a transport protocol message	rmk_mp.h	2-116
	NUMERIC COPROCESSOR MANAGEMENT		
KN_initialize_NDP	Initialize the Numeric Coprocessor	rmk_dev.h	2-74
	PIT MANAGEMENT		
KN_get_PIT_interval	Return the PIT interval	rmk_dev.h	2-52
KN_initialize_PIT	Initialize a PIT	rmk_dev.h	2-79
KN_start_PIT	Start the PIT counting	rmk_dev.h	2-141
	SUBSYSTEM SUPPORT		
KN_initialize_subsystem	Allows application to be divided into multiple subsystems when application interfaces to Kernel through a call gate	rmk_dev.h	2-87
KN_translate_ptr	Converts pointer that will be based on user-specified selector	rmk_base.h	2-148

Table 2-1. Kernel System Calls (Continued)

System Call Name	System Call Function	Include	Page
	TASK MANAGEMENT		
KN_create_task	Create a task	rmk_base.h	2-27
KN_current_task_token	Return a token for the current task	rmk_base.h	2-36
KN_delete_task	Delete a task	rmk_base.h	2-42
KN_get_priority	Return static priority of a task	rmk_base.h	2-54
KN_reset_handler	Remove previously set task handler	rmk_base.h	2-102
KN_resume_task	Cancel one level of task suspension	rmk_base.h	2-103
KN_set_handler	Set task handler dynamically	rmk_base.h	2-134
KN_set_priority	Set the static priority of a task	rmk_base.h	2-138
KN_start_scheduling	Cancel one scheduling lock	rmk_base.h	2-142
KN_stop_scheduling	Temporarily lock the scheduling mechanism	rmk_base.h	2-143
KN_suspend_task	Add one level to task suspension	rmk_base.h	2-144
KN_token_to_ptr	Return a pointer to the area holding the object	rmk_base.h	2-147
	TIME MANAGEMENT		
KN_create_alarm	Create and start a virtual alarm clock	rmk_base.h	2-15
KN_delete_alarm	Delete an alarm	rmk_base.h	2-37
KN_get_time	Get the current value of the Kernel clock timer	rmk_base.h	2-56
KN_reset_alarm	Reset an existing alarm	rmk_base.h	2-101
KN_set_time	Set the kernel clock timer	rmk_base.h	2-139
KN_sleep	Put the calling task to sleep	rmk_base.h	2-140
KN_tick	Notify Kernel that a clock tick has occurred	rmk_base.h	2-145

attach_protocol_handler

```
void KN_attach_protocol_handler(protocol_id, handler_ptr,
                                flags);
```

Data Type		Parameter
KN_PROTOCOL_ID		protocol_id
void	*	handler_ptr
KN_FLAGS		flags

Description

The **attach_protocol_handler** system call is used with the data link layer of the message passing module. This 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.

If a protocol handler is already established for the specified protocol ID, the new handler overrides the existing handler. The data link layer will start invoking the new protocol handler the next time a message arrives which specifies that protocol ID.

Before the data link layer's interrupt handler invokes the protocol handler, it stops the Kernel scheduling (calls **stop_scheduling**). When the protocol handler returns, the interrupt handler re-starts scheduling (calls **start_scheduling**).

Protocol handlers should perform whatever operations are necessary to handle arriving messages. However, they should not send data link messages themselves. Instead, they should signal other tasks to send responses to the messages they receive.

See also: Message Passing, *Installation and User's Guide*

NOTE

To receive Multibus II MIC type (interrupt control) messages, which are four bytes in length, attach a protocol handler with a protocol ID of 0. Whenever a four-byte message is received, the data link module invokes the user-supplied handler. The environment for this handler is the same as that for other protocol handlers.

The message remote host ID and type (KN_UNSOL) are the only defined fields of a received MIC message. Users can send messages to a MIC-based host using the **send_dl** system call.

Scheduling Category

Unsafe for use by interrupt handlers.

Parameters

protocol_id

A KN_PROTOCOL_ID specifying the protocol ID to be associated with a specific protocol handler. Specify any value between 80h and 0FFh. Values 1 through 7Fh are reserved for Intel applications.

handler_ptr

A pointer to the first instruction of the protocol handler. The protocol handler must be re-entrant. In the small model interface, the handler pointer is assumed to be relative to the caller's CS register. This handler must have the following calling sequence:

```
protocol_handler(message_ptr);
```

Where:

message_ptr: A pointer to an area where the incoming message resides. This message has the KN_DATA_LINK_MSG structure shown in the **send_dl** system call.

attach_receive_mailbox

```
status = KN_attach_receive_mailbox(port_ID, mailbox);
```

Data Type	Parameter
KN_PORT_ID	port_ID
KN_TOKEN	mailbox

Description

The **attach_receive_mailbox** system call is used with the transport layer of the message passing protocol. It associates a mailbox with a port ID. Once this association has been made, all incoming messages for the specified port are queued at the associated mailbox. Port IDs can be assigned to the range 0-0FFFFh. The application assigns port IDs and must maintain their consistent use.

Each port can be serviced by only one mailbox. Assigning another mailbox to the same port ID overrides the existing mailbox assignment. A single mailbox can be assigned to service more than one port.

Transport protocol messages received via this mailbox have the structure KN_TRANSPORT_MBX_REMOTE_MSG or KN_TRANSPORT_MBX_LOCAL_MSG. Refer to the description of the **send_tp** system call for more information about these structures. The message type is KN_UNSOL, KN_BROADCAST, or KN_BUFFER_REQUEST.

Scheduling Category

Rescheduling. Unsafe for use by interrupt handlers.

Returned Value

status A KN_STATUS indicating the result of the requested operation. Values are:

Literal	Meaning
E_OK	Indicates that the call completed successfully.
E_RESOURCE_LIMIT	Indicates that an internal resource limit has been reached.

Parameters

port_id

A KN_PORT_ID indicating the value of the port ID for which messages are to be queued. This parameter can have any value from 0 through 0FFFFh.

mailbox

A KN_TOKEN for the mailbox at which messages for the specified port ID are to be queued.

cancel_dl

```
status = KN_cancel_dl(message_ptr);
```

Data Type		Parameter
KN_STATUS		status
KN_DATA_LINK_MSG	*	message_ptr

Description

The **cancel_dl** system call is used with the data link layer of the message passing protocol. It cancels a previously queued buffer request, buffer grant, or ongoing solicited message transfer.

Cancel a sent solicited message any time after the buffer request is sent, until the transfer is complete. Cancel a received solicited message any time after the buffer grant is sent, until the transfer is complete.

See also: Message Passing, *Installation and User's Guide*

Scheduling Category

Rescheduling. Unsafe for use by interrupt handlers.

Returned Value

status A KN_STATUS indicating the result of the requested operation. Values are:

Literal	Meaning
E_OK	The call completed successfully.
E_TOO_LATE	The request came too late to cancel the message.

Parameters

message_ptr

A pointer to the buffer request or buffer grant message associated with the solicited transfer. A sending task points to the buffer request message; a receiving task points to the buffer grant message. The pointer should point to a KN_DATA_LINK_MSG structure. See the **send_dl** system call for a description of the KN_DATA_LINK_MSG structure.

If the solicited message is successfully cancelled (status=E_OK), the Kernel changes the `data_status` value in the message to E_SO_CANCEL or E_SI_CANCEL (cancel solicited output or cancel solicited input).

cancel_tp

```
status = KN_cancel_tp(message_ptr);
```

Data Type		Parameter
KN_STATUS		status
void	*	message_ptr

Description

The **cancel_tp** system call is used with the transport layer of the message passing protocol. It cancels an ongoing solicited message or request-response transaction. This system call is a local operation only. It does not send a cancellation message to the other host.

Cancel a solicited message any time after the buffer request is sent, until the transfer is complete. Cancel a solicited message any time after the buffer grant is sent, until the transfer is complete. Cancel a request-response transaction any time after the request is sent, until the response is received.

Scheduling Category

Rescheduling. Unsafe for use by interrupt handlers.

Return Value

status A KN_STATUS indicating the result of the requested operation. Values are:

Literal	Meaning
E_OK	The call completed successfully.
E_TOO_LATE	The request came too late to cancel the message.
E_TRANS_ID	Non-unique or invalid transaction ID in the message.

Parameters

message_ptr

A pointer to a KN_TRANSPORT_MSG or a KN_RSVP_TRANSPORT_MSG structure containing the message to be cancelled. See the **send_tp** system call for descriptions of these structures.

```
char = KN_ci();
```

Data Type	Parameter
UINT_8	char

Description

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.

See also: **initialize_console**

Scheduling Category

Blocking. Use with caution in interrupt handlers.

Return Value

char A UINT_8 variable which receives the ASCII character entered at the console.

co

```
void KN_co(char);
```

Data Type	Parameter
UINT_8	char

Description

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 it was, **co** waits for the operator to enter CONTROL-Q before transmitting the character.

See also: **initialize_console**

Scheduling Category

Blocking. Use with caution in interrupt handlers.

Parameters

char A UINT_8 containing the ASCII character to be output to the console.

```
alarm = KN_create_alarm(area_ptr, handler_ptr, time_limit,
                        flags);
```

Data Type		Parameter
KN_TOKEN		alarm
UINT_32	*	area_ptr
void	*	handler_ptr
UINT_32		time_limit
KN_FLAGS		flags

Description

The **create_alarm** system call creates and starts a virtual alarm clock. With this system call, specify a time limit and a handler. When the time limit elapses, the Kernel invokes the handler, thereby simulating a timer interrupt. When the alarm handler is invoked, interrupts are disabled and scheduling is locked.

Two types of alarms can be set, depending on the value of the flags parameter. A single shot alarm becomes inactive after its initial time interval elapses, and its memory becomes available for reuse. A repetitive alarm resets after each invocation of the handler so that the handler is called again after the next interval elapses. Repetitive alarms generate periodic interrupts until they are explicitly deleted.

See also: Time Management, Interrupt Management, *Installation and User's Guide*

Scheduling Category

Non-scheduling. Safe for use by interrupt handlers.

Return Value

alarm A KN_TOKEN for the newly created alarm.

create_alarm

Parameters

area_ptr

A pointer to an area to be used to hold the alarm's state. The area supplied must be at least KN_ALARM_SIZE bytes long.

handler_ptr

A pointer to a procedure to be executed when the time period elapses. This procedure is called with scheduling stopped and interrupts disabled. The application must ensure that the mapping of the `handler_ptr` parameter to physical memory remains constant until either the alarm is deleted or until a single-shot alarm handler is invoked. This pointer is assumed to be relative to the caller's code segment. The entry point for an alarm interrupt handler is written as follows:

```
alarm_handler(alarm_ptr);
```

Where:

alarm_ptr A pointer to the area holding the alarm's state. If additional information is associated with the alarm, this pointer can be used to access it.

time_limit

A UINT_32 specifying the number of clock ticks which must elapse before the handler is invoked. A value of 0 indicates that the alarm handler is called on the next clock tick (and for repetitive alarms, on every clock tick).

A value of 1 indicates that the remainder of the current clock tick and one complete clock tick occurs before the alarm handler is called.

flags A KN_FLAGS whose bit structure specifies the following attributes of the alarm:

Alarm Type Specifies whether the alarm generates a single interrupt or repeated interrupts. The following literals apply to this flag:

Literal	Meaning
KN_ALARM_REPETITION_MASK	A mask for this field of the flag.
KN_SINGLE_SHOT	The alarm object is to generate a single interrupt.
KN_REPEATER	The alarm object is to generate repeated interrupts.

Calling Convention

Specifies whether the handler is in the same subsystem or a different subsystem from the Kernel. The following literals apply to this flag:

Literal	Meaning
KN_HANDLER_CONVENTION_MASK	A mask for this field of the flag.
KN_CALL_NEAR	The handler is in the same subsystem as the Kernel.
KN_CALL_FAR	The handler is in a different subsystem from the Kernel's.

create_area

```
area = KN_create_area(pool, size);
```

Data Type		Parameter
void	*	area
KN_TOKEN		pool
UINT_32		size

Description

The **create_area** system call allocates an area of memory of the specified size from the specified memory pool (previously created using the **create_pool** system call). If the memory pool was created from memory aligned on a four-byte boundary, the area assigned with this system call will also be aligned on a four-byte boundary. If there is insufficient contiguous memory in the pool to satisfy the request, a null pointer is returned.

To allocate an area of size X, an available area of size X + KN_AREA_OVERHEAD must exist within the pool. KN_AREA_OVERHEAD is the number of bytes of overhead associated with each area allocated from the pool.

See also: Memory Management, Pool and Area Overhead, *Installation and User's Guide*

Scheduling Category

Blocking. Use with caution in interrupt handlers.

Return Value

area A pointer to an area of the desired size. If no area can be allocated, the Kernel returns a null pointer.

Parameters

pool A KN_TOKEN for the memory pool from which the area is to be allocated. This is the token returned from a **create_pool** system call.

size A UINT_32 specifying the size of the requested area in bytes. This value can range from KN_MINIMUM_AREA_SIZE to the `pool_largest` value returned by the **get_pool_attributes** system call. If you specify a value smaller than KN_MINIMUM_AREA_SIZE, the Kernel rounds up the request to the minimum size.

create_mailbox

```
mailbox = KN_create_mailbox(area_ptr, message_size, queue_size,
                            flags);
```

Data Type		Parameter
KN_TOKEN		mailbox
UINT_32	*	area_ptr
UINT_32		message_size
UINT_32		queue_size
KN_FLAGS		flags

Description

The **create_mailbox** system call creates a mailbox in a specified area of memory. Once the mailbox is created, tasks can send messages to it by calling the **send_data** system call or the **send_priority_data** system call, and they can receive messages from it by calling the **receive_data** system call.

When you create a mailbox, you specify the queuing mechanism that is used when tasks wait for messages at the mailbox. Both FIFO and priority queuing are possible. FIFO queuing means that tasks are queued in the order that they arrive at the mailbox. In priority queuing, the tasks are queued based on their task priority.

When you create a mailbox, you can specify one of the slots in its queue as reserved for a high priority message. When the high priority message comes, the Kernel attempts to place this message ahead of all the other messages in the regular queue. If the message queue is full, the Kernel puts the high-priority message into the reserved slot instead. If the reserved slot is also taken, an exception (E_LIMIT_EXCEEDED) is returned. This is the same exception code that is returned when a non-priority message cannot be sent because the mailbox queue is full.

The purpose of the reserved slot is to ensure at least one high priority message is accepted even when the mailbox queue is full.

Scheduling Category

Non-scheduling. Safe for use by interrupt handlers.

Return Value

mailbox

A KN_TOKEN for the newly created mailbox.

Parameters

area_ptr

A pointer to the area in which the mailbox is to be created. For better performance, this area should be aligned on a four-byte boundary. The size of this area must be:

KN_MAILBOX_SIZE + (message_size + KN_MAILBOX_MSG_OVERHEAD) * queue_size

Literal	Meaning
KN_MAILBOX_SIZE	The number of bytes required for a mailbox object, excluding the message queue.
KN_MAILBOX_MSG_OVERHEAD	The number of bytes of overhead for each message in the message queue of a mailbox.

message_size

A UINT_32 specifying the maximum size in bytes of the messages to be exchanged through this mailbox. When sending messages to the mailbox, never send messages larger than the maximum message size specified for the mailbox. Sending messages which are larger than the maximum specified message size may produce unexpected results.

NOTE

Keep messages as small as possible. Transferring large messages can degrade the interrupt latency of the system.

create_mailbox

queue_size

A UINT_32 specifying the maximum number of messages that can be stored in the mailbox. When the Kernel assigns messages to the mailbox, it assigns them in a circular fashion, assuming that the number of message slots is equal to `queue_size` and the size of each message is equal to `message_size`.

Even if the number of messages queued at the mailbox never reaches `queue_size`, the circular queuing means that all the memory allocated for messages will be accessed at one time or another. Therefore it is important that the amount of memory you assign to the mailbox matches the values you specify for `message_size` and `queue_size`.

flags A KN_FLAGS specifying the type of mailbox to be created.

Exchange Type

Specifies whether the mailbox uses FIFO or Priority queueing. The following literals apply to this flag:

Literal	Meaning
KN_EXCH_TYPE_MASK	A mask for this field of the flag.
KN_FIFO_QUEUEING	The mailbox uses FIFO queueing.
KN_PRIORITY_QUEUEING	The mailbox uses priority queueing.

Reserved Priority

Specifies whether the mailbox queue has a slot reserved for a high priority message.

Literal	Meaning
KN_RESERVE_PRIORITY_DATA_MASK	A mask for this field of the flag.
KN_DONT_RESERVE_PRIORITY_DATA	Don't reserve a slot for a high priority message.
KN_RESERVE_PRIORITY_DATA	Reserve a slot for a high priority message.

```
pool = KN_create_pool(pool_ptr, size);
```

Data Type		Parameter
KN_TOKEN		pool
void	*	pool_ptr
UINT_32		size

Description

The **create_pool** system call creates a memory pool in a specified range of memory. Once the memory pool is created, tasks can access portions of this memory by invoking the **create_area** system call, but they should not access the memory without calling **create_area**. If the memory used to contain the pool is aligned on a four-byte boundary, all areas allocated from the pool are also aligned on four-byte boundaries.

See also: Memory Management, Pool and Area Overhead, *Installation and User's Guide*

Scheduling Category

Non-scheduling. Safe for use by interrupt handlers.

Return Value

pool A KN_TOKEN for the newly created memory pool.

create_pool

Parameters

pool_ptr

A pointer to the first location in memory to be included in the new memory pool.

size A UINT_32 specifying the number of bytes to include in the new memory pool.

One way to determine the size needed is to consider the number of areas that could conceivably be allocated at the same time. For many applications, all areas allocated from a memory pool are of the same size. Therefore, to create a pool that can exactly allocate N areas all of size M, an area of the following size is required for the pool. M must be greater than or equal to KN_MINIMUM_AREA_SIZE:

N * (M + KN_AREA_OVERHEAD) + KN_POOL_OVERHEAD

Literal	Meaning
KN_AREA_OVERHEAD	The number of bytes of overhead associated with each area allocated from the pool.
KN_POOL_OVERHEAD	The number of bytes of overhead in a new pool. If a pool of X bytes is desired, a pool of X + KN_POOL_OVERHEAD must be requested using the **create_pool** system call. The smallest pool size is therefore: KN_MINIMUM_POOL_SIZE + KN_POOL_OVERHEAD
KN_MINIMUM_POOL_SIZE	The minimum number of bytes necessary for a pool object.
KN_MINIMUM_AREA_SIZE	The smallest area which can be allocated from a memory pool.

```
semaphore = KN_create_semaphore(area_ptr, flags);
```

Data Type		Parameter
KN_TOKEN		semaphore
UINT_32	*	area_ptr
KN_FLAGS		flags

Description

The **create_semaphore** system call creates one of three kinds of semaphores with zero or one initial units. The kinds of semaphores that can be created are FIFO, priority, and region. With FIFO semaphores, tasks that wait for units at the semaphore are queued in the order they arrive. With priority semaphores, tasks are queued according to their task priorities. FIFO and priority semaphores can contain as many as 65,535 units, which are placed in the semaphore by calling **send_unit** once for each unit. If more than 65,535 units are sent to the semaphore, the Kernel does not increment the semaphore, but rather invokes the disaster handler with an exception code of E_LIMIT_EXCEEDED and an invoked function code of KN_SEND_UNIT_CODE.

Regions are special cases of priority semaphores that manage a single unit. A task holding a region's unit is the owner of the region, and its priority is dynamically adjusted depending on the priority of other tasks waiting to access the region. If a low-priority task has access to the region and a high-priority task requests access (via the **receive_unit** system call), the priority of the task holding the region is temporarily raised to the priority of the waiting task. The new, higher priority of this task prevents it from being placed in the ready queue by the Kernel until it finishes its work within the region. When the task relinquishes all regions (by calling the **send_unit** system call), its priority returns to its normal (static) level.

If a region is created with 0 units, the creating task is considered to be holding the region's unit and is therefore the owning task. If a region is created with 1 unit, no task owns the region until it invokes **receive_unit** for that region.

Scheduling Category

Non-scheduling. Safe for use by interrupt handlers.

create_semaphore

Return Value

semaphore
A KN_TOKEN for the newly created semaphore.

Parameters

area_ptr
A pointer to the area in which the semaphore is to be created. This area must be at least KN_SEMAPHORE_SIZE bytes long. For better performance, this area should be aligned on a four-byte boundary.

flags A KN_FLAGS whose bit structure specifies the following attributes of the semaphore:

Type — Specifies the type of semaphore. The following literals apply to this flag:

Literal	Meaning
KN_EXCH_TYPE_MASK	A mask for this field of the flag.
KN_FIFO_QUEUEING	The semaphore uses FIFO queueing
KN_PRIORITY_QUEUEING	The semaphore uses priority queueing
KN_REGION	The exchange is a region with one unit

Units — Specifies the number of initial units the semaphore receives. The following literals apply to this flag:

Literal	Meaning
KN_INITIAL_SEM_STATE_MASK	A mask for this field of the flag.
KN_ZERO_UNITS	The semaphore is created with no units
KN_ONE_UNIT	The semaphore is created with one unit

```
task = KN_create_task(area_ptr, stack_ptr, code_ptr,
                      data_seg, priority, flags);
```

DATA TYPE		PARAMETER
KN_TOKEN		task
UINT_32	*	area_ptr
void	*	stack_ptr
void	*	code_ptr
void	*	data_seg
UINT_16		priority
KN_FLAGS		flags

Description

The **create_task** system call creates a new task using the supplied resources. The Kernel assigns this new task its own task state and, by default, gives it the same LDT as the calling task.

When you invoke **create_task**, specify the initial values associated with the task (its code, data and stack segments, its priority, and whether it is created in the ready or suspended state). Also supply an area that the Kernel uses to store the task's state.

When you invoke **create_task**, the Kernel overwrites most of the fields in the task state. However, you can set some of these fields before calling **create_task** to set up the task in special ways. For example, you can set up the stacks for different privilege rings before calling **create_task**. Other fields that the Kernel initializes can be set after calling **create_task**, so that you can change the Kernel's defaults. For example, you can change the LDT associated with the new task and change its time slice, but not until after you call **create_task**.

See the description of KN_TASK_STATE in this section for more information about the fields that can be changed and the fields that must be left as is.

create_task

The new task's segments, referenced by the `stack_ptr`, `code_ptr` and `data_seg` parameters, must all have the same descriptor privilege level (DPL). This privilege level determines the initial CPL (current privilege level) of the new task. The **create_task** system call sets up a stack for the new task only at this privilege level. If the task must execute at any other privilege levels, the application is responsible for setting up additional stacks for the task.

The application can set up additional stacks by reserving memory for the stack, accessing the task state area, and assigning values to the `SSi` and `ESPi` (where i = 0, 1, or 2) fields. These changes can be made before calling **create_task**, because the Kernel doesn't initialize these fields of the TSS. The application can provide a task creation handler to perform this function, or the calling task can access the KN_TASK_STATE structure pointed to by `area_ptr` and fill in those fields before calling **create_task**.

Tasks must always have a ring 0 stack. Thus if a task is created whose initial CPL is greater than 0, a ring 0 stack must be supplied by one of the methods mentioned.

Tasks can access their own task states and the task states of other tasks in multiple ways. A task can access its own task state by using the **current_task** system call, which provides a pointer to the state of the current task. A task can access the state of another task by supplying the second task's token as a parameter to the **token_to_ptr** system call.

The Kernel implements task switches using software routines. Although the Kernel does not use hardware task switching, it still creates a task state segment for each task.

See also: Kernel Handlers, Chapter 4
Stacks, Task Management, *Installation and User's Guide*

Scheduling Category

Signalling. Use scheduling lock in interrupt handlers.

Return Value

task A KN_TOKEN for the newly-created task.

Parameters

area_ptr

A pointer to the area in which the Kernel stores the new task's task state. This area must be at least KN_TASK_SIZE bytes long, and it must be aligned on a four-byte boundary. If you include the Kernel's numeric coprocessor support in your system and you use the default processor state handler, you must allocate an additional KN_387_SAVE_AREA_SIZE bytes for the Kernel to use to save and restore the state of the coprocessor. Refer to the description of **initialize_NDP** for more information about setting up the Numeric Coprocessor Module.

KN_TASK_SIZE

The number of bytes of memory required to contain the task state.

KN_387_SAVE_AREA_SIZE

The number of bytes of memory required to contain the coprocessor state.

KN_TASK_STATE

A structure that can overlay a task state to provide access to the individual fields. This structure has the format shown below. The fields from `link` through `IO_map_base` correspond to fields in the TSS. The remaining fields are specific to the Kernel.

Only the fields `ESPi`, `SSi` (where i = 0 through 2), `CR3_reg`, `LDT_reg`, `TRAP_reg`, `IO_map_base`, and `task_slice` can be written by applications. If applications attempt to write other fields, the results are undefined. Only the `ESPi` and `SSi` fields can be written before calling **create_task**. The **create_task** system call initializes the other fields regardless of how they were set before the system call was invoked.

```
typedef struct {
    KN_SELECTOR             link;
    UINT_16                 link_h;
    UINT_32                 ESP0;
    KN_SELECTOR             SS0;
    UINT_16                 SS0_h;
    UINT_32                 ESP1;
    KN_SELECTOR             SS1;
    UINT_16                 SS1_h;
    UINT_32                 ESP2;
    KN_SELECTOR             SS2;
    UINT_16                 SS2_h;
    UINT_32                 CR3_reg;
    UINT_32                 EIP_reg;
    UINT_32                 EFLAGS_reg;
    UINT_32                 EAX_reg;
    UINT_32                 ECX_reg;
    UINT_32                 EDX_reg;
    UINT_32                 EBX_reg;
    UINT_32                 ESP_reg;
    UINT_32                 EBP_reg;
    UINT_32                 ESI_reg;
    UINT_32                 EDI_reg;
    KN_SELECTOR             ES_reg;
    UINT_16                 ES_h;
    KN_SELECTOR             CS_reg;
    UINT_16                 CS_h;
    KN_SELECTOR             SS_reg;
    UINT_16                 SS_h;
    KN_SELECTOR             DS_reg;
    UINT_16                 DS_h;
    KN_SELECTOR             FS_reg;
    UINT_16                 FS_h;
    KN_SELECTOR             GS_reg;
    UINT_16                 GS_h;
    KN_SELECTOR             LDT_reg;
    UINT_16                 LDT_h;
    UINT_16                 TRAP_reg;
    UINT_16                 IO_map_base;
    KN_TOKEN                task token;
    UINT_32                 task slice;
    UINT_16                 dynamic priority;
    UINT_16                 static_priority;
    KN_FLAGS                flags;
} KN_TASK_STATE;
```

Where:

link	A back link to the previous TSS.
link_h	A reserved field in the TSS.
ESP0	ESP register for privilege ring 0 operation.
SS0	SS register for privilege ring 0 operation.
SS0_h	A reserved field in the TSS.
ESP1	ESP register for privilege ring 1 operation.
SS1	SS register for privilege ring 1 operation.
SS1_h	A reserved field in the TSS.
ESP2	ESP register for privilege ring 2 operation.
SS2	SS register for privilege ring 2 operation.
SS2_h	A reserved field in the TSS.
CR3_reg	CR3 register.
EIP_reg	EIP register.
EFLAGS_reg	EFLAGS register.
EAX_reg	EAX register.
ECX_reg	ECX register.
EDX_reg	EDX register.
EBX_reg	EBX register.
ESP_reg	ESP register.
EBP_reg	EBP register.
ESI_reg	ESI register.
EDI_reg	EDI register.
ES_reg	ES register.
ES_h	A reserved field in the TSS.
CS_reg	CS register.
CS_h	A reserved field in the TSS.

SS_reg SS register. Tasks that use the message passing module should not change this descriptor.

SS_h A reserved field in the TSS.

DS_reg DS register.

DS_h A reserved field in the TSS.

FS_reg FS register.

FS_h A reserved field in the TSS.

GS_reg GS register.

GS_h A reserved field in the TSS.

LDT_reg Selector for the LDT. By default, the new task's LDT is the same as its parent task's. To give the task a different LDT, you should assign memory for the LDT, set up an LDT descriptor in the GDT, and place a selector for that descriptor in the `LDT_reg` field.

LDT_h A reserved field in the TSS.

TRAP_reg The trap bit is bit 0 of the low-order byte.

IO_map_base Offset to the start of the I/O permission map from the base of the TSS.

task_token A token for the task.

task_slice The total number of clock ticks in the task's time slice. Once changed, this value becomes effective the next time the task receives a new time slice.

dynamic_priority
The current dynamic priority of the task. This field is equal to the static priority field unless the task's priority has been adjusted because of region ownership, in which case it is equal to the adjusted priority. The dynamic priority of tasks is used in scheduling the processor.

static_priority The current static priority of the task. This field gives the priority of the task if priority adjustment due to regions is ignored.

flags A KN_FLAGS whose bit structure specifies the following attributes of the task:

Idle task — Specifies whether the task is the idle task. The following literals apply to this flag:

Literal	Meaning
KN_IDLE_TASK_MASK	A mask for this field of the flag.
KN_IDLE_TASK	The task is the idle task.
KN_NOT_IDLE_TASK	The task is not the idle task.

Initial state — Specifies the initial state of the task. The following literals apply to this flag:

Literal	Meaning
KN_INITIAL_TASK_STATE_MASK	A mask for this field of the flag.
KN_CREATE_READY	Create the task in the ready state.
KN_CREATE_SUSPENDED	Create the task in the suspended state.

stack_ptr

A pointer to the stack to be used by the task. The task's SS and ESP registers are initialized using this pointer. The pointer must point to the highest memory address of the stack. Therefore, if you have a pointer to the start of the memory area that will be used as the stack, add the stack size to the offset portion of the pointer.

If the task is to execute in any other privilege rings, the application must set up additional stacks for the task and directly store pointers to them in the task's TSS. The application must ensure that the mapping of the `stack_ptr` parameter to physical memory remains constant for as long as the task executes in the stack. Stacks should be 4-byte aligned.

Compact applications can provide protected stacks by setting up descriptors for the area of memory used as the stack. In small model the stack and data are combined into the data segment, so no protected stacks can be created. For both compact and small applications, you must ensure that the area of memory reserved for the stack is large enough for the task's needs. However, for small model, extra care should be taken because stack overflows can cause unexpected errors by overwriting important information.

create_task

code_ptr

A pointer to the initial instruction to be executed by the new task. In the small model interface, this pointer is assumed to be relative to the caller's code segment. The application must ensure that the mapping of the `code_ptr` parameter to physical memory remains constant for as long as the task executes in the code segment.

data_seg

A pointer to the new task's data segment. The task's DS, ES, FS, and GS registers are loaded with the selector portion of this pointer (or for small model applications, the caller's DS is used). The application must ensure that the mapping of the `data_seg` parameter to physical memory remains constant for as long as the task uses the data segment.

priority

A UINT_16 specifying the priority of the new task. This parameter must be in the range 0-511, with 0 being the highest priority.

flags A KN_FLAGS whose bit structure specifies the following attributes of the task:

Task State — Specifies whether the task should be created in the ready state or in the suspended state. The following literals apply to this flag:

Literal	Meaning
KN_CREATE_READY	The task is to be created in the ready state.
KN_CREATE_SUSPENDED	The task is to be created in the suspended state.

```
char = KN_csts();
```

Data Type	Parameter
UINT_8	char

Description

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, **csts** returns an ASCII null character (value zero).

See also: **initialize_console**

Scheduling Category

Non-scheduling. Safe for use by interrupt handlers.

Return Value

char A UINT_8 variable which receives the ASCII character entered at the console. Its value is zero if there is no character available at the input device.

current_task_token

```
task = KN_current_task_token();
```

Data Type	Parameter
KN_TOKEN	task

Description

The **current_task_token** system call returns the token of the currently running task.

Scheduling Category

Non-scheduling. Safe for use by interrupt handlers.

Return Value

task A KN_TOKEN for the currently running task.

```
void KN_delete_alarm(alarm);
```

Data Type	Parameter
KN_TOKEN	alarm

Description

The **delete_alarm** system call deletes a previously created alarm. As a result of this call, the handler associated with the alarm will not be invoked. The area occupied by the alarm is available for reuse.

If the alarm you intend to delete is a single-shot alarm, you might not know if the alarm has already gone off (and thus has been deleted) when you invoke the **delete_alarm** system call. Therefore, it is acceptable to delete alarms even if they have already been deleted when they executed. This prevents race conditions in which task execution speed is responsible for error conditions.

NOTE

Do not delete an alarm that has not yet been created.

See also: Time Management, *Installation and User's Guide*

Scheduling Category

Non-scheduling. Safe for use by interrupt handlers.

Parameters

alarm A KN_TOKEN for the alarm to be deleted.

delete_area

```
void KN_delete_area(area, pool);
```

Data Type		Parameter
void	*	area
KN_TOKEN		pool

Description

The **delete_area** system call returns an area of memory to the memory pool from which it was allocated. The area becomes a part of the available space in the pool and should no longer be accessed directly by the application.

Scheduling Category

Blocking. Use with caution in interrupt handlers.

Parameters

area A pointer to the area to be deleted.

pool A KN_TOKEN for the memory pool from which the area was allocated.

```
void KN_delete_mailbox(mailbox);
```

Data Type	Parameter
KN_TOKEN	mailbox

Description

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. After this call, the memory assigned to the mailbox is available for reuse.

Scheduling Category

Signalling. Use scheduling lock in interrupt handlers.

Parameters

mailbox

A KN_TOKEN for the mailbox to be deleted.

delete_pool

```
void KN_delete_pool(pool);
```

Data Type	Parameter
KN_TOKEN	pool

Description

The **delete_pool** system call deletes a memory pool. This system call makes the entire address range of the memory pool available for reuse. No system calls that use the pool (such as **create_area** and **delete_area**) should be invoked after the pool has been deleted.

Memory pools can be deleted even if tasks currently have access to areas of memory allocated from those pools. The tasks accessing the areas will still have access to them. However, the Kernel does not prevent other tasks from accessing these in-use areas after the pool is deleted.

Scheduling Category

Non-scheduling. Safe for use by interrupt handlers.

Parameters

pool A KN_TOKEN for the memory pool to be deleted.

```
void KN_delete_semaphore(semaphore);
```

Data Type	Parameter
KN_TOKEN	semaphore

Description

The **delete_semaphore** system call deletes the specified semaphore. All tasks waiting at the semaphore are awakened with the E_NONEXIST status code.

If you delete a region semaphore while a task has access to the region, that task is no longer guarded by a region. Any dynamic adjustments that were made to that task's priority as a result of accessing the region are nullified and the task resumes its static priority. Because the region no longer exists, the task must not send the region's unit back to the region.

See also: Semaphores, *Installation and User's Guide*

Scheduling Category

Signalling. Use scheduling lock in interrupt handlers.

Parameters

semaphore
A KN_TOKEN for the semaphore to be deleted.

delete_task

```
void KN_delete_task(task);
```

Data Type	Parameter
KN_TOKEN	task

Description

The **delete_task** system call deletes the specified task regardless of the task's current state. The **delete_task** system call may be used to delete any task including the calling task. **Delete_task** does not return if it is invoked on the running task. All resources dedicated to the task are available for reuse when this system call returns.

The task to be deleted should not be inside a region, because the region will never again be available.

If you have defined a deletion task handler, the Kernel invokes the handler as part of executing the **delete_task** system call.

Scheduling Category

Non-scheduling. Safe for use by interrupt handlers.

Rescheduling when performed on the calling task. Unsafe for use by interrupt handlers when performed on the calling task.

Parameters

task A KN_TOKEN for the task to be deleted.

```
code_sel = KN_get_code_selector();
```

Data Type	Parameter
KN_SELECTOR	code_sel

Description

The **get_code_selector** system call is provided only with the Kernel's small-model interface library (*c_call.lib*). It returns a selector for the current code segment.

Scheduling Category

Non-scheduling. Safe for use by interrupt handlers.

Return Value

code_sel

A KN_SELECTOR in which the Kernel returns the selector for the current code segment.

get_data_selector

```
data_seg = KN_get_data_selector();
```

Data Type	Parameter
KN_SELECTOR	data_seg

Description

The **get_data_selector** system call is provided only with the Kernel's small-model interface library, (*c_call.lib*). It returns a selector for the current data segment.

Scheduling Category

Non-scheduling. Safe for use by interrupt handlers.

Return Value

data_seg

A KN_SELECTOR in which the Kernel returns the selector for the current data segment.

```
void KN_get_descriptor_attributes(table, descriptor,
                                  attribute_ptr);
```

Data Type	Parameter
KN_SELECTOR	table
KN_SELECTOR	descriptor
void	* attribute_ptr

Description

The **get_descriptor_attributes** system call returns a structure containing the attributes of a specified descriptor. The first byte of the structure (the access byte) determines the type of the descriptor. The remaining fields of the structure depend upon the type. For invalid descriptors, the attributes are undefined.

Scheduling Category

Non-scheduling. Safe for use by interrupt handlers.

Parameters

table A KN_SELECTOR for the descriptor table containing the descriptor whose attributes are to be returned. Possible values are:

- A KN_SELECTOR for the GDT alias
- A KN_SELECTOR for an LDT (a GDT descriptor that references an LDT)

descriptor
A KN_SELECTOR for the descriptor whose attributes are to be returned.

attribute_ptr
A pointer to an area where the Kernel returns the attributes of the descriptor. If the descriptor represents a segment, the attributes are returned in the KN_SEGMENT_ATTRIBUTES_STRUC structure. If the descriptor represents a gate, the attributes are returned in the KN_GATE_ATTRIBUTES_STRUC structure. When the system call returns, you can determine which structure to apply to the area by examining the first byte, which is the access byte in both structures.

When assigning memory for this area, you should allow enough room for the Kernel to return the largest possible structure (KN_SEGMENT_ATTRIBUTES_STRUC) unless you already know which type of descriptor will be returned.

When using the small model interface, the following structures have the format shown. For compact model, the `sel` field is not present because the base field is a full 48-bit pointer.

```
typedef struct {
    UINT_8                  access;
    UINT_8                  mode;
    UINT_8                * base;
    KN_SELECTOR             sel;
    UINT_32                 size;
} KN_SEGMENT_ATTRIBUTES_STRUC;

typedef struct {
    UINT_8                  access;
    UINT_8                  word_count;
    UINT_8                * base;
    KN_SELECTOR             sel;
} KN_GATE_ATTRIBUTES_STRUC;
```

Where:

access (applies to both structures) Indicates the type of descriptor this is. This field corresponds to the access byte of the descriptor (bits 7 through 15 of the descriptor's second doubleword). Refer to the *386 DX Programmer's Reference Manual* for more information about the access byte. There are several masks that you can apply to this field to obtain the access information. The flags are described at the end of this section.

mode A UINT_8 that indicates whether the segment is a 16-bit or a 32-bit segment. The following literals apply.

Literal	Meaning
KN_MODE_BIT	A mask for this field of the value.
KN_MODE_32	The segment is a 32-bit segment.
KN_MODE_16	The segment is a 16-bit segment.

base (applies to both structures) A pointer that specifies the beginning address of the segment. In small-model applications, this pointer is a 32-bit offset and the `sel` parameter indicates the selector for the segment from which the offset starts. In compact-model applications, this pointer is a full 48-bit pointer encompassing both the offset and the selector.

sel (applies to both structures) A KN_SELECTOR that identifies the segment from which the `base` is assumed to start. This field is only present for small-model applications.

size A UINT_32 indicating the size of the segment in bytes. The **get_descriptor_attributes** system call returns a 0 value in this field to indicate a 4G-byte segment.

word_count A UINT_8 indicating the number of words that are transferred from the calling procedure's stack to the new stack whenever a procedure makes an inter-level call using this gate.

The following flag literals can be used to get information from the `access` field.

KN_DATA_SEG
The descriptor represents a data segment. Data segments can have the following attributes:

Writable Determines if the data segment is writable.

Literal	Meaning
KN_DATA_D_WRITABLE_BIT	A mask for this field.
KN_WRITABLE	The data segment is writable.
KN_NOT_WRITABLE	The data segment is not writable.

Expand Determines if the data segment is expand-up or expand-down.

Literal	Meaning
KN_DATA_D_EXPAND_DIR_BIT	A mask for this field.
KN_EXPAND_DOWN	The segment is expand down.
KN_EXPAND_UP	The segment is expand up.

KN_EXEC_SEG

The descriptor represents an executable code segment. Executable segments can have the following attributes:

Readable Determines if the executable segment is readable.

Literal	Meaning
KN_EXEC_READABLE_BIT	A mask for this field.
KN_READABLE	The segment is readable.
KN_NOT_READABLE	The segment is not readable.

Conforming

Attributes of conforming segments are taken from the following literals:

Literal	Meaning
KN_EXEC_D_CONFORMING_BIT	A mask for this field.
KN_CONFORMING	The segment is conforming.
KN_NOT_CONFORMING	The segment is not conforming.

KN_SYS_SEG

The descriptor represents a system segment (a gate, a TSS, or an LDT). It can have the following attributes:

Literal	Meaning
KN_286_COMPATIBLE	The segment is 286-compatible.
KN_386_SPECIFIC	The segment is 386-specific.
KN_AVAILABLE_286_TSS	The segment is an available 286 Task State Segment.
KN_LDT	The segment is an LDT.
KN_BUSY_286_TSS	The segment is a busy 286 Task State Segment.
KN_286_CALL_GATE	The segment is a 286 call gate.
KN_286_OR_386_TASK_GATE	The segment is a 286 or 386 task gate.
KN_286_INTR_GATE	The segment is a 286 interrupt gate.
KN_286_TRAP_GATE	The segment is a 286 trap gate.
KN_AVAILABLE_386_TSS	The system is an available 386 Task State Segment.
KN_BUSY_386_TSS	The segment is a busy 386 Task State Segment.
KN_386_CALL_GATE	The segment is a 386 call gate.
KN_386_INTR_GATE	The segment is a 386 interrupt gate.
KN_386_TRAP_GATE	The segment is a 386 trap gate.

get_descriptor_attributes

In addition to the information that is specific to particular types of descriptors, the following masks apply to all descriptors:

Privilege Level

Determines the privilege level of the descriptor.

Literal	Meaning
KN_DPL_MASK	A mask for this field.
KN_DPL_0	The descriptor is privilege level 0.
KN_DPL_1	The descriptor is privilege level 1.
KN_DPL_2	The descriptor is privilege level 2.
KN_DPL_3	The descriptor is privilege level 3.

The value KN_DPL_SHIFT_COUNT can be used to shift the mask left to the proper bit position in the access byte.

Present

Determines if the data is present in memory.

Literal	Meaning
KN_PRESENT_BIT	A mask for this field.
KN_PRESENT	The data is present in memory.
KN_NOT_PRESENT	The data is not present in memory.

Accessed

Indicates whether the descriptor is currently accessed (that is, whether a selector for it is currently loaded into a segment register).

Literal	Meaning
KN_ACCESSED_BIT	A mask for this field.
KN_ACCESSED	The descriptor is currently accessed.
KN_NOT_ACCESSED	The descriptor is not currently accessed.

```
value = KN_get_interconnect(slot_number, register_number);
```

Data Type	Parameter
UINT_8	value
UINT_8	slot_number
UINT_16	register_number

Description

The **get_interconnect** system call retrieves the value in the specified interconnect register.

Scheduling Category

Non-scheduling. Safe for use by interrupt handlers.

Return Value

value A UINT_8 in which the Kernel returns the value of the specified interconnect register.

Parameters

slot_number

A UINT_8 that specifies the Multibus II card slot ID of the board on which the specified interconnect register is located. Values 0-20 indicate a slot on the Parallel System Bus. Values 24-29 indicate slots on the LBX II bus. A value of 31 indicates the current host. All other values are invalid.

register_number

A UINT_16 specifying the interconnect register to be read. Refer to the hardware manual for the particular board type to determine the proper register number.

get_PIT_interval

```
interval = KN_get_PIT_interval();
```

Data Type	Parameter
UINT_16	interval

Description

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.

Scheduling Category

Non-scheduling. Safe for use by interrupt handlers.

Parameters

interval

A UINT_16 specifying the current value of the PIT interval in milliseconds.

```
void KN_get_pool_attributes(pool, attributes_ptr);
```

Data Type		Parameter
KN_TOKEN		pool
KN_POOL_ATTRIBUTES_STRUC	*	attributes_ptr

Description

The **get_pool_attributes** system call provides information about the specified memory pool. The memory pool must previously be established with the **create_pool** system call.

Scheduling Category

Non-scheduling. Safe for use by interrupt handlers.

Parameters

pool A KN_TOKEN for the memory pool whose attributes are requested.

attributes_ptr

A pointer to a KN_POOL_ATTRIBUTES_STRUC in which the Kernel returns the attributes of the specified memory pool. The format of this structure is as follows:

```
typedef struct {
    UINT_32                 pool_size;
    UINT_32                 pool_available;
    UINT_32                 pool_largest;
} KN_POOL_ATTRIBUTES_STRUC;
```

Where:

pool_size The total number of bytes in the memory pool. That is, the size of the memory supplied when the memory pool was created.

pool_available The total number of bytes of available space in the memory pool.

pool_largest The number of bytes in the largest contiguous available space in the memory pool.

get_priority

```
priority = KN_get_priority(task);
```

Data Type	Parameter
UINT_16	priority
KN_TOKEN	task

Description

The **get_priority** system call returns the static priority of the specified task. This is either the priority that the application assigned to a task when the task was created or the priority the application set later with **set_priority**. Priority adjustment due to the task's ownership of a region has no effect on the result of this system call (that is, a task's static priority rather than its dynamic priority is always returned).

See also: Task Management, Priority Adjustment, *Installation and User's Guide*

Scheduling Category

Non-scheduling. Safe for use by interrupt handlers.

Return Value

priority A UINT_16 in which the Kernel returns the static priority of the specified task.

Parameters

task A KN_TOKEN for the task whose priority is requested.

```
slot = KN_get_slot();
```

Data Type	Parameter
UINT_8	slot

Description

The **get_slot** system call returns the highest priority interrupt slot currently in service. This identifies the highest priority interrupt source for which an interrupt handler is active (the handler has not yet sent an end-of-interrupt signal).

Scheduling Category

Non-scheduling. Safe for use by interrupt handlers.

Return Value

slot A UINT_8 indicating the highest priority active interrupt source. This value is an index into the IDT (that is; 1 indicates IDT slot 1, 2 indicates IDT slot 2, and so on). If no interrupt sources are currently active, the value 0 is returned.

get_time

```
time = KN_get_time();
```

Data Type	Parameter
UINT_64	time

Description

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. When the Kernel is initialized, the count is set to zero. Applications can set the count to any value with the **set_time** system call.

See also: Time Management, *Installation and User's Guide*

Scheduling Category

Non-scheduling. Safe for use by interrupt handlers.

Return Value

time A UINT_64 containing the current value of the system clock.

```
task = KN_initialize(configuration_ptr, area_ptr,
                     idle_task_area_ptr);
```

Data Type		Parameter
KN_TOKEN		task
KN_CONFIGURATION_DATA_STRUC	*	configuration_ptr
UINT_32	*	area_ptr
UINT_32	*	idle_task_area_ptr

Description

The **initialize** system call initializes the Kernel and transforms the calling program into the initial Kernel task. In the initialization you may set up the entry points for the task creation handler, the task deletion handler, the disaster handler, the task switch handler, and the priority change handler. If you are going to debug your code, you must call **initialize_RDS** before any other call. The **initialize** system call must be invoked before calling any other Kernel system call and before invoking any of the device managers supplied with the Kernel. The procedure that calls **initialize** becomes a Kernel task.

Before your program invokes this system call, the processor must be in protected mode.

The `configuration_ptr` parameter indicates the task handlers supplied by the application. For the calling format of the handlers, see Kernel Handlers, Chapter 4.

See also: Configuration, Chapter 5

Scheduling Category

Rescheduling. Unsafe for use by interrupt handlers.

Return Value

task A KN_TOKEN for the current running task (created from the caller).

initialize

Parameters

configuration_ptr

A pointer to a KN_CONFIGURATION_DATA_STRUC structure specifying the configuration parameters for the Kernel. The structure has the following format:

```
typedef struct {
   UINT_32                 time_slice;
   UINT_32                 real_time_fence;
   UINT_16                 priority;
   void                  * task_creation_handler_ptr;
   UINT_16                 task_creation_handler_ptr_fill;
   KN_FLAGS                task_creation_handler_flags;
   void                  * task_deletion_handler_ptr;
   UINT_16                 task_deletion_handler_ptr_fill;
   KN_FLAGS                task_deletion_handler_flags;
   void                  * task_switch_handler_ptr;
   UINT_16                 task_switch_handler_ptr_fill;
   KN_FLAGS                task_switch_handler_flags;
   void                  * priority_change_handler_ptr;
   UINT_16                 priority_change_handler_fill;
   KN_FLAGS                priority_change_handler_flags;
   void                  * disaster_handler_ptr;
   UINT_16                 disaster_handler_ptr_fill;
   KN_FLAGS                disaster_handler_flags;
} KN_CONFIGURATION_DATA_STRUC;
```

Where:

time_slice The value of the time slice (in clock ticks) to be used for time-sliced tasks.

real_time_fence

The priority value (between 0 and 511, with 0 being the highest priority) that determines which tasks are to be time-sliced. All tasks with priorities equal to or lower (numerically higher) than this value are time-sliced.

priority The priority (in the range 0-511) of the initial task created by this system call.

task_creation_handler_ptr

A pointer to a procedure used as the task creation handler. The Kernel calls this procedure whenever it creates a new task except for this initial task. If the pointer is null, the Kernel does not call a handler at task creation. This pointer is assumed to be relative to the caller's CS. The application must ensure that the mapping of this pointer to physical memory remains constant.

task_creation_handler_ptr_fill

task_deletion_handler_ptr_fill

task_switch_handler_ptr_fill

priority_change_handler_fill

disaster_handler_ptr_fill

Do not set this field. In small model, it is a placeholder. In compact model, it holds the selector of the preceding pointer.

task_creation_handler_flags

A KN_FLAGS for the task creation handler. The following literals apply to this flag:

Literal	Meaning
KN_HANDLER_CONVENTION_MASK	A mask for this field of the flag.
KN_CALL_NEAR	The handler is in the same subsystem as the Kernel.
KN_CALL_FAR	The handler is in a different subsystem from the Kernel's.

task_deletion_handler_ptr

A pointer to a procedure to be used as the task deletion handler. If null, the Kernel does not call a handler at task deletion. This pointer is assumed to be relative to the caller's CS. The application must ensure that the mapping of this pointer to physical memory remains constant.

task_deletion_handler_flags

A KN_FLAGS for the task deletion handler. The following literals apply to this flag:

Literal	Meaning
KN_HANDLER_CONVENTION_MASK	A mask for this field of the flag.
KN_CALL_NEAR	The handler is in the same subsystem as the Kernel.
KN_CALL_FAR	The handler is in a different subsystem from the Kernel's.

task_switch_handler_ptr

A pointer to a procedure to be used as the task switch handler. If null, the Kernel does not call a handler at task switch time. This pointer is assumed to be relative to the caller's CS. The application must ensure that the mapping of this pointer to physical memory remains constant.

task_switch_handler_flags

A KN_FLAGS for the task switch handler. The following literals apply to this flag:

Literal	Meaning
KN_HANDLER_CONVENTION_MASK	A mask for this field of the flag.
KN_CALL_NEAR	The handler is in the same subsystem as the Kernel.
KN_CALL_FAR	The handler is in a different subsystem from the Kernel's.

priority_change_handler_ptr

A pointer to a procedure to be used as the priority change handler. If null, the Kernel does not call a handler at priority change time. This pointer is assumed to be relative to the caller's CS. The application must ensure that the mapping of this pointer to physical memory remains constant.

priority_change_handler_flags

A KN_FLAGS for the priority change handler. The following literals apply to this flag:

Literal	Meaning
KN_HANDLER_CONVENTION_MASK	A mask for this field of the flag.
KN_CALL_NEAR	The handler is in the same subsystem as the Kernel.
KN_CALL_FAR	The handler is in a different subsystem from the Kernel's.

disaster_handler_ptr

A pointer to a procedure to be used as the disaster handler. If null, the Kernel executes an INT3 instruction when a disaster occurs. This pointer is assumed to be relative to the caller's CS. The application must ensure that the mapping of this pointer to physical memory remains constant.

disaster_handler_flags

A KN_FLAGS for the disaster handler. The following literals apply to this flag:

Literal	Meaning
KN_HANDLER_CONVENTION_MASK	A mask for this field of the flag.
KN_CALL_NEAR	The handler is in the same subsystem as the Kernel.
KN_CALL_FAR	The handler is in a different subsystem from the Kernel's.

area_ptr

A pointer to an area in which the Kernel stores the initial task's task state. This area must be at least KN_TASK_SIZE bytes long, and it must be aligned on a four-byte boundary. If you are using the numeric coprocessor, the area must be at least KN_TASK_SIZE plus KN_387_SAVE_AREA_SIZE bytes long. The Kernel applies a KN_TASK_STATE structure on this area when it maintains the state.

KN_TASK_SIZE

The minimum number of bytes of memory required for a task object.

KN_TASK_STATE

A structure that can overlay a task state to provide access to the individual fields. See the **create_task** system call for the KN_TASK_STATE structure.

idle_task_area_ptr

A pointer to the area in which the Kernel stores the idle task's task state. This area must be at least KN_TASK_SIZE plus KN_387_SAVE_AREA_SIZE bytes long, and it must be aligned on a four byte boundary. The Kernel applies a KN_TASK_STATE structure on this area when it maintains the state. See the **create_task** system call for the KN_TASK_STATE structure.

```
void KN_initialize_console(configuration_ptr);
```

Data Type		Parameter
KN_CONSOLE_CONFIGURATION_STRUC	*	configuration_ptr

Description

The **initialize_console** system call provides initialization for the 82530 Serial Communication Controller device, used for character I/O functions. This system call should be invoked before starting any character I/O. The 82530 Serial Communication Controller is not used in interrupt driven mode.

See also: Configuration, Chapter 5
Device Management, *Installation and User's Guide*

Scheduling Category

Unsafe.

Parameters

configuration_ptr

A pointer to a data structure specifying the configuration of the I/O device. The configuration data structure is not modified, and thus may reside in ROM. The format of this data structure is:

```
typedef struct{
   UINT_8                  type;
   UINT_32                 data_port;
   UINT_32                 control_port;
   UINT_32                 clock_frequency_hi;
   UINT_32                 clock_frequency;
   UINT_32                 baud_rate;
   UINT_32                 interrupt_level;
}KN_CONSOLE_CONFIGURATION_STRUC;
```

Where:

type The type of the I/O device. KN_82530A_USART supports channel A, and KN_82530B_USART supports channel B.

data_port Data port address for the I/O device.

control_port Control port address for the I/O device.

clock_frequency_hi
High word of the clock frequency in hertz of the I/O device.

clock_frequency
Low word of the clock frequency in hertz of the I/O device.

baud_rate Character transmission rate allowed by the I/O device.

interrupt_level
This parameter is not used.

Table 2-2 gives initialization values for some Intel boards. For other boards, refer to the appropriate hardware reference manual.

Table 2-2. Initialization Values for the 82530 Device

Board	Data Port	Control 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.

```
void KN_initialize_interconnect(configuration_ptr);
```

Data Type		Parameter
KN_INTERCONNECT_STRUC	*	configuration_ptr

Description

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.

See also: Configuration, Chapter 5

Scheduling Category

Non-scheduling. Safe for use by interrupt handlers.

Parameters

configuration_ptr

A pointer to a KN_INTERCONNECT_STRUC structure specifying configuration information necessary to access interconnect space. The structure has the following format:

```
typedef struct {
    UINT_16                 address_port;
    UINT_16                 data_port;
    UINT_8                  port_separation;
} KN_INTERCONNECT_STRUC;
```

Where:

address_port The I/O port used to specify the address for interconnect operations.

data_port The first of the I/O ports used to read and write data in interconnect operations.

initialize_interconnect

port_separation

The number of bytes separating the data ports used to access interconnect space.

```
void KN_initialize_LDT(descriptor, base_ptr, entries);
```

Data Type		Parameter
KN_SELECTOR		descriptor
void	*	base_ptr
UINT_16		entries

Description

The **initialize_LDT** system call sets up a specified descriptor in the GDT as a local descriptor table (LDT) descriptor. It also initializes all descriptors in the new LDT with null descriptors. The previous contents of the GDT descriptor are overwritten.

The calling task has direct control over all fields in the descriptor for the LDT, except the `present bit` and the `DPL` field. The Kernel initializes the LDT descriptor with `DPL` = 0 and present = 1.

If another task invokes the **get_descriptor_attributes** system call on the same descriptor entry while the **initialize_LDT** system call is underway, that task will get either the old or new attributes of the descriptor, but not a combination of the two. Once the new descriptor for the LDT is in place, all the entries in the LDT are also initialized.

Scheduling Category

Non-scheduling. Safe for use by interrupt handlers.

initialize_LDT

Parameters

descriptor

A KN_SELECTOR for the GDT descriptor to be initialized.

base_ptr

A pointer to the area to be used for the LDT. This area must be large enough to contain the number of descriptors requested. The new table contains:

entries * KN_DESCRIPTOR_SIZE bytes.

KN_DESCRIPTOR_SIZE

The number of bytes required for each descriptor in a descriptor table. Multiply this value by the number of entries in a descriptor table to determine the amount of memory required by the table.

entries

A UINT_16 specifying the number of descriptors in the new LDT.

```
void KN_initialize_message_passing(configuration_ptr,
                                   working_storage_ptr);
```

Data Type		Parameter
KN_MP_CONFIGURATION_STRUC	*	configuration_ptr
void	*	working_storage_ptr

Description

The **initialize_message_passing** system call initializes the message passing module. You must invoke this system call before using the message passing system calls. You must invoke the **initialize_interconnect** call prior to making this call. The interconnect code initializes the host ID that the message passing code will use when dealing with the MPC.

Before initializing message passing, you must initialize the PIC and the PIT, in that order. If these steps are out of order, or interconnect is not initialized, message passing will function incorrectly.

Tasks that use the message passing module must not change the task descriptor in their TSS.

See also: Configuration, Chapter 5

Scheduling Category

Unsafe for use by interrupt handlers.

Parameters

configuration ptr

A pointer to a KN_MP_CONFIGURATION_STRUC area containing the configuration information for the message passing module. The structure has the following format:

```
typedef struct {
/* Data Link Configuration */
    UINT_8                  max_protocol_id;
    UINT_16                 number_data_chain_elements;
    UINT_32                 PSB_MPC_port;
    UINT_8                  MPC_config;
    UINT_8                  MPC_int_slot;
    UINT_8                  MPC_duty_cycle;
    UINT_32                 delay_scale;
    UINT_8                  si_failsafe_timer;
    UINT_8                  so_failsafe_timer;
/* DMA Configuration */
    UINT_32                 DMA_port;
    UINT_8                  auxiliary_DMA_support;
    UINT_32                 auxiliary_DMA_port;
    UINT_8                  in_channel;
    UINT_8                  out_channel;
    UINT_16                 data_link_task_priority;
    UINT_32                 data_link_rcv_queue_size;
    UINT_8                  number_of_data_link_retries;
/* Transport Configuration */
    UINT_16                 transport_task_priority;
    UINT_32                 transport_mbx_queue_size;
    UINT_32                 attached_mbx_hash_table_size;
    UINT_32                 transaction_hash_table_size;
    UINT_32                 number_attached_mbxs;
} KN_MP_CONFIGURATION STRUC;
```

Where:

max_protocol_id
: The maximum value of any protocol handler protocol ID that can be attached. The default value is 4. Non-Intel protocol handlers must use the range 80h-FFh.

number_data_chain_elements
: The maximum number of data chain elements that can be constructed. The default value is 64.

PSB_MPC_port
: The I/O port address of the Multibus II MPC device. The default value is 0.

MPC_config The value of the MPC configuration register. The default value is 8AH.

MPC_int_slot

The interrupt IDT slot of the MPC MINT interrupt. The default value is 58.

MPC_duty_cycle

Used to determine the MPC buffer grant duty cycle to allocate the local bus bandwidth for the ADMA or the 82380/82370 Integrated System Peripheral.

The following values specify the percentage of the local bus bandwidth devoted to message passing. The default value is 3.

0	100%
1	75%
2	50%
3	25%
4	12%
5	6%
6	3%
7	1%

Note that DMA bus bandwidth can affect CPU bus bandwidth and hence processor interrupt latency and execution times.

delay_scale Used to set delays in timing-dependent operations. The larger the delay scale, the longer the Kernel waits before checking the status of an operation. If the processor or MPC execution rates change in future hardware products, you can change the delay scale to make the message passing module function correctly. The default value is 3.

si_failsafe_timer

Used on buffer grant messages to control the fail-safe counter. When set (`si_failsafe_timer = 80H`), the counter operation is disabled. When not set (`si_failsafe_timer = 00H`), the counter operation is enabled. The default value is 80H.

so_failsafe_timer

Used on buffer request messages to control the fail-safe counter. When set (`so_failsafe_timer = 80H`), the counter operation is disabled. When not set (`so_failsafe_timer = 00H`), the counter operation is enabled. The default value is 80H.

DMA_port The I/O port address of the ADMA (82258) component or the 82380/82370 Integrated System Peripheral. The default value is 200H.

auxiliary_DMA_support
A Boolean value indicating whether burst mode support is desired. This value should be TRUE when using the iSBC 386/133, 486/125, 386/020 or 486/133 boards and burst mode is desired; otherwise it should be FALSE. The default value is FALSE.

auxiliary_DMA_port
The I/O port address of the high speed DMA component of an iSBC 386/133, 486/125, 386/020 or 486/133 board. The default value is 0300H.

in_channel The ADMA or 82380/82370 Integrated System Peripheral channel number used for Multibus II solicited input. The default value is 2.

out_channel The ADMA or 82380/82370 Integrated System Peripheral channel number used for Multibus II solicited output. The default value is 3.

data_link_task_priority
The priority of the internal Kernel task that handles the in channel and out channel DMA. The default value is 4.

data_link_rcv_queue_size
The size, in number of queue elements, of the data link mailbox used by the data link internal Kernel task. The default value is 16.

number_of_data_link_retries
The number of times the message passing software attempts to send an unsolicited message when the hardware status upon transmission indicates an E_RETRY_EXPIRED error. The default value is 0.

transport_task_priority
The priority of the transport protocol internal task. This priority should be less (numerically greater) than the solicited channel tasks. The default value is 5.

transport_mbx_queue_size
The size, in number of queue elements, of the transport mailbox used by the transport protocol internal Kernel task. The default value is 16.

attached_mbx_hash_table_size
The size, in number of buckets, of the hash table used to keep track of attached receive mailboxes. The default value is 256.

transaction_hash_table_size
The size, in number of buckets, of the hash table used to keep track of outstanding RSVP transactions and send next fragment messages. The default value is 256.

number_attached_mbxs
The number of mailboxes that can be attached as transport protocol receive mailboxes. The default value is 64.

working_storage_ptr
A pointer to an area of memory to be used as working storage by the message passing module. The exact size of this area can be determined from the **mp_working_storage_size** system call.

initialize_NDP

```
void KN_initialize_NDP(configuration_ptr);
```

Data Type		Parameter
KN_NDP_CONFIGURATION_STRUC	*	configuration_ptr

Description

The **initialize_NDP** system call initializes the Kernel's optional Numeric Coprocessor Module. This module provides an Interrupt 7 handler that is invoked whenever a coprocessor instruction is executed. If there has been a task switch, the handler saves the coprocessor state for the task that previously invoked the coprocessor, and restores the coprocessor state for the task that is accessing the coprocessor now. If you include the coprocessor module, you must invoke the **initialize_NDP** system call before using coprocessor instructions, but after invoking the **initialize** system call.

Applications can include the support module to avoid implementing their own code that saves and restores the coprocessor state. To use this module, tasks must include an area of size KN_387_SAVE_AREA_SIZE in the task state immediately following the area allocated for Kernel task management. The support module uses this area to save the state of the coprocessor for each task.

By default, the support module initializes the coprocessor state area for the current task and every task created thereafter in the application. Initialization occurs when the task is created. This initialization places reasonable values into the coprocessor save area so that no errors occur the first time the task executes a coprocessor instruction. However, by setting a flag in the **initialize_NDP** system call, you can prevent the Kernel from initializing the coprocessor save areas. By preventing initialization, you can selectively initialize save areas for only the tasks that use the coprocessor.

If no initialization handler is specified, the Kernel initializes only one save area, the save area for the calling task. If other tasks need to use the coprocessor, they must set up and initialize coprocessor save areas of their own, immediately following the memory used for the task state. You can determine the correct initialization values to use in other tasks by saving the coprocessor state immediately after invoking the **initialize_NDP** system call. Perform the following steps:

1. Disable interrupts, so that you can perform the entire operation without being interrupted by other tasks.
2. Call **initialize_NDP** to initialize the Numeric Coprocessor Module.
3. Execute the FSAVE instruction to save the coprocessor state in an area of memory you can pass on to other tasks.
4. Set the TS (task switch) flag in the CR0 register. This tells the CPU to cause an interrupt 7 the first time a numeric coprocessor instruction occurs after a task switch. The interrupt 7 invokes the Kernel's coprocessor module so that it can save and restore the coprocessor state.
5. Enable interrupts and continue processing.

Scheduling Category

Non-scheduling. Safe for use by interrupt handlers.

initialize_NDP

Parameters

configuration_ptr

A pointer to a KN_NDP_CONFIGURATION_STRUC structure containing configuration data for the coprocessor. The format of this structure is as follows:

```
typedef struct {
   UINT_8                  ndp_type;
   KN_FLAGS                flags;
} KN_NDP_CONFIGURATION_STRUC;
```

Where:

ndp_type Indicates the type of coprocessor. The only value currently supported is KN_387_NDP.

flags A KN_FLAGS whose bit structure specifies characteristics of the coprocessor support. Currently only one flag is defined, which indicates whether or not the coprocessor manager initializes the coprocessor save areas in tasks. The following literals apply to this flag:

Literal	Meaning
KN_387_HANDLER_MASK	A mask for this field of the flag.
KN_387_NO_HANDLER	Don't use the default 387 initialization handler for this task.
KN_387_DEFAULT_HANDLER	Use the default 387 initialization handler for this task.

If the default initialization handler is specified for this flag, the Kernel initializes a coprocessor save area in the calling task, and it also initializes coprocessor save areas for all subsequently created tasks during task creation. With the default initialization handler in effect, all tasks must allocate space at the end of the task state for a coprocessor save area.

```
void KN_initialize_PICs(configuration_ptr);
```

Data Type		Parameter
KN_PIC_CONFIGURATION_STRUC	*	configuration_ptr

Description

The **initialize_PICs** system call lets you specify the configuration of the 8259A or 82380/82370 PICs in the system. The system call also initializes the PICs so that interrupt processing can begin. After this call, the PICs are set up as if all slots had been masked by the **mask_slot** system call and the **new_masks** system call had been invoked with a parameter of 255.

See also: Configuration, Chapter 5

Scheduling Category

Non-scheduling. Safe for use by interrupt handlers.

Parameters

configuration_ptr

A pointer to a KN_PIC_CONFIGURATION_STRUC structure specifying the PIC configuration. This structure consists of an array of nine KN_PIC_INDIV_STRUC structures, each of which specifies configuration information for an individual PIC. The first structure specifies the characteristics of the master PIC. The remaining eight specify the slave PICs, with 0 first, then 1, and so on. If a slave PIC is not present, set its `first_slot` field to zero.

The format of each KN_PIC_INDIV_STRUC data structure is as follows:

```
typedef struct{
    UINT_16                 port_address;
    UINT_8                  port_separation;
    UINT_8                  first_slot;
    UINT_8                  sources_map;
    UINT_8                  type;
    UINT_8                  mode;
} KN_PIC_INDIV_STRUC;
```

Where:

port_address The first I/O port used to program the PIC.

port_separation
The distance in bytes between consecutive I/O ports used to program the PIC.

first_slot The number of the entry in the interrupt descriptor table (IDT) corresponding to the highest priority (lowest numeric) PIC input. The seven IDT entries that follow correspond to the remaining seven PIC inputs.

sources_map A bit map indicating which of the PIC inputs are connected to interrupt sources. Bit 0 corresponds to input 0, bit 1 to input 1, etc. A one in a bit position indicates the corresponding input is connected. A zero in a bit position indicates either that the corresponding input is not connected or that the input is connected to a slave PIC.

type Indicates the type of PIC. Currently the values KN_8259A_PIC and KN_82380_PIC are supported. If you use the 82380/82370 Integrated System Peripheral, be sure to call out the pic_8259 module in the 82380/82370 library during the bind sequence.

mode Indicates whether the PIC should be programmed for edge or level mode. If the PIT manager module is used, the PIC that is connected to the PIT's timer must be programmed for edge mode. Use the literal KN_EDGE_MODE or KN_LEVEL_MODE to indicate the mode.

```
void KN_initialize_PIT(configuration_ptr);
```

Data Type		Parameter
KN_PIT_CONFIGURATION_STRUC	*	configuration_ptr

Description

The **initialize_PIT** system call lets you specify configuration information about the 8254 or 82380/82370 timer. It initializes the timer to supply clock ticks to the Kernel at the specified interval. The PIT is not actually started by this system call; the **start_PIT** system call must be used for that purpose.

You must call **initialize_PIC** before calling **initialize_PIT**.

See also: Configuration, Chapter 5

Scheduling Category

Non-scheduling. Safe for use by interrupt handlers.

Parameters

configuration_ptr

A pointer to a KN_PIT_CONFIGURATION_STRUC data structure specifying the PIT configuration. The format of this data structure is as follows:

```
typedef struct {
    UINT_16                 port_address;
    UINT_8                  port_separation;
    UINT_16                 in_frequency;
    UINT_8                  slot;
    UINT_8                  type;
    UINT_8                  timer_out;
} KN_PIT_CONFIGURATION_STRUC;
```

Where:

port_address The port address of the first I/O port to be used in programming the PIT.

initialize_PIT

slot A UINT_8 indicating the IDT slot number to which the PIT handler is assigned.

port_separation
The distance in bytes between I/O ports used in programming the PIT. The default value for the 8254 device is 2. The default value for the 82370/82380 device is 1.

in_frequency The input frequency to the PIT, expressed in kHz.

type The type of the PIT. Currently, the values KN_8254_PIT and KN_82380_PIT are supported. If you use the 82380/82370 Integrated System Peripheral, be sure to call out the pit_8254 module in the 82380/82370 library during the bind sequence.

timer_out Indicates which timer on the PIT is to be used. The values supported are 0, 1, 2, and 3. The default timer for the 8254 device is timer 0. The default timer for the 82370/82380 device is timer 3.

```
status = KN_initialize_RDS(configuration_ptr)
```

Data Type		Parameter
KN_STATUS		status
KN_RDS_STRUC	*	configuration_ptr

Description

The **initialize_RDS** system call sets values used by the Kernel part of the debugger software. This should be the first executable statement in the application. Only the code following this call can be debugged.

The **initialize_RDS** system call also sets up the trace queue for the **ktrace** debugger command, and does device and console setup.

For the System V/iRMK environment, 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 the Soft_Scope III debugger.
- 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 other than `iM` or `on`, RDS will not be initialized.

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.

initialize_RDS

See also: Configuration, Chapter 5
Table of `cc_console` device names, *Installation and User's Guide*

Scheduling Category

Rescheduling. Unsafe for use by interrupt handlers.

Return Value

status A KN_STATUS indicating the result of the call. Values are:

Literal	Meaning
E_OK	Either RDS successfully initialized or RDS is not present in the application.
E_NOT_CONFIGURED	RDS did not initialize. The application does not break to the Monitor, and Soft-Scope III will not be able to synchronize.

Parameter

configuration_ptr

A pointer to the following structure containing initialization values. If you don't use the **ktrace** and **kmsgq** debugger commands, the fields `dl_trace_queue_size` through `rcv_buf_ptr` may be null. To enable full trace capability, all trace-related values should be set.

```
typedef struct {
        UINT_32                 reserved1 [8];
        UINT_32                 dl_trace_queue_size;
        UINT_8                  reserved2 [14];
        UINT_8              *   xmit_buf_ptr;
        UINT_16                 xmit_buf_ptr_fill;
        UINT_8              *   rcv_buf_ptr;
        UINT_16                 rcv_buf_ptr_fill;
        UINT_32                 debug_flags;
        UINT_32                 MPC_base_address;
        UINT_32                 MPC_port_separation;
        UINT_32                 IC_base_address;
        UNIT_32                 IC_port_separation;
        UINT_8              *   default_cc_stdin;
        UINT_16                 default_cc_stdin_fill;
        UINT_8              *   default_cc_stdout;
        UINT_16                 default_cc_stdout_fill;
        UINT_8              *   default_cc_stderr;
        UINT_16                 default_cc_stderr_fill;
} KN_RDS_STRUC;
```

Where:

reserved[n] Set to zero.

dl_trace_queue_size The maximum number of data-link messages that can be held in the trace queue of the **ktrace** debugger command. If null, no trace can be maintained.

xmit_buf_ptr A pointer to a user buffer where traced *transmit* messages are stored. This buffer must be large enough to hold all transmitted messages stored in the queue. If the pointer is null, transmitted messages are not stored.

xmit_buf_ptr_fill Do not set this field. In small model, it is a placeholder. In compact model, it holds the selector of the preceding pointer.

rcv_buf_ptr A pointer to a user buffer where traced received messages are stored. This buffer must be large enough to hold all received messages stored in the queue. If the pointer is null, received messages are not stored.

rcv_buf_ptr_fill Do not set this field. In small model, it is a placeholder. In compact model, it holds the selector of the preceding pointer.

debug_flags Reserved. Set to zero.

MPC_base_address The I/O port address of the MPC device. The value is 0 for Intel boards.

MPC_base_port_separation
The number of bytes separating the MPC ports. The default value is 4.

IC_base_address The I/O port address of interconnect operations. Typically, it is `MPC_base_address` + 0x30.

IC_port_separation The number of bytes separating the data ports used to access interconnect space. The default value is 4.

default_cc_stdin A pointer to a null terminated string of the device name that represents the default standard input. The System V/iRMK BPS parameter `CC_console` overrides this field.

default_cc_stdin_fill Do not set this field. In small model, it is a placeholder. In compact model, it holds the selector of the preceding pointer.

default_cc_stdout A pointer to a null terminated string of the device name that represents the default standard output. The System V/iRMK BPS parameter `CC_console` overrides this field.

default_cc_stdout_fill Do not set this field. In small model, it is a placeholder. In compact model, it holds the selector of the preceding pointer.

default_cc_stderr A pointer to a null terminated string of the device name that represents the default standard error console. The System V/iRMK BPS parameter `CC_console` overrides this field.

default_cc_stderr_fill Do not set this field. In small model, it is a placeholder. In compact model, it holds the selector of the preceding pointer.

initialize_stdio

```
void initialize_stdio();
```

Description

The **initialize_stdio** system call prepares the Kernel for using standard I/O functions by making the functions reentrant. Although stdio initialization does not have to be done at Kernel initialization, the **initialize_stdio** system call must be called before any stdio functions are used.

NOTE

The **initialize_stdio** system call does not have a **KN_** prefix.

See also: Chapter 3
kstdio libraries, *Installation and User's Guide*

Scheduling Category

Non-scheduling. Safe for use by interrupt handlers.

```
void KN_initialize_subsystem(configuration_ptr);
```

Data Type	Parameter
KN_SUBSYSTEM_CONFIG	* configuration_ptr

Description

All application subsystems that run outside the Kernel's subsystem must invoke the **initialize_subsystem** system call before any other system call. There is one exception to this rule. One of the subsystems must first invoke the Kernel **initialize** system call (and may invoke **initialize_RDS**). The **initialize** system call should execute before any of the **initialize_subsystem** system call invocations.

The **initialize_subsystem** system call has different effects, depending on whether it is called from a "direct call" model or a "gated call" model.

- Direct call. An application using the direct call model is in the same subsystem as the Kernel. In this case, the **initialize_subsystem** system call has no effect. If you never intend to port your application to the gated call model, you can eliminate the **initialize_subsystem** system call.
- Gated call. The **initialize_subsystem** system call saves the configuration data. The Kernel then uses the configuration data to transform returned pointer values into values accessible by the application.

  Normally, the Kernel returns pointers relative to the Kernel's subsystem (that is, the pointer has the selector for the Kernel's data segment). However, if you've issued an **initialize_subsystem** system call, the Kernel translates the pointer into one containing the selector specified in KN_SUBSYSTEM_CONFIG.

Scheduling Category

Non-scheduling. Safe for interrupt handlers.

initialize_subsystem

Parameters

configuration_ptr

A pointer to a KN_SUBSYSTEM_CONFIG data structure specifying the subsystem configuration. The format of this data structure is as follows:

```
typedef struct {
   KN_SELECTOR          subsystem_rel_sel;
   KN_FLAGS             subsystem_flags;
} KN_SUBSYSTEM_CONFIG;
```

Where:

subsystem_rel_sel

A KN_SELECTOR that the Kernel uses for pointer translation. For example, in the small model, you can obtain the value used for this parameter with the **get_code_selector** or the **get_data_selector** system call. If you supply a null for this parameter, the Kernel performs no translation.

subsystem_flags

A KN_FLAGS specifying whether or not the user segment is restricted to 4 gigabytes. The following literals apply to this flag:

Literal	Meaning
KN_RELAX_LIMIT_MASK	The mask for this field of the flag.
KN_RELAX_LIMIT	The limit in the descriptor pointed to by the pointer's selector is determined by the Kernel. The Kernel sets the limit to 4 gigabytes. This is one way of ensuring that you don't go beyond a segment limit.
KN_NO_RELAX_LIMIT	The limit in the descriptor pointed to by the pointer's selector is not changed by the Kernel. It is your responsibility to ensure that you don't go beyond a segment limit. This is the default.

```
abs_ptr = KN_linear_to_ptr(linear_address);
```

Data Type		Parameter
UINT_8	*	abs_ptr
UINT_32		linear_address

Description

Given a linear address, the **linear_to_ptr** system call generates a pointer that refers to the same location. This system call is useful for referencing memory or for passing configuration parameters to Kernel interfaces. For example, linear addresses might be specified during the configuration of a system or as input from a human interface.

Scheduling Category

Non-scheduling. Safe for use by interrupt handlers.

Return Value

abs_ptr

A pointer that refers to the specified linear address.

Parameters

linear_address

A UINT_32 containing a linear address to be translated into a pointer.

local_host_ID

```
host_ID = KN_local_host_ID();
```

Data Type	Parameter
KN_HOST_ID	host_ID

Description

The **local_host_ID** system call returns the host ID of the board on which the application is executing.

During initialization, the Kernel determines the host ID by reading the Multibus II Interconnect Space host ID record. If the host ID field is non-zero, the Kernel uses that value as the local host ID. If the field is zero, or if the record does not exist, the Kernel uses the host's slot ID as the host ID. If the slot ID is used, the Kernel fills in the Interconnect Space host ID record (if it exists) with that slot ID, so that the Kernel's local host ID and the Interconnect Space host ID are equal.

Scheduling Category

Non-scheduling. Safe for use by interrupt handlers.

Return Value

host_ID

A KN_HOST_ID containing the local host ID.

```
void KN_mask_slot(slot);
```

Data Type	Parameter
UINT_8	slot

Description

The **mask_slot** system call causes the PIC(s) to mask interrupts that occur on the specified interrupt line until a corresponding **unmask_slot** system call is invoked on the slot.

See also: Masks and Interrupt Sources, *Installation and User's Guide*

Scheduling Category

Non-scheduling. Safe for use by interrupt handlers.

Parameters

slot A UINT_8 indicating the entry (slot number) in the IDT to be masked. This slot must be one that was assigned to the PIC with the **initialize_PICs** system call.

mp_working_storage_size

```
storage_size = KN_mp_working_storage_size(configuration_ptr);
```

Data Type		Parameter
UINT_32		storage_size
KN_MP_CONFIGURATION_STRUC	*	configuration_ptr

Description

The **mp_working_storage_size** system call calculates the size in bytes of the area of memory to be used by the message passing module. All data chain elements must be in the GDT or same LDT table. The calculation is based on the same configuration information used in the **initialize_message_passing** system call. Use the storage size calculated to create an area of the size needed for **initialize_message_passing**.

Scheduling Category

Non-scheduling. Safe for use by interrupt handlers.

Return Value

storage_size

A UINT_32 that contains the calculated size of the working storage area in bytes.

Parameters

configuration_ptr

A pointer to a KN_MP_CONFIGURATION_STRUC area containing the configuration information for the message passing module. See **initialize_message_passing** for the structure.

new_masks

```
void KN_new_masks(slot);
```

Data Type	Parameter
UINT_8	slot

Description

The **new_masks** system call changes the masking of interrupt sources based upon a specified entry (slot number) in the IDT. All interrupt lines less important than the one corresponding to the specified slot are masked out. The specified interrupt line and all lines more important are unmasked, as long as they have been unmasked by the **unmask_slot** system call. Higher priority sources are associated with lower IDT slot numbers.

See also: Masks and Interrupt Sources, *Installation and User's Guide*

Scheduling Category

Non-scheduling. Safe for use by interrupt handlers.

Parameters

slot A UINT_8 specifying the entry (slot number) in the IDT for the new mask interrupt line.

```
void KN_null_descriptor(table, descriptor);
```

Data Type	Parameter
KN_SELECTOR	table
KN_SELECTOR	descriptor

Description

The **null_descriptor** system call overwrites a specified descriptor with a null descriptor. The null descriptor is a descriptor with a DPL of 3 that points to an unused one-byte, read-only data location within the Kernel. A selector for a null descriptor can be loaded into a segment register without generating a fault, but almost any attempt to use a null descriptor for referencing memory will generate a fault.

Scheduling Category

Non-scheduling. Safe for use by interrupt handlers.

Parameters

table A KN_SELECTOR for the descriptor table containing the descriptor to be nulled.

descriptor
A KN_SELECTOR for the descriptor to be nulled.

ptr_to_linear

```
linear_address = KN_ptr_to_linear(table, sel, ptr); (SMALL)

linear_address = KN_ptr_to_linear(table, ptr);       (COMPACT)
```

Data Type	Parameter
UINT_32	linear_address
KN_SELECTOR	table
KN_SELECTOR	sel
void *	ptr

Description

The **ptr_to_linear** system call returns the linear address referred to by a given pointer. The descriptor table in which the pointer is valid must be supplied by the caller.

Scheduling Category

Non-scheduling. Safe for use with interrupt handlers.

Return Value

linear_address

A UINT_32 containing the linear address referred to by the specified pointer.

Parameters

table A KN_SELECTOR for the descriptor table containing the segment descriptor for the specified pointer.

sel A KN_SELECTOR indicating the segment from which the `ptr` parameter is offset. This parameter is present only in the small-model interface library (*c_call.lib*). In the compact-model interface, the `ptr` parameter includes both the selector and the offset.

ptr A pointer to the location whose linear address is requested. In the small-model interface, this parameter is a 32-bit offset. In the compact-model interface, the parameter is a full pointer, containing both a selector and an offset.

```
status = KN_receive_data(mailbox, data_ptr, length_ptr,
                         time_limit);
```

Data Type		Parameter
KN_STATUS		status
KN_TOKEN		mailbox
void	*	data_ptr
UINT_32	*	length_ptr
UINT_32		time_limit

Description

The **receive_data** system call requests a message from the specified mailbox. If the mailbox currently contains at least one message, the oldest message (or the latest high-priority message) is removed from the message queue and returned to the calling task. If there are no messages queued at the mailbox and the task is willing to wait, it is put to sleep and queued at the mailbox for the amount of time it is willing to wait. The task is queued at the mailbox in either FIFO or priority-based order, depending on the type of mailbox. The task will be awakened by one of three events:

- The task is at the head of the mailbox queue and another task invokes **send_data** on the mailbox.
- The number of clock ticks specified by the task expires.
- The mailbox is deleted.

When receiving (using the **receive_data** system call) and sending (using the **send_data** system call) mailbox messages, interrupts are disabled for the time it takes to copy the message. Hence, a large data transfer via mailboxes may affect interrupt latency.

Scheduling Category

Blocking. Use with caution in interrupt handlers.

receive_data

Return Value

status A KN_STATUS indicating the result of the call. Values are:

Literal	Meaning
E_OK	Indicates that the task received a message.
E_TIME_OUT	Indicates that the time limit expired.
E_NONEXIST	Indicates that the mailbox was deleted while the task was waiting.

NOTE

If the mailbox is deleted before the task begins waiting, the call will not return the E_NONEXIST message. It is the responsibility of the application to ensure that the mailbox has not been deleted before a task begins waiting.

Parameters

mailbox

A KN_TOKEN for the mailbox from which the message is requested.

data_ptr

A pointer to an area where the message is to be placed. The size of this area must be equal to the message size parameter specified when the mailbox was created.

length_ptr

A pointer to a UINT_32 where the Kernel specifies the length (in bytes) of the message it returns.

time_limit

A UINT_32 specifying the number of clock ticks the caller is willing to wait for a message. Values are:

Literal	Meaning
KN_DONT_WAIT	Indicates the task will not wait at all.
KN_WAIT_FOREVER	Indicates that the task is willing to wait indefinitely.
UINT_32 value	Indicates that the task will wait for the specified number of clock ticks.

```
status = KN_receive_unit(semaphore, time_limit);
```

Data Type	Parameter
KN_STATUS	status
KN_TOKEN	semaphore
UINT_32	time_limit

Description

The **receive_unit** system call requests a unit from the specified semaphore. If the semaphore currently 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 task will be awakened by one of three events:

- The task is at the head of the semaphore queue and another task invokes **send_unit** on this semaphore.
- The number of clock ticks specified by the task expires.
- The semaphore is deleted.

Scheduling Category

Blocking. Use with caution in interrupt handlers.

Return Value

status A KN_STATUS indicating the result of the call. Values are:

Literal	Meaning
E_OK	Indicates that the task received the requested unit.
E_TIME_OUT	Indicates that the time limit expired.
E_NONEXIST	Indicates the semaphore was deleted while the task was waiting.

NOTE

If the semaphore is deleted before the task begins waiting, the call will not return the E_NONEXIST message. It is the responsibility of the application to ensure that the semaphore has not been deleted before a task begins waiting.

Parameters

semaphore

A KN_TOKEN for the semaphore from which a unit is requested.

time_limit

A UINT_32 indicating the number of clock ticks the calling task is willing to wait for the unit. Possible values are:

Literal	Meaning
KN_DONT_WAIT	Indicates the task will not wait at all.
KN_WAIT_FOREVER	Indicates that the task is willing to wait indefinitely.
UINT_32 value	Indicates that the task will wait for the specified number of clock ticks.

```
void KN_reset_alarm(alarm);
```

Data Type	Parameter
KN_TOKEN	alarm

Description

The **reset_alarm** system call returns a previously created alarm to its creation state. In all cases, this operation is equivalent to invoking the **delete_alarm** system call, then invoking the **create_alarm** system call.

Because **reset_alarm** may be invoked on single-shot alarms even if the alarm has gone off, synchronization between an alarm reset and the expiration of the alarm time interval is not necessary.

See also: Time Management, *Installation and User's Guide*

Scheduling Category

Non-scheduling. Safe for use by interrupt handlers.

Parameters

alarm A KN_TOKEN for the alarm to be reset.

reset_handler

```
void KN_reset_handler(hdlr_area);
```

Data Type		Parameter
KN_HDLR_STRUC	*	hdlr_area

Description

The **reset_handler** system call dynamically removes an application-supplied task handler previously set by the **set_handler** system call or at initialization.

See also: Installing and Removing Task Handlers, Chapter 4

Scheduling Category

Non-scheduling. Safe for use by interrupt handlers.

Parameters

hdlr_area

A pointer to a KN_HDLR_STRUC used to set and reset the task creation, task deletion, task switch, and task change priority handlers dynamically. See the **set_handler** system call for the format of this structure.

```
void KN_resume_task(task);
```

Data Type	Parameter
KN_TOKEN	task

Description

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 called, the Kernel removes the task from the suspended state and does one of the following:

- Puts it in the ready state if it was suspended.
- Puts it in the asleep state if it was asleep-suspended.

An attempt to resume a task that is not suspended causes the disaster handler to be invoked with an exception code of E_STATE and an invoked function code of KN_RESUME_TASK_CODE.

See also: Disaster Handler, Chapter 4

Scheduling Category

Signalling. Use scheduling lock in interrupt handlers.

Parameters

task A KN_TOKEN for the task to be resumed.

send_data

```
status = KN_send_data(mailbox, data_ptr, length);
```

Data Type		Parameter
KN_STATUS		status
KN_TOKEN		mailbox
void	*	data_ptr
UINT_32		length

Description

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.

When receiving (using the **receive_data** system call) and sending (using the **send_data** system call) mailbox messages, interrupts are disabled for the time it takes to copy the message. Hence, a large data transfer via mailboxes may affect interrupt latency.

Scheduling Category

Signalling. Use scheduling lock in interrupt handlers.

Return Value

status A KN_STATUS indicating the result of the call. Values are:

Literal	Meaning
E_OK	Indicates that the mailbox accepted the message.
E_LIMIT_EXCEEDED	Indicates that the message was rejected because the mailbox was full.

Parameters

mailbox

A KN_TOKEN for the mailbox to which the message is to be sent.

data_ptr

A pointer to an area containing the message to be sent.

length

A UINT_32 indicating the number of bytes in the message to be sent. Its maximum value is the maximum message size specified when the mailbox was created.

send_dl

```
status = KN_send_dl(message_ptr);
```

Data Type		Parameter
KN_STATUS		status
KN_DATA_LINK_MSG	*	message_ptr

Description

The **send_dl** system call is used with the data link layer of the message passing protocol. It sends a data link message to the specified protocol handler on the specified host. If the message to be sent initiates a solicited transfer (a buffer request or buffer grant message), the following items are required:

- The data that will be transferred must reside in the area of physical memory required by the specific DMA device.
- The message area and any referenced data must remain stable until a transmission complete indication is received.
- The message pointer must be valid within an arbitrary addressing context, because a data link protocol handler can be called, with a message pointer, from any task that permits the handling of message interrupts. You can meet this criterion by making the message GDT-based.

If the message to be sent is unsolicited, the message area is free for reuse when the system call returns.

See also: Performance Considerations for Message Passing, Aligning Message Passing Buffers, *Installation and User's Guide*

Scheduling Category

Rescheduling. Unsafe for use by interrupt handlers.

Return Value

status A KN_STATUS indicating the results of the requested operation. Values are:

Literal	Meaning
E_OK	Indicates the message was sent or, in the case of a buffer request or grant, queued successfully for later transmission.
E_BUS_ERROR	Indicates a Parallel System Bus error.
E_BUS_TIME_OUT	Indicates a Parallel System Bus timeout.
E_RETRY_EXPIRED	Indicates the backoff-retry algorithm expired without successfully delivering the unsolicited message.
E_TRANSMISSION	Indicates a combination of retry expired, bus error, and/or bus timeout occurred.

Parameters

message_ptr

A pointer to a KN_DATA_LINK_MSG structure containing all parameters for the message transmission.

```
typedef struct {
    UINT_8                  reserved1[16];
    KN_HOST_ID              remote_host_ID;
    UINT_8                  reserved2[4];
    KN_MESSAGE_TYPE         type;
    UINT_8                  liaison_ID;
    UINT_8                  reserved3[4];
    KN_PROTOCOL_ID          protocol_ID;
    UINT_8                  reserved4;
    UINT_8                  unsol_data[26];
    KN_STATUS               data_status;
    KN_DATA_TYPE            data_type;
    UINT_8                  reserved5[3];
    UINT_32                 data_length;
    UINT_8                *  data_ptr;
    UINT_16                 data_ptr_fill;
    UINT_8                  reserved6[2];
} KN_DATA_LINK_MSG;
```

Where:

reserved[n] Various reserved fields defined by Kernel message passing which should not be written (should be preserved) by data link applications. Erroneous or unpredictable results can occur if the application modifies these fields.

remote_host_ID

A 16-bit host ID that defines the destination host (when sending a message) or source host (when receiving a message). The remote host ID must be in the range 0-254. When tasks send messages, the remote host ID serves as the PSB destination message ID. When messages are received, this field contains the PSB source message ID to identify the host that sent the message.

type A KN_MESSAGE_TYPE that specifies the type of Multibus II message being passed. The following types are permitted:

Literal	Meaning
KN_UNSOL	Unsolicited message. When receiving messages of this type, only the fields `reserved1` through `unsol_data` are defined.
KN_BROADCAST	Unsolicited broadcast message. When receiving messages of this type, only the fields `reserved1` through `unsol_data` are defined.
KN_BUFFER_REQUEST	Solicited buffer request message. When receiving messages of this type, only the fields `reserved1` through `data_length` are defined.
KN_BUFFER_GRANT	Solicited buffer grant message.
KN_BUFFER_REJECT	Solicited buffer reject message.
KN_SEND_COMPLETE	Message sent by the Kernel to the protocol handler indicating that the entire solicited message has been sent.
KN_RECEIVE_COMPLETE	Message sent by the Kernel to the protocol handler indicating that the entire solicited message has been received.

The remainder of the KN_DATA_LINK_MSG structure items listed in this section contain the actual data link message.

liaison_ID An identifier that associates buffer requests and buffer grants. It is defined only for received buffer request messages and transmitted buffer grant and buffer reject messages. When a task sends a buffer request message, the Kernel fills in the `liaison_ID` field with an ID. When the host that fields the buffer request message sends either a buffer grant or buffer reject message in reply, it must specify the same ID in the `liaison_ID` field.

protocol_ID A KN_PROTOCOL_ID that identifies the protocol handler that should receive this message. You attach a protocol handler with a given protocol ID with the **attach_protocol_handler** system call. The data link layer's interrupt handler will call that protocol handler (with scheduling stopped) whenever it receives a message with the protocol handler's ID in the `protocol_ID` field. Protocol IDs can have the following values: 0 -- a 4-byte message (`remote_host_ID` through `type=KN_UNSOL`)is being sent to a MIC host; 1-7FH -- Intel reserved; 80H-0FFH -- non-Intel applications.

unsol_data When a task sends an unsolicited message (type KN_BROADCAST or KN_UNSOL) or sends unsolicited data with a KN_BUFFER_REQUEST, it can fill in this field with any information it wishes to send. This is the space remaining for unsolicited messages after the hardware and data link fields are accounted for. However, for KN_BUFFER_REQUEST messages, only the first 22 bytes of the `unsol_data` field are available for use.

data_status A KN_STATUS field that the Kernel fills in when it sends KN_SEND_COMPLETE and KN_RECEIVE_COMPLETE messages. This status field indicates the status of the solicited message that was transformed. The possible values are as follows:

Literal	Meaning
E_OK	Message transmission successful.
E_BUS_ERROR	A Parallel System Bus error occurred.
E_BUS_TIMEOUT	A Parallel System Bus timeout occurred.
E_NO_RESOURCE	Solicited channel resources were busy (masked by the data link).

E_RETRY_EXPIRED	The backoff-retry algorithm expired without successfully delivering the message.
E_SI_CANCEL	Solicited input was cancelled due to a **cancel_dl** system call from the message originator or a buffer reject message from the receiver.
E_SI_FAIL_SAFE_EXPIRED	The solicited input failsafe counter expired, indicating data packets stopped coming in.
E_SO_CANCEL	Solicited output was cancelled due to a **cancel_dl** system call from the message originator or a buffer reject from the receiver.
E_SO_FAIL_SAFE_EXPIRED	The solicited output failsafe counter expired, indicating no buffer grant or reject was received.
E_SO_PROTOCOL	A solicited output error occurred due to a Parallel System Bus problem.
E_SO_RETRY_EXPIRED	The backoff-retry algorithm expired without successfully delivering a data packet of a solicited output transfer.
E_TRANSMISSION	A combination of retry expired, bus error, and/or bus timeout occurred.

data_type A KN_DATA_TYPE field that defines whether a segment or a data chain is being transferred as a solicited message. If the type is KN_SEGMENT, the `data_ptr` field refers to a single, contiguous processor segment. If the type is KN_CHAIN, the `data_ptr` field refers to an array of KN_CHAIN_STRUC structures. Each structure in the array indicates one data block of the chain. In the case of KN_CHAIN, the message-passing DMA controller must support data chain blocks. The format for KN_CHAIN_STRUC is as follows:

```
typedef struct {
   UINT_32            byte_count;
   UINT_8            *data_ptr;
   UINT_16            data_ptr_fill;
   UINT_8             reserved[2];
   } KN_CHAIN_STRUC;
```

Where:

byte_count The length of a data block in bytes. A maximum value of 0FFFFH is permitted. A count of 0 indicates a null block and the end-of-chain. The remainder of the fields in this structure are undefined following a byte count of 0.

data_ptr A pointer to the start of the data block. All data in the chain must be in the same descriptor table.

data_ptr_fill Do not set this field. In small model, it is a placeholder. In compact model, it holds the selector of the preceding pointer.

reserved Should not be used by the application.

data_length For buffer request messages, the size of the data being sent. For buffer grant messages, this field specifies the total length of the segment or data chain to be received. If the data is chained, then the data length is the total length of the data chain (the sum of all data chain byte counts). The data length should be a multiple of 4 (for fast mode transfers) or 16 (for burst mode transfers) for best performance in data transfers.

data_ptr Pointer to the segment or data chain that is being transferred as a solicited message. KN_BUFFER_REQUEST and KN_BUFFER_GRANT messages are used to initiate the transfer of solicited messages. When the solicited message transfer is complete, the Kernel sends KN_SEND_COMPLETE and KN_RECEIVE_COMPLETE messages to the protocol handlers of the sending and receiving hosts, respectively. The data pointer should be a multiple of 4 (for fast mode transfers) or 16 (for burst mode transfers) for best performance in data transfers.

If this is a pointer to a chain block, it must be in the same descriptor table as the elements in the chain block. The application must preserve buffers used for buffer requests and buffer grants, as well as the solicited data, from the time a KN_BUFFER_REQUEST or KN_BUFFER_GRANT message is sent until a KN_SEND_COMPLETE or KN_RECEIVE_COMPLETE message is received.

data_ptr_fill Do not set this field. In small model, it is a placeholder. In compact model, it holds the selector of the preceding pointer.

```
void KN_send_EOI(slot);
```

Data Type	Parameter
UINT_8	slot

Description

The **send_EOI** system call is used by interrupt handlers to indicate to the PIC(s) that the specified interrupt has been processed, and that another interrupt from the same interrupt line can be serviced.

Scheduling Category

Non-scheduling. Safe for use by interrupt handlers.

Parameters

slot A UINT_8 indicating the entry (slot number) in the IDT that corresponds to the interrupt line to which the EOI is to be sent.

send_priority_data

```
status = KN_send_priority_data(mailbox, data_ptr, length);
```

Data Type		Parameter
KN_STATUS		status
KN_TOKEN		mailbox
void	*	data_ptr
UINT_32		length

Description

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 queued. If the mailbox is full, an exception is returned.

Mailboxes normally store messages in a FIFO queue. **Send_priority_data** places a message at the head of the queue. A series of **send_priority_data** calls results in messages being queued in LIFO order.

When you create a mailbox, you can specify one of the slots in its queue as reserved for a high priority message. The reserved slot ensures that at least one high priority message is accepted even when the mailbox queue is full. When the high priority message arrives, the Kernel attempts to place this message ahead of all the other messages. If the message queue is full, the Kernel puts the high-priority message into the reserved slot instead. If that reserved slot is also taken, an exception (E_LIMIT_EXCEEDED) is returned. This is the same exception code that is returned when a non-priority message cannot be sent because the mailbox queue is full.

Scheduling Category

Signalling. Use scheduling lock in interrupt handlers.

Return Value

status

A KN_STATUS indicating the result of the call. Values are:

Literal	Meaning
E_OK	Indicates that the mailbox accepted the message.
E_LIMIT_EXCEEDED	Indicates that the message was rejected because the mailbox was full.

Parameters

mailbox

A KN_TOKEN for the mailbox to which the message is to be sent.

data_ptr

A pointer to an area containing the message to be sent.

length

A UINT_32 indicating the number of bytes in the message to be sent. This value can be no greater than the maximum message size specified when the mailbox was created.

send_tp

```
status = KN_send_tp(message_ptr);
```

Data Type		Parameter
KN_STATUS		status
void	*	message_ptr

Description

The **send_tp** system call is used to send a transport protocol message. The type of message you send determines the actions you must take to ensure proper receipt of the message.

If you send an unsolicited message that is not part of a transaction, you are free to reuse the message area as soon as the system call completes.

If your message initiates a solicited transfer (a buffer grant or buffer request) that is not part of a transaction, the message area and any referenced data must remain stable until you receive a transmission complete indication from the completion mailbox. If a completion mailbox is not used in a non-transaction buffer request, you must otherwise ensure the transmission is complete before reusing the message area or changing the referenced data.

If your message initiates a request/response transaction (either unsolicited or a buffer request message, with a non-zero transaction ID and a transaction control field of KN_REQUEST), the message area and any referenced data must remain stable until you receive a transmission complete indication from the response mailbox. Transaction response messages do not initiate a transaction and can be considered as either unsolicited or solicited messages for the purpose of calculating transmission completion.

If your message initiates the reception of the next fragment of a request message (the transaction control field is set to KN_SEND_NEXT_FRAGMENT), the message area and any referenced data must remain stable until you receive a transmission complete indication from the completion mailbox.

For solicited or transaction messages, the message pointer must be valid within an arbitrary addressing context, because the transport protocol handler can be called, with a message pointer, from any task that permits the handling of message interrupts. This criterion can be met by making messages GDT-based, for example.

See also: Aligning Message Passing Buffers, Message Passing, *Installation and User's Guide*
Multibus II Transport Protocol Specification

Scheduling Category

Rescheduling. Unsafe for use by interrupt handlers.

Return Value

status A KN_STATUS indicating the results of the requested operation. Values are:

Literal	Meaning
E_OK	Indicates the message was sent or, in the case of a buffer request or grant, successfully queued for later transmission.
E_RETRY_EXPIRED	Indicates the backoff-retry algorithm expired without successfully delivering the unsolicited message.
E_BUS_ERROR	Indicates a Parallel System Bus error.
E_BUS_TIME_OUT	Indicates a Parallel System Bus timeout.
E_ILLEGAL_PARAM	Indicates the message has an invalid structure.
E_RESOURCE_LIMIT	Indicates a configuration-defined resource limit has been reached.
E_TRANS_ID	Indicates the specified transaction ID is not unique or is invalid.
E_TRANSMISSION	Indicates a combination of retry expired, bus error, and/or bus timeout errors occurred.

Parameters

message_ptr

A pointer to a KN_RSVP_TRANSPORT_MSG or a KN_TRANSPORT_MSG structure containing the message to be sent.

KN_RSVP_TRANSPORT_MSG is a structure that shows the format of a request/response message. It contains two of the following KN_TRANSPORT_MSG structures. Send a KN_RSVP_TRANSPORT_MSG only when initiating a transaction.

```
typedef struct {
    KN_TRANSPORT_MSG        request_message;
    KN_TRANSPORT_MSG        response_message;
} KN_RSVP_TRANSPORT_MSG;
```

In the response_message fields, remote_host_ID through trans_control are ignored. In other words, the response message specifies response message buffers; the request message specifies the host port ID and the transaction ID of the request-response transaction.

The KN_TRANSPORT_MSG structure shows the format of the request or response portion of a transport message. This structure can also be used for the entire message if the message is not a request-response transaction. Its format is as follows:

```
typedef struct {
    UINT_8                  reserved1[16];
    KN_HOST_ID              remote_host_ID;
    UINT_8                  reserved2[4];
    KN_MESSAGE_TYPE         type;
    UINT_8                  dl_part[7];
    KN_PORT_ID              dst_port_ID;
    KN_PORT_ID              src_port_ID;
    KN_TRANS_ID             trans_ID;
    KN_TRANS_CONTROL        trans_control;
    UINT_8                  control_message[20];
    KN_STATUS               data_status;
    KN_DATA_TYPE            data_type;
    UINT_8                  reserved3[3];
    UINT_32                 data_length;
    UINT_8                * data_ptr;
    UINT_16                 data_ptr_fill;
    UINT_8                  reserved4[2];
    KN_TOKEN                completion_mbx;
    UINT_8                  transport_reserved[64];
} KN_TRANSPORT_MSG;
```

reserved[n] Several fields defined by Kernel message passing that should not be written (should be preserved) by applications. Erroneous or unpredictable results can occur if the application modifies these fields.

remote_host_ID

A 16-bit host ID that defines either the destination or source of a message. The `remote_host_ID` must be in the range 0-254. When tasks send messages, they must fill in the low-order byte of this field with the host ID of the board that is to receive the message. When the message is received, this field contains the PSB source message ID to identify the host that sent the message. When sending KN_BROADCAST messages, this field is ignored.

type

Specifies the type of Multibus II message being passed. The following types are permitted:

Literal	Meaning
KN_UNSOL	Unsolicited message. When receiving messages of this type, only the fields `reserved1` through `control_message` are defined.
KN_BROADCAST	Unsolicited broadcast message. When receiving messages of this type, only the fields `reserved1` through `control_message` are defined.
KN_BUFFER_REQUEST	Buffer request message. When receiving messages of this type, only the fields `reserved1` through `data_length` are defined.
KN_BUFFER_GRANT	Buffer grant message.
KN_BUFFER_REJECT	Buffer reject message.
KN_SEND_COMPLETE	Kernel-specified value in a received message, indicating that the entire solicited message has been sent.
KN_RECEIVE_COMPLETE	Kernel-specified value in a received message, indicating that the entire solicited message has been received.

dl_part Data link-defined message fields. Transport messages are defined so that they can overlay data link messages. The application should treat the `dl_part` field as reserved by the transport protocol.

The `dl_part` field of a KN_BUFFER_GRANT or KN_BUFFER_REJECT message must match the `dl_part` field of a received KN_BUFFER_REQUEST message. Unpredictable or erroneous results can occur otherwise.

dst_port_ID Specifies the port ID of the port that receives the message.

src_port_ID Specifies the port ID of the port that is sending the message.

trans_ID Specifies the transaction ID. This ID associates the request and response portions of transactions so that, for example, a task that receives several response messages knows which request messages correspond to those responses. The transaction ID also associates a response buffer with a particular transaction. You can establish your own conventions for assigning transaction IDs, in the range 1-255. For messages that are not request/response messages, use the value KN_NO_TRANSACTION.

trans_control Defines transport transaction control. This field indicates the portion of the transaction that this message provides. There are two basic values you can specify: KN_REQUEST and KN_RESPONSE. You can OR these values with additional values to provide more information about the request or response.

For KN_REQUEST messages, you can OR the KN_REQUEST value with one of the following:

Literal	Meaning
KN_NO_FRAGMENTATION	The client (in a client/server transaction) will not permit fragmentation. If the server replies with a KN_BUFFER_REJECT message, the transaction is terminated.
KN_FRAGMENTATION	The client permits fragmentation. If the server replies with a KN_BUFFER_REJECT message, the Kernel does not terminate the transaction, but waits for a server message with a KN_SEND_NEXT_FRAGMENT control, to initiate fragmentation.

KN_SEND_NEXT_FRAGMENT	The server requests the next fragment of the data. A message with this control must have message type KN_UNSOL and use the data fields and completion mailbox to receive the next fragment of a message.
KN_NEXT_FRAGMENT	This control is reserved for the Kernel. It uses this control when sending fragments to the server.

For KN_RESPONSE messages, you can OR the KN_RESPONSE value with one of the following:

Literal	Meaning
KN_NOT_EOT	This message contains only a fragment of the response. There are more fragments remaining.
KN_EOT	This message contains the last fragment of the response.
KN_CANCEL	This control is reserved for the Kernel. It is returned to the client when the remote port if not available (not attached to a mailbox).

control_message

The Multibus II unsolicited message application-defined bytes available to transport applications. After the hardware, data link, and transport-defined fields are accounted for, the remaining control message bytes can be used for application data in unsolicited messages. Note that if a message is of type KN_BUFFER_REQUEST, only the first 16 bytes of control_message are available for use.

data_status Defined only in KN_SEND_COMPLETE and KN_RECEIVE_COMPLETE messages. It gives asynchronous transmission status. The possible values are as follows:

Literal	Meaning
E_OK	Message transmission successful.
E_BUS_ERROR	A Parallel System Bus error occurred.
E_BUS_TIMEOUT	A Parallel System Bus timeout occurred.

E_CANCELLED	A request/response transaction has been cancelled by the Kernel because the remote port was not available.
E_FRAGMENT	In a request message, the fragmentation transmission failed (a KN_SEND_NEXT_FRAGMENT message could not be satisfied or contained a fragment length of zero).
E_RETRY_EXPIRED	The backoff-retry algorithm expired without successfully delivering the message.
E_SI_CANCEL	Solicited input was cancelled due to a **cancel_tp** system call from the client or a buffer reject from the server.
E_SI_FAIL_SAFE_EXPIRED	The solicited input failsafe counter expired, indicating data packets stopped coming in.
E_SO_CANCEL	Solicited output was cancelled due to a **cancel_tp** system call from the client or a buffer reject from the server.
E_SO_FAIL_SAFE_EXPIRED	The solicited output failsafe counter expired, indicating no buffer grant or reject was received.
E_SO_PROTOCOL	A solicited output error occurred due to a Parallel System Bus problem.
E_SO_RETRY_EXPIRED	The backoff-retry algorithm expired without successfully delivering a data packet of a solicited output transfer.
E_TRANSMISSION	A combination of retry expired, bus error, and/or bus timeout errors occurred.

data_type A KN_DATA_TYPE field that defines whether a segment or a data chain is being transferred as a solicited message. If the type is KN_SEGMENT, the `data_ptr` field refers to a single, contiguous processor segment. If the type is KN_CHAIN, the `data_ptr` field refers to an array of KN_CHAIN_STRUC structures. Each structure in the array indicates one data block of the chain. In the case of KN_CHAIN, the message-passing DMA controller must support data chain blocks.

```
typedef struct {
      UINT_32              byte_count;
       UINT_8 *            data_ptr;
      UINT_16              data_ptr_fill;
       UINT_8              reserved[2];
} KN_CHAIN_STRUC;
```

Where:

byte_count The length of a data block in bytes. A maximum value of 0FFFFH is permitted. A count of 0 indicates a null block and the end-of-chain. The remainder of the fields in this structure are undefined following a byte count of 0.

data_ptr A pointer to the start of the data block. All data in the chain must be in the same descriptor table.

data_ptr_fill Do not set this field. In small model, it is a placeholder. In compact model, it holds the selector of the preceding pointer.

reserved Should not be used by the application.

data_length For buffer request messages, the size of the data being sent. For buffer grant messages, this field specifies the total length of the segment or data chain to be received. If the data is chained, the data length is the total length of the data chain (the sum of all data chain byte counts). The data length should be a multiple of 4 (for fast mode transfers) or 16 (for burst mode transfers) for best performance in data transfers.

data_ptr Pointer to the segment or data chain that is being transferred as a solicited message. KN_BUFFER_REQUEST and KN_BUFFER_GRANT messages are used to initiate the transfer of solicited messages. When the solicited message transfer is complete, the Kernel sends KN_SEND_COMPLETE and KN_RECEIVE_COMPLETE messages to the completion mailboxes of the sending and receiving hosts, respectively. The data pointer should be a multiple of 4 (for fast mode transfers) or 16 (for burst mode transfers) for best performance in data transfers.

If this is a pointer to a chain block, it must be in the same descriptor table as the elements in the chain block. The application must preserve buffers used for buffer requests and buffer grants, as well as the solicited data, from the time a KN_BUFFER_REQUEST or KN_BUFFER_GRANT message is sent until a KN_SEND_COMPLETE or KN_RECEIVE_COMPLETE message is received.

data_ptr_fill Do not set this field. In small model, it is a placeholder. In compact model, it holds the selector of the preceding pointer.

completion_mailbox

A KN_TOKEN for a mailbox where the calling task expects a message to be sent when the solicited message transfer is complete. If the **send_tp** system call initiates a request-response message, the completion mailbox in the request portion indicates the mailbox that should be notified when the request is transmitted. (The request portion of the message is the first KN_TRANSPORT_MSG structure of a KN_RSVP_TRANSPORT_MSG structure.) The completion mailbox in the response portion indicates the mailbox that should be notified when the response has been received. When sending a response message, the `completion_mailbox` field specifies a transaction completion mailbox.

If the calling task is sending an unsolicited, non-transaction message, or does not want to receive a completion indication, it should specify KN_NULL_TOKEN for this field.

The completion mailbox will receive a message of type KN_TRANSPORT_MBX_LOCAL_MSG. If this mailbox is also used to receive messages for a port (the **attach_receive_mailbox** call), the task must be able to differentiate this type from the KN_TRANSPORT_MBX_REMOTE_MSG type. These are described below. The first field of the structures identifies the structure type.

Data message completion and transaction response messages are received by reference from mailboxes (that is, the data received from the mailbox contains a pointer to the actual message). These messages originate locally and use the structure KN_TRANSPORT_MBX_LOCAL_MSG, as follows:

```
typedef struct {
    UINT_16                mbx_message_type;
    UINT_8               * mbx_message;
    UINT_16                mbx_message_fill;
} KN_TRANSPORT_MBX_LOCAL_MSG;
```

Kernel mailboxes attached to port IDs with the **attach_receive_mailbox** system call receive messages in the following structure. This includes all fields in the KN_TRANSPORT_MSG through `data_length` (the first 68 bytes). The format of this structure is:

```
typedef struct{
    UINT_16                mbx_message_type;
    UINT_8                 mbx_message[66];
} KN_TRANSPORT_MBX_REMOTE_MSG;
```

Where:

mbx_message_type

(Applies to both structures) Indicates the origin of the message. For remote messages, the `mbx_message_type` field overlays the first two bytes of the received message (`reserved1[0]` and `reserved1[1]`). Possible values include:

Literal	Meaning
KN_REMOTE_MSG	The message originates remotely and is received by value. The entire message is contained in the `mbx_message` field.
KN_LOCAL_MSG	The message originates locally and is received by reference. The `mbx_message` field contains a pointer to the actual message.

If application-defined messages are also sent to mailboxes that are being concurrently used for transport message passing, they should include an `mbx_message_type` UINT_16 field as well. The application-defined messages should have a `mbx_message_type` value different than KN_REMOTE_MSG and KN_LOCAL_MSG, so that they can be distinguished from transport messages.

mbx_message (Applies to both structures) A mailbox message of `mbx_message_type` KN_REMOTE_MSG contains a received message that is unsolicited, broadcast, or a buffer request. Received remote messages have a length of 68 bytes (fields `reserved1` through `data_length`). Received unsolicited and broadcast messages do not contain a valid `data_length` field.

A mailbox message of `mbx_message_type` KN_LOCAL_MSG contains a pointer to a message previously supplied in a **send_tp** system call, with returned information in the appropriate fields. Received local messages have a length of 8 bytes (two bytes for the `mbx_message_type` field and six bytes for the pointer).

Applications must supply a buffer of at least KN_REMOTE_MSG_OVERHEAD bytes when receiving data from a port mailbox. Applications must supply a buffer of at least 8 bytes when receiving data from a completion mailbox.

KN_REMOTE_MSG_OVERHEAD
: The minimum amount of space that should be reserved for receiving a remote message from a port mailbox. If the task uses the same buffer to send a return message, it must instead reserve a buffer the length of KN_TRANSPORT_MSG.

mbx_message_fill
: Do not set this field. In small model, it is a placeholder. In compact model, it holds the selector of the preceding pointer.

```
void KN_send_unit(semaphore);
```

Data Type	Parameter
KN_TOKEN	semaphore

Description

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 and given the unit.

If **send_unit** is invoked on a semaphore that contains 65,535 units, the count of units in the semaphore is not incremented, and the disaster handler is invoked with an exception code of E_LIMIT_EXCEEDED and an invoked function code of KN_SEND_UNIT_CODE.

Scheduling Category

Signalling. Use scheduling lock in interrupt handlers.

Parameters

semaphore

A KN_TOKEN for the semaphore to which the unit is to be sent.

set_descriptor_attributes

```
void KN_set_descriptor_attributes(table, descriptor,
                                  attribute_ptr);
```

Data Type		Parameter
KN_SELECTOR		table
KN_SELECTOR		descriptor
void	*	attribute_ptr

Description

The **set_descriptor_attributes** system call sets the attribute values of a specified descriptor. You provide the new attributes by specifying a KN_SEGMENT_ATTRIBUTES_STRUC or KN_GATE_ATTRIBUTES_STRUC structure containing the new information.

When changing the attributes of a segment descriptor, you can modify the `base` and `size` fields of the descriptor to specify the section of memory accessible with the segment descriptor. For both expand-up and expand-down segments, you must set the `base` and `sel` fields of the KN_SEGMENT_ATTRIBUTES_STRUC structure to refer to the starting address of the segment and the `size` field to refer to the segment length.

For expand-up segments, the Kernel sets the descriptor's `base` field with the base address you specify and the `limit` field with a value derived from the `size` field as described later.

For expand-down segments, the Kernel derives the `limit` field by subtracting the `size` value you supply from the upper bound of the segment (4 gigabytes for 32-bit segments and 1Mbyte for 16-bit segments). It sets the `base` field of the descriptor by subtracting the derived `limit` field from the `base` and `sel` combination you specified. Refer to the *386 DX Programmer's Reference Manual* for a description of address translation using expand-down segments.

When you use the **set_descriptor_attributes** system call, you cannot set the granularity bit of a segment descriptor. The system call attempts to use byte granularity whenever possible. Therefore, segment descriptors whose limit values are less than 1 Mbyte are given byte granularity. Larger segments are given page granularity (4K bytes).

If you use **set_descriptor_attributes** to set up segments that are larger than 1 Mbyte, the limit value must be a multiple of 4K bytes (the page size), or results are undefined. This means that you must include a `size` field that is a multiple of 4K bytes for expand-up segments that are greater than or equal to 1 Mbyte and for all mode-32 expand-down segments.

The operation of **set_descriptor_attributes** is indivisible. That is, two **set_descriptor_attributes** system calls simultaneously invoked on the same descriptor are performed one after the other. Also, if a **get_descriptor_attributes** system call is invoked on a descriptor while a **set_descriptor_attributes** system call is invoked on the same descriptor, the Kernel returns either the old or the new attributes of the descriptor, but not a combination of the two.

Scheduling Category

Non-scheduling. Safe for use by interrupt handlers.

Parameters

table A KN_SELECTOR for the descriptor table containing the descriptor whose attributes are to be set. Possible values are:

- A KN_SELECTOR for the GDT alias (in slot 1 of the GDT)
- A KN_SELECTOR for an LDT alias (a GDT descriptor that references an LDT)
- A KN_SELECTOR for an LDT

descriptor

A KN_SELECTOR for the descriptor whose attributes are to be set.

attribute_ptr

A pointer to an area containing the values of the new attributes for the specified descriptor. To set the attributes of a segment, use the KN_SEGMENT_ATTRIBUTES_STRUC structure for this area. To set the attributes of a gate, use the KN_GATE_ATTRIBUTES_STRUC structure.

When using the small model interface, KN_SEGMENT_ATTRIBUTES_STRUC has the following format (for compact model, the `sel` field is not be present because the `base` field is a full 48-bit pointer):

set_descriptor_attributes

```
typedef struct {
   UINT_8                  access;
   UINT_8                  mode;
   UINT_8                * base;
   KN_SELECTOR             sel;
   UINT_32                 size;
} KN_SEGMENT_ATTRIBUTES_STRUC;

typedef struct {
   UINT_8                  access;
   UINT_8                  word_count;
   UINT_8                * base;
   KN_SELECTOR             sel;
} KN_GATE_ATTRIBUTES_STRUC;
```

Where:

access (applies to both structures) Indicates the type of descriptor this is. This field corresponds to the access byte of the descriptor (bits 7 through 15 of the descriptor's second doubleword). Refer to the *386 DX Programmer's Reference Manual* for more information about the access byte. There are several masks that you can apply to this field to obtain the access information. The flags are described at the end of this section.

mode A UINT_8 that indicates whether the segment is a 16-bit or a 32-bit segment. The following literals apply.

Literal	Meaning
KN_MODE_BIT	A mask for this field of the value:
KN_MODE_32	The segment is a 32-bit segment.
KN_MODE_16	The segment is a 16-bit segment.

base (applies to both structures) A pointer that specifies the beginning address of the segment. In small-model applications, this pointer is a 32-bit offset and the `sel` parameter indicates the selector for the segment from which the offset starts. In compact-model applications, this pointer is a full 48-bit pointer encompassing both the offset and the selector.

sel (applies to both structures) A KN_SELECTOR that identifies the segment from which the `base` is assumed to start. This field is only present for small-model applications.

size A UINT_32 indicating the size of the segment in bytes. The **get_descriptor_attributes** system call returns a 0 value in this field to indicate a 4G-byte segment.

word_count A UINT_8 indicating the number of words that are transferred from the calling procedure's stack to the new stack whenever a procedure makes an inter-level call using this gate.

The following flag literals can be used to get information from the `access` field.

KN_DATA_SEG:

The descriptor represents a data segment. Data segments can have the following attributes:

Writable Determines if the data segment is writable.

Literal	Meaning
KN_DATA_D_WRITABLE_BIT	A mask for this field.
KN_WRITABLE	The data segment is writable.
KN_NOT_WRITABLE	The data segment is not writable.

Expand Determines if the data segment is expand-up or expand-down.

Literal	Meaning
KN_DATA_D_EXPAND_DIR_BIT	A mask for this field.
KN_EXPAND_DOWN	The segment is expand down.
KN_EXPAND_UP	The segment is expand up.

KN_EXEC_SEG

The descriptor represents an executable code segment. Executable segments can have the following attributes:

Readable Determines if the executable segment is readable.

Literal	Meaning
KN_EXEC_READABLE_BIT	A mask for this field.
KN_READABLE	The segment is readable.
KN_NOT_READABLE	The segment is not readable.

Conforming

Attributes of conforming segments are taken from the following literals:

Literal	Meaning
KN_EXEC_D_CONFORMING_BIT	A mask for this field.
KN_CONFORMING	The segment is conforming.
KN_NOT_CONFORMING	The segment is not conforming.

KN_SYS_SEG

The descriptor represents a system segment (a gate, a TSS, or an LDT). It can have the following attributes:

Literal	Meaning
KN_286_COMPATIBLE	The segment is 286-compatible.
KN_386_SPECIFIC	The segment is 386-specific.
KN_AVAILABLE_286_TSS	The segment is an available 286 Task State Segment.
KN_LDT	The segment is an LDT.
KN_BUSY_286_TSS	The segment is a busy 286 Task State Segment.
KN_286_CALL_GATE	The segment is a 286 call gate.
KN_286_OR_386_TASK_GATE	The segment is a 286 or 386 task gate.
KN_286_INTR_GATE	The segment is a 286 interrupt gate.
KN_286_TRAP_GATE	The segment is a 286 trap gate.
KN_AVAILABLE_386_TSS	The system is an available 386 Task State Segment.
KN_BUSY_386_TSS	The segment is a busy 386 Task State Segment.
KN_386_CALL_GATE	The segment is a 386 call gate.
KN_386_INTR_GATE	The segment is a 386 interrupt gate.
KN_386_TRAP_GATE	The segment is a 386 trap gate.

In addition to the information that is specific to particular types of descriptors, the following masks apply to all descriptors:

Privilege Level

Determines the privilege level of the descriptor.

Literal	Meaning
KN_DPL_MASK	A mask for this field.
KN_DPL_0	The descriptor is privilege level 0.
KN_DPL_1	The descriptor is privilege level 1.
KN_DPL_2	The descriptor is privilege level 2.
KN_DPL_3	The descriptor is privilege level 3.

The value KN_DPL_SHIFT_COUNT can be used to shift the mask left to the proper bit position in the access byte.

Present

Determines if the data is present in memory.

Literal	Meaning
KN_PRESENT_BIT	A mask for this field.
KN_PRESENT	The data is present in memory.
KN_NOT_PRESENT	The data is not present in memory.

Accessed

Indicates whether the descriptor is currently accessed (that is, whether a selector for it is currently loaded into a segment register).

Literal	Meaning
KN_ACCESSED_BIT	A mask for this field.
KN_ACCESSED	The descriptor is currently accessed.
KN_NOT_ACCESSED	The descriptor is not currently accessed.

set_handler

```
void KN_set_handler(hdlr_area);
```

Data Type	Parameter
KN_HDLR_STRUC	* hdlr_area

Description

The **set_handler** system call dynamically installs an application-supplied task handler. Multiple task handlers for creation, deletion, task switching, and priority change may be installed by invoking **set_handler** multiple times. Install a user-supplied disaster handler with the **initialize** system call at Kernel initialization. Only one disaster handler is allowed.

See also: Installing and Removing Task Handlers, Chapter 4

Scheduling Category

Non-scheduling. Safe for use by interrupt handlers.

Parameters

hdlr_area

A pointer to a KN_HDLR_STRUC that is used to set and reset the task creation, task deletion, task switch, and task change priority handlers dynamically. Its format is:

```
typedef struct {
   UINT_32                reserved[2];
   KN_FLAGS               hdlr_flags;
   void                 * hdlr_ptr;
   KN_HDLR_TYPE           hdlr_type;
   UINT_8                 hdlr_res[3];
} KN_HDLR_STRUC;
```

Where:

reserved Should not be used by the application.

hdlr_flags A KN_FLAGS specifies whether the handler is in the same subsystem or a different subsystem from the Kernel. The following literals apply to this flag:

Literal	Meaning
KN_CALL_NEAR	The handler is in the same subsystem as the Kernel.
KN_CALL_FAR	The handler is in a different subsystem from the Kernel.

hdlr_ptr A pointer to the task handler.

hdlr_type A KN_HDLR_TYPE. Possible values are:

KN_TASK_CREATION_HANDLER
KN_TASK_DELETION_HANDLER
KN_TASK_SWITCH_HANDLER
KN_TASK_PRIORITY_CHANGE_HANDLER

hdlr_res Should not be used by the application.

NOTE

This structure must be preserved by the application until the associated handler is reset using the **reset_handler** system call. Do not reuse the handler structure.

set_interconnect

```
void KN_set_interconnect(value, slot_number,
                         register_number);
```

Data Type	Parameter
UINT_8	value
UINT_8	slot_number
UINT_16	register_number

Description

The **set_interconnect** system call sets the contents of the specified interconnect register to a specified value.

Scheduling Category

Non-scheduling. Safe for use by interrupt handlers.

Parameters

value A UINT_8 containing the value to which the specified interconnect register is to be set.

slot_number
A UINT_8 specifying the Multibus II card slot ID of the board on which the specified interconnect register is located. Values 0-20 indicate a slot on the Parallel System Bus. Values 24-29 indicate slots on the LBX II bus. A value of 31 indicates the current host. All other values are invalid.

register_number
A UINT_16 specifying the interconnect register to be set. This parameter must be in the range 0-511. Refer to the hardware manual for your Multibus II board to determine the proper register number.

```
void KN_set_interrupt(slot, handler_ptr);
```

Data Type		Parameter
UINT_8		slot
void	*	handler_ptr

Description

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 interrupt source is identified by the slot number (not the selector) of the descriptor in the IDT. Therefore an interrupt source is identified by a value in the range 0-255.

When you assign an interrupt handler to an IDT slot, any previous assignment of an interrupt handler to the IDT slot is overwritten.

Scheduling Category

Non-scheduling. Safe for use by interrupt handlers.

Parameters

slot A UINT_8 indicating the IDT slot number to which the specified handler is to be assigned. You can specify values in the range 0-255.

handler_ptr
A pointer to the first instruction of the interrupt handler. In the small model interface, this pointer is assumed to be relative to the caller's code segment.

set_priority

```
void KN_set_priority(task, priority);
```

Data Type	Parameter
KN_TOKEN	task
UINT_16	priority

Description

The **set_priority** system call changes the static priority of the specified task. If the task's dynamic priority has been adjusted due to ownership of a region and the requested priority change would lower the static priority of the specified task, the Kernel delays the priority change until the task gives up control of all regions. Even if the priority change is delayed, the system call returns immediately.

See also: Region Semaphores, *Installation and User's Guide*

Scheduling Category

Signalling. Use scheduling lock in interrupt handlers.

Parameters

task A KN_TOKEN for the task whose priority is to be changed.

priority

A UINT_16 specifying the new priority for the task.

```
void KN_set_time(time);
```

Data Type	Parameter
UINT_64	time

Description

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. When the Kernel is initialized, the count is set to zero. Applications can determine the current value of the clock by calling the **get_time** system call.

Scheduling Category

Non-scheduling. Safe for use by interrupt handlers.

Parameters

time A UINT_64 specifying the new value of the system clock.

sleep

```
void KN_sleep(time_limit);
```

Data Type	Parameter
UINT_32	time_limit

Description

The **sleep** system call puts the calling task in the asleep state for the specified number of clock ticks.

Scheduling Category

Rescheduling. Unsafe for use by interrupt handlers.

Parameters

time_limit

A UINT_32 specifying the number of clock ticks for which the task is to sleep, or one of the following literals.

Literal	Meaning
KN_DONT_WAIT	Indicates a zero length time interval.
KN_WAIT_FOREVER	Indicates an infinite time interval.

KN_DONT_WAIT does not cause the running task to go to sleep. It has an effect only if there are other ready tasks of equal priority. In that case, the running task is made ready and put in the ready queue after all other ready tasks of equal priority. If there are no other ready tasks of equal priority, the current task remains running.

The value KN_WAIT_FOREVER causes the task to sleep forever. This effectively deletes the task but the task's memory is not released.

```
void KN_start_PIT(interval);
```

Data Type	Parameter
UINT_16	interval

Description

The **start_PIT** system call starts the PIT counting using the specified interval. You must initialize the PIT (with the **initialize_PIT** system call) before invoking this system call. If the PIT is already counting when this system call is called, the PIT is reprogrammed and restarted using the new interval specified.

Scheduling Category

Non-scheduling. Safe for use by interrupt handlers.

Parameters

interval

A UINT_16 specifying the interval in milliseconds to be used by the PIT. The value must be in the range 1 to (2^{16})/(input clock frequency in kHz).

start_scheduling

```
void KN_start_scheduling();
```

Description

The **start_scheduling** system call cancels one scheduling lock imposed by **stop_scheduling**. If the lock that is cancelled is the last outstanding scheduling lock, all task state transitions that were temporarily delayed are carried out, and the highest priority ready task begins executing.

NOTE

The Kernel sometimes stops scheduling internally, so that scheduling might not restart immediately even though the application has cancelled all the scheduling locks that it established.

If **start_scheduling** is invoked when scheduling is not stopped, the results are undefined.

Scheduling Category

Signalling. Use scheduling lock in interrupt handlers.

```
void KN_stop_scheduling();
```

Description

The **stop_scheduling** system call temporarily locks the scheduling mechanism (or places an additional lock on the mechanism) for the running task. Any task state transitions that would move the task from the running state to the ready state are delayed until scheduling is resumed. For example, with scheduling stopped, if the running task sends a message to a mailbox at which a higher-priority task is waiting, that waiting task becomes ready, but it would not become the running task until scheduling is resumed.

The **stop_scheduling** system call does not necessarily halt task switching. If the running task invokes a blocking system call (such as waiting at a mailbox for a message or suspending itself) while scheduling is stopped, the task enters the asleep or suspended state immediately and the highest priority ready task becomes the running task. The new task is restored with all its scheduling locks in place. When the first task is again restored to the running state, its scheduling locks are also restored to the level they were at the time of the block.

Scheduling can be stopped repeatedly. That is, **stop_scheduling** can be invoked when scheduling is locked. Scheduling is resumed only when all scheduling locks are cancelled.

Scheduling Category

Non-scheduling. Safe for use by interrupt handlers.

suspend_task

```
void KN_suspend_task(task);
```

Data Type	Parameter
KN_TOKEN	task

Description

The **suspend_task** system call puts the specified task in the suspended state (or asleep-suspended, if the task is currently asleep.) If the task is already suspended, this system call increases its suspension depth by one. The maximum suspension depth for any task is 255. An attempt to exceed the maximum suspension depth of a task causes the disaster handler to be invoked with the exception code E_LIMIT_EXCEEDED and the invoked function code of KN_SUSPEND_TASK_CODE.

See also: Disaster Handler, Chapter 4

Scheduling Category

Signalling. Use scheduling lock in interrupt handlers.

Rescheduling when performed on the calling task. Unsafe for use by interrupt handlers when performed on the calling task.

Parameters

task A KN_TOKEN for the task to be suspended.

```
void KN_tick();
```

Description

The **tick** system call is used by applications to notify the Kernel that another clock tick has occurred. Rescheduling can occur as a result of this call. Applications that use the **initialize_PIT** and **start_PIT** system calls should not call **tick**: it is invoked automatically by the default PIT manager.

Normally, the **tick** system call is used in applications that have their own PIT handlers and omit the Kernel's PIT management module. A typical PIT handler is an interrupt handler that receives control when the timer sends an interrupt. In response, the PIT handler invokes the **tick** system call to inform the Kernel that a timer tick has occurred.

The Kernel cannot guarantee how long the **tick** system call takes to return. Because the **tick** system call can cause alarm handlers to start running (see the description of the **create_alarm** system call in this chapter), **tick** often enables interrupts during its processing. This prevents PIT handlers from having unbounded interrupt latency. (Interrupts are disabled when the timer interrupt occurs, and they aren't normally enabled until the handler executes an IRET instruction.) Because **tick** can enable interrupts, a PIT handler should send an end of interrupt (EOI) signal to the timer interrupt before calling **tick**.

It may seem that sending an EOI before invoking **tick** could cause two potential problems:

- Another interrupt could occur while the **tick** system call is active, causing **tick** to be called while **tick** is still active. This could happen even if the PIT handler doesn't enable interrupts before calling **tick**, because **tick** itself enables interrupts under many circumstances.
- Another timer interrupt could occur before the PIT handler returns. If this happens often enough, stack overflow could occur.

However, the **tick** system call has been designed to avoid both problems. First, **tick** is reentrant, so calling **tick** while **tick** is active causes no problems. In addition, **tick** maintains a counter to keep track of outstanding **tick** calls, so that it can return quickly and eliminate the stack overflow problem.

tick

Each time **tick** is called, it increments its internal counter. If **tick** increments the counter to 1, it proceeds to start the full **tick** processing, which may enable interrupts, and will return an indeterminate time later. However, if **tick** increments the counter to 2 or greater, it simply increments the counter and returns. This takes much less time than the full **tick** processing, so it happens with interrupts disabled. Because interrupts are disabled, the PIT handler is not interrupted until it returns, therefore eliminating the risk of stack overflow.

Scheduling Category

Signalling. Use scheduling lock in interrupt handlers.

```
area_ptr = KN_token_to_ptr(object);
```

Data Type		Parameter
void	*	area_ptr
KN_TOKEN		object

Description

The **token_to_ptr** system call accepts a token for an object and returns a pointer to the area that holds the contents of the object.

Scheduling Category

Non-scheduling. Safe for use by interrupt handlers.

Return Value

area_ptr
A pointer to the area containing the state of the specified object.

Parameters

object A KN_TOKEN that specifies the object.

translate_ptr

```
alias_ptr = KN_translate_ptr(original_ptr, alias_base);
```

Data Type		Parameter
void	*	alias_ptr
void	*	original_ptr
KN_SELECTOR		alias_base

Description

The **translate_ptr** system call accepts a pointer (called the original pointer) and a selector (called the alias base). It returns a pointer that points to the same memory location as the original pointer. The new pointer contains the alias base as a selector. The system call calculates the appropriate offset. It is the responsibility of the caller to verify or guarantee that the linear address spaces for the segments overlap sufficiently for the translated pointer to be valid.

This system call is useful when the application uses the Kernel's gate-based interface. If the application calls a Kernel system call that returns a pointer, but expects a pointer based on a selector different from the one set up in **initialize_subsystem**, use the **translate_ptr** system call.

Scheduling Category

Non-scheduling. Safe for use by interrupt handlers.

Return Value

alias_ptr

A pointer that is an alias for the original pointer, using the selector from the `alias_base` variable.

Parameters

original_ptr

A pointer that is assumed by the Kernel to be valid.

alias_base

A UINT_16 containing a selector for the desired segment. The selector is used to construct the alias pointer which is returned.

unmask_slot

```
void KN_unmask_slot(slot);
```

Data Type	Parameter
UINT_8	slot

Description

The **unmask_slot** system call unmasks the specified slot in the IDT. It causes the PIC(s) to unmask interrupts on the line serviced by the specified entry in the IDT.

If the specified slot is currently masked due to the effects of the **new_masks** system call, the slot remains masked until it is unmasked by the **new_masks** system call. Thus the **unmask_slot** system call overrides the effect of the **mask_slot** system call, but not the effect of the **new_masks** system call.

Scheduling Category

Non-scheduling. Safe for use by interrupt handlers.

Parameters

slot A UINT_8 indicating the entry (slot number) in the IDT corresponding to the interrupt line that is to be unmasked.

***

STANDARD INPUT AND OUTPUT FUNCTIONS 3

This chapter discusses the on-board input and output capabilities provided by the Kernel.

Kernel I/O Overview

The Kernel provides two methods of sending and receiving data using a serial device on the host running the Kernel application:

- Console I/O system calls via an optional 82530 serial communication controller (SCC) manager
- Standard I/O functions that are a subset of C library stdio

The SCC manager is a standard interface provided with the Kernel. It supports low-level access to character I/O through several Kernel system calls. The Kernel standard I/O library functions are built on these character I/O system calls. The library functions are based on standard C stdio library functions and can be used with PL/M or C language applications. An application may use either of these methods for character I/O or may combine Kernel system calls and Kernel standard I/O functions.

Figure 3-1 shows the relationship between the Kernel and the different methods of character I/O.

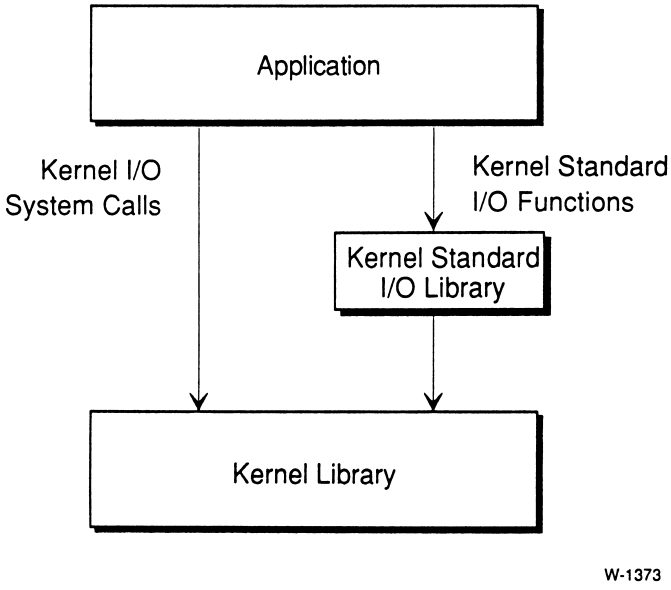


Figure 3-1. Character I/O Access Paths

I/O Initialization Required

Before the character I/O system calls can be used, console driver initialization must be performed using the **initialize_console** system call. This system call provides an initialization structure (KN_CONSOLE_CONFIGURATION_STRUC) for the 82530 serial communication controller device on the Kernel host board.

Before the Kernel standard I/O functions can be used, the application must call **initialize_console**, then initialize the standard I/O library using the **initialize_stdio** system call.

See also: Standard I/O, Kstdio, *Installation and User's Guide*

Character I/O System Calls

The Serial Communication Controller (SCC) manager, provided as a standard Kernel interface, supports low-level character I/O functions. The SCC manager source code is included and can be modified to provide a custom SCC module. The Kernel system calls supported through this interface provide direct character I/O support and also provide a basis for the Kernel standard I/O functions. These system calls are described in Chapter 2, and include:

- **KN_initialize_console**: initializes the 82530 SCC device.
- **KN_ci**: waits for character input from console.
- **KN_csts**: checks for character input from console and returns immediately.
- **KN_co**: transfers an ASCII character to the console output device but first checks for CONTROL-S or CONTROL-Q.

Kernel Standard I/O Functions

These functions are provided in the *kstdios.lib* library (small model) and *kstdioc.lib* library (compact model). The appropriate library must by bound with the application. The following functions are described more fully on the reference pages which follow.

- **putchar**: writes a single character to the standard output stream.
- **getchar**: reads a single character from the input stream and returns it as an integer value, waiting for a character if one is not immediately available.
- **printf**: formats output characters based on a format control string and sends them to the standard output stream.
- **scanf**: parses formatted character input, reading characters from the standard input stream and produces sequences of characters based on a control string format.

NOTE

These functions may block the calling task. Use them with caution in interrupt handlers.

```
void putchar(c);
```

Data Type	Parameter
UINT_8	c

Description

The **putchar** function writes a single character to the standard output stream. Values sent by **putchar** are ASCII values. For example, the following use of **putchar** places the letter "A" into the standard output stream:

```
putchar (65);
```

Assigning values to the variable and calling the **putchar** function also sends the letter "A":

```
X = 'A';
putchar (X);
```

Scheduling Category

Blocking. Use with caution in interrupt handlers.

Parameters

c Any possible 8-bit value.

getchar

```
c = getchar();
```

Data Type	Parameter
UINT_8	c

Description

The **getchar** function reads a single character from the input stream and returns it as an integer value. The **getchar** function waits for a character if one is not immediately available. It is defined to return all possible character values.

Scheduling Category

Blocking. Use with caution in interrupt handlers.

Return Value

c A UINT_8 representing the ASCII value of the character input. For example, if the input stream contains the letter "A", the returned value is 65.

```
count = printf(format_str, arg1, arg2, ... );
```

Description

The **printf** function formats character output, controlling such things as numeric format, field size, and print position. This formatting is accomplished through type conversion characters, conversion modifiers, and escape sequences.

The following are simple **printf** examples.

Example 1:

```
INPUT          printf("The number is:  %d", 5);

OUTPUT:        The number is:  5
```

Example 2:

```
INPUT          a = 5;
               printf("The number is:  %d", a);

OUTPUT:        The number is:  5
```

NOTE

The binder-generated map will have the following warning when the **printf** function is used from a PL/M application. This warning does not cause any error.

```
*** WARNING 126:  SYMBOL TYPES MISMATCH
    FILE:  kstdio.lnk
    MODULE:  KSTDIO
    SYMBOL:  PRINTF
```

Scheduling Category

Blocking. Use with caution in interrupt handlers.

printf

Return Value

count An integer that returns the actual number of characters printed. This parameter is usually ignored in C programs.

Parameters

format_str

A character string that describes how the remaining arguments (if any) are to be displayed.

The `format_str` contains a string of characters enclosed in quotation marks. Characters not preceded by a percent sign (%) or a backslash (\) are written literally to standard output. If various conversion controls are used, each control is preceded by a %. Conversion controls can affect the format, prefix, and padding of a value to be output. For example, controls can format a numeric value as an integer with a plus sign as a prefix and number of spaces to position it properly within a field.

arg1, arg2 ...

The arguments to be displayed.

The `arg` parameter(s) specify literal or variable values to be associated with formats in the `format_str` parameter. Each `%` in `format_str` corresponds sequentially with one `arg` value.

Type Conversion Modifiers

The **printf** function provides flexibility by providing character modifiers that can be specified between the percent sign and a type conversion character. The general format of a conversion specification is shown below. Optional fields are enclosed in brackets.

`%[`*flags*`][`*width*`][.`*precision*`][`*l*`]`*type*

flags Values that modify how a number will appear. They control such things as left justification and whether the number is preceded by a plus or minus sign or a space.

width	A decimal value specifying the minimum field width when a number is output. If the number is less than the minimum, the field is padded with spaces or zeros. The **width** value may also be specified by an asterisk (*) in which case the next **arg** value from the **printf** call (which must be an integer) will be used as the minimum field width.
.precision	This modifier has different effects depending upon the type conversion character that follows it. It is expressed as a period followed by an optional integer. If the integer is missing, it is assumed to be zero. The precision may also be specified using an asterisk (*) in which case the next **arg** value from the **printf** call (which must be an integer) will be used.
l	The lowercase L is used to signify that a value is a long integer.
type	The final part of the conversion specification is a type conversion character as described in Table 3-2.

The effects of individual modifiers are described in Table 3-1.

Table 3-1. printf Type Conversion Modifiers

Modifier	Meaning
flag	
-	left justify the value
+	precede the value with + or -
<space>	precede a positive value with a space character
#	The # sign has several effects depending on the type conversion character that follows it. The effects are: precede an octal value with 0 precede a hexadecimal value with 0x (or 0X) display a decimal point for floating point numbers leave the trailing zeros on for g or G format
width	minimum size of field (integer value or an *)
*	means take next argument as field width
.precision	minimum number of digits to display for integers number of decimal places for e and f formats maximum number of significant digits to display for g maximum number of characters for s format
*	means take the next argument as field width
l	display long integer
type	a type conversion character (see Table 3-2)

Conversion Characters

The conversion characters are allowed in the `format_str` parameter. Conversion characters control numeric and character formatting for specific arguments.

d Signed integer conversion from type `int` or `long`. Negative values are preceded by a minus sign. (The + flag, described later, is used to display a + or - sign before numbers.)

u Unsigned integer conversion from type `unsigned` or `unsigned long`.

o Unsigned octal conversion from type `unsigned` or `unsigned long`.

x Unsigned hexadecimal conversion from type `unsigned` or `unsigned long`. The **x** conversion operation uses the values: 0 1 2 3 4 5 6 7 8 9 a b c d e f.

X Unsigned hexadecimal conversion from type `unsigned` or `unsigned long`. The **X** conversion operation uses the values: 0 1 2 3 4 5 6 7 8 9 A B C D E F.

f Signed decimal floating-point conversion. The general output format is [-]ddd.ddd. The actual precision is controlled by the precision flag, described in the Type Conversion Modifiers section.

e Signed decimal floating-point conversion. The general format is scientific notation using a small **e** character: [-]d.ddddde+dd. The actual number of digits following the decimal point is controlled by the precision flag, described in the Type Conversion Modifiers section.

E Signed decimal floating-point conversion. The general format is scientific notation using a capital **E** character: [-]d.dddddE+dd. The actual number of digits following the decimal point is controlled by the precision flag, described in the Type Conversion Modifiers section.

g Signed decimal floating-point conversion. The format is the **f** or **e** format described above, whichever is more compact.

G Signed decimal floating-point conversion. The format is the **f** or **E** format described above, whichever is more compact.

c The argument is output as a single character.

s — The argument is output as a null-terminated character string.

% — The percent sign precedes each conversion character. To print an actual percent sign, place two percent signs together in a format field.

Table 3-2 summarizes these type conversion characters.

Table 3-2. printf Type Conversion Characters

Char	Use for Printing
d	signed integers
u	unsigned integers
o	octal integers
x	hexadecimal integers, using a-f; example: c6
X	hexadecimal integers, using A-F; example: C6
f	floating point numbers
e	floating point numbers in exponential format using e before the exponent
E	floating point numbers in exponential format using E before the exponent
g	floating point numbers in f or e format, whichever is more compact
G	floating point numbers in f or E format, whichever is more compact
c	single characters
s	null-terminated character strings
%	percent sign: precedes format characters

Escape Sequences

The final set of **printf** `format_str` value modifiers are the escape sequences, which perform formatting functions such as carriage returns or form feeds. Escape sequences are characters preceded by a backslash (\).

NOTE

The PL/M compiler does not support the "\" escape. The ASCII value corresponding to the escape sequence must be placed in the format string by the programmer.

Table 3-3 describes the effect of the various escape sequences.

Table 3-3. printf Escape Sequences

Symbol	Purpose
\a	alarm: generates an audible or visible alarm
\b	causes a backspace
\f	generates a formfeed
\n	starts a newline (CR LF)
\r	performs a carriage return without a line feed (CR)
\t	moves to the next horizontal tab
\v	moves to the next vertical tab

printf Conversion Character Examples

The following examples illustrate several of the type conversion format options.

INPUT:

```
int a = 55;
double b = 8888.88888;
printf("\n%d",a);
printf("\n%g",b);
```

OUTPUT:

```
55
8888.89
```

INPUT:

```
int a = 55;
double b = 8888.88888;
printf("\n%3d",a);
printf("\n%.3f",b);
```

OUTPUT:

```
 55
8888.889
```

scanf

```
count = scanf(format_str, arg1, arg2, ... );
```

Description

The **scanf** function parses formatted input text, reading characters from the standard input stream. The **scanf** function control `format_str` accepts a number of modifiers which allow the user to set up character input as desired. Arguments may be included after `format_str`. Each argument must be a pointer to the location where converted values from the input stream are to be stored.

If the input operation terminates prematurely because of an EOF indication or a conflict between the `format_str` controls and the input characters, **scanf** returns the number of successful assignments performed before termination occurred.

NOTE

The binder-generated map will have the following warning when the **scanf** function is used from a PL/M application. This warning does not cause any error.

```
*** WARNING 126:  SYMBOL TYPES MISMATCH
    FILE:  kstdio.lnk
    MODULE:  KSTDIO
    SYMBOL:  SCANF
```

Scheduling Category

Blocking. Use with caution in interrupt handlers.

Return Value

count An integer that returns the number of items matched. This parameter is usually ignored in C programs.

Parameters

format str

The `format_str` control is essentially a picture of the form expected of the input characters. It controls whitespace and character conversions. Each conversion specification in `format_str` begins with a % character. The `format_str` control can include both conversion characters and conversion modifiers.

arg1, arg2 ...

A list of arguments corresponding to the characters to be input. The number of arguments must match the number and type of input specifications in the `format_str` control.

Conversion Characters and Modifiers

The `format_str` parameter can include both conversion characters and conversion modifiers in the following general form:

`%[*][`*size*`][`*l*`][`*h*`]`*type*

%	The percent sign precedes each field in the format string.
[*]	The asterisk indicates a field which is to be skipped.
size	The optional maximum field width specification must be a positive (nonzero) integer.
[l]	Lowercase L indicates the value is to be stored as a `long int` or `double`.
[h]	Lowercase H indicates the value is to be stored as a `short int`.

Table 3-4 lists the meanings of the **scanf** conversion modifiers.

Table 3-4. scanf Conversion Modifiers

Modifier	Meaning
*	field to be skipped and not assigned
size	maximum size of the input field
l	value is to be stored in long integer or double
h	value is to be stored in short integer
type	conversion character

type Conversion characters declare the type of character expected or the way a character string is expected to be terminated.

The `type` conversion characters are:

d	An input that can be converted to a signed decimal is expected. A value of type `int`, `short`, or `long` is assigned, depending on the size specification.
u	An input that can be converted to a unsigned decimal is expected. A value of type `unsigned`, `unsigned short` or `unsigned long` is assigned, depending on the size specification.
o	An input that can be converted to unsigned octal is expected. A value of type `unsigned short`, or `unsigned long` is assigned, depending on the size specification.
x	An input that can be converted to unsigned hexadecimal is expected. A value of type `unsigned`, `unsigned short`, or `unsigned long` is assigned, depending on the size specification. The **x** operation will accept either abcde or ABCDE characters as input.
e, f, g	An input that can be converted to a signed floating-point number is expected. A value of type `float` or `double` is assigned, depending on the size specification. All of these operations are identical. Capital E floating-point input is also accepted.
c	Characters are expected; the number depends on the size specification.
s	A string terminated by whitespace is expected. An extra terminating null character is appended.
[...]	A string terminated by any character not within the brackets is expected.
[^...]	A string terminated by any character within the brackets is expected.
<space>	A whitespace character in the format control string causes whitespace characters in the input to be read and discarded until a non-whitespace character is read.

Table 3-5 lists the **scanf** input type conversion characters.

Table 3-5. scanf Type Conversion Characters

Character	Use for reading:
d	integers
u	unsigned integers
o	octal integers
x	hexadecimal integers, using a-f
e,f,g	floating point numbers
c	single character
s	character strings terminated by whitespace
[...]	character strings terminated by any character not listed in brackets
[^...]	character strings terminated by any character listed inside brackets
<space>	causes whitespace input characters to be discarded until a non-whitespace character is read
%	percent sign precedes each format field

In the following example, if you input 555, you will get an output of 555.

PROGRAM:

```
int a;
scanf("%d", &a);
printf("%d\n", a);
```

INPUT:

```
555
```

OUTPUT:

```
555
```

Using Kstdio Libraries

The Kernel package supplies two standard I/O libraries. These libraries provide a C language interface for the four Kernel standard I/O functions discussed earlier. Applications in languages other than C and PL/M should use appropriate compiler controls to invoke these functions.

The Kernel provides library support for both small and compact programming models using `kstdios.lib` and `kstdioc.lib` respectively. These libraries are used in several programming models including:

- Small. The entire application is in the small programming model.
- Compact. The entire application is in the compact programming model, or you are using multiple segments.

  Compact model is also used for mixed models, since mixed models must use multiple segments.

In addition to binding the appropriate libraries, include files need to be used in the applications as shown in Table 3-6. Include files are not available for FORTRAN or Assembly for the Kernel standard I/O libraries.

Table 3-6. Include Files for Stdio Models

	COMPACT	SMALL
PL/M	kstdioc.inc kstdio.lit kstdio.ext	kstdios.inc kstdio.lit kstdio.ext
C	kstdio.h	kstdio.h

Usage Notes

All Kernel standard I/O functions use C calling conventions. Use proper compiler controls so that these conventions are observed. You must include the Kernel standard I/O header files to use the Kernel standard I/O functions from a C application.

If an iC-386 library is used with the Kernel standard I/O library, `kstdioc.lib` or `kstdios.lib` must be bound before the iC-386 library.

A compiler that allows variable length parameters is required to use the Kernel standard I/O libraries. The PL/M version required is 3.3 or later.

More than one task can use a Kernel standard I/O function at the same time. The Kernel ensures that each use of the function is completed properly. However, an interrupt handler or an alarm handler runs in the context of the current task. Using a Kernel standard I/O function within one of these handlers while the task is also using the function causes deadlock.

If you use a serial line for Soft-Scope III host-target debugging communication, you cannot use the same port for I/O by the application.

***

KERNEL HANDLERS 4

This chapter discusses installation and removal of user-supplied task handlers and the Kernel system calls used to invoke these handlers. It also describes the handlers supplied by the Kernel to handle spurious interrupts. These are called level_x7 handlers.

The calling format shown in this chapter is the format used when the Kernel calls the handler.

Overview of User-supplied Task Handlers

At certain points in manipulating tasks, your application may need to add functions to those provided by the Kernel. For example, you may want to set up particular data structures when a task is created, and remove the structures when the task is deleted. The Kernel allows you to set up task handlers that it automatically calls at critical points in manipulating tasks.

Task handlers are procedures that may be installed at Kernel initialization using the **initialize** system call. Task handlers may also be installed and removed dynamically using the Kernel system calls **set_handler** and **reset_handler.**

The following task handlers can be provided by the application:

- create_task_handler
- delete_task_handler
- disaster_handler
- task_switch_handler
- priority_change_handler

NOTE

A **disaster_handler** can only be installed during Kernel initialization. Only one **disaster_handler** is allowed.

By providing these procedures, you can add functionality to your Kernel-based system and/or handle error situations.

Task Handlers Called by the Kernel

Task handlers are procedures the Kernel invokes. Task handlers may then invoke Kernel system calls and perform needed application operations. The Kernel expects these procedures to be as correct as one of its own internal calls. Incorrect task handler code can impact performance and can corrupt application operation.

Task handlers are invoked by the Kernel when the task makes a system call such as **create_task** or **delete_task**, or a system call that causes a task switch or change in priority of a task. All handlers are invoked with interrupts disabled and scheduling locked.

Figure 4-1 illustrates the interrelation between the Kernel and task handlers. The application has a task handler associated with it. When the application makes a Kernel call that would be affected by the task handler, the Kernel calls the task handler.

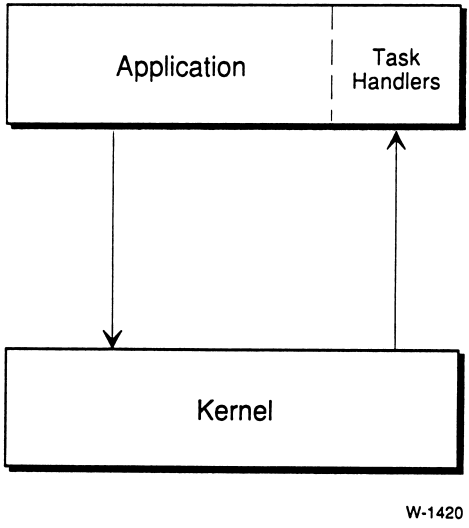


Figure 4-1. Kernel Invoking of Task Handlers

Installing and Removing Task Handlers

User-supplied task handlers may be installed at Kernel initialization using the **initialize** system call or dynamically using the **set_handler** system call. Using **set_handler** you may install more than one handler of the same kind. The **reset_handler** system call dynamically removes a specific task handler previously installed using **set_handler** or the **initialize** system call.

NOTE

Multiple task handlers degrade the performance of your system and should be removed using **reset_handler** when they are no longer needed.

The following example describes Kernel operation using task creation handlers.

1. With no task creation handlers installed, the application calls **create_task**. No handlers are invoked.
2. Using **set_handler**, the application installs two task creation handlers (`createA_hdlr` and `createB_hdlr`).
3. Then, when the application calls the **create_task** system call, the Kernel initializes the new task. Before the task is allowed to execute, the Kernel calls `createA_hdlr`, then `createB_hdlr`. Finally, it allows the task to execute.
4. Next, **reset_handler** is used to remove `createA_hdlr`. When **create_task** is next called, the Kernel initializes the new task. It calls `createB_hdlr`, then allows the task to execute.
5. Now the application reinstalls `createA_hdlr` using **set_handler**. When **create_task** is called, the Kernel initializes the new task, calls `createB_hdlr`, then `createA_hdlr`, then allows the new task to execute.
6. Finally, the application removes both task creation handlers using **reset_handler**. When **create_task** is called, the Kernel performs only its standard **create_task** functions.

See also: Task Management, *Installation and User's Guide*

create_task_handler

```
void create_task_handler(task_ptr);
```

Data Type	Parameter
void	* task_ptr

Description

The **create_task_handler** is a user-supplied procedure that the Kernel invokes whenever it creates a task. When you initialize the Kernel (via the **initialize** system call), you indicate which procedure the Kernel should use by placing a pointer to the procedure in the KN_CONFIGURATION_DATA_STRUC structure.

The **create_task_handler** can also be set up using the **set_handler** system call.

During task creation, the Kernel invokes **create_task_handler** after it initializes the new task but before the task is allowed to execute. The handler will typically perform additional task initialization, either to the Kernel maintained task state, or to any additional task state maintained by the application.

Task creation handlers are invoked with interrupts disabled and scheduling locked.

See also: **initialize**, **set_handler**, Chapter 2

Parameters

task_ptr

A pointer to the area containing the state of the new task. This area is supplied by the user with the **create_task** system call. Part of this area can be dereferenced using the structure KN_TASK_STATE. This is the only part of the task structure that is visible to the user. Some parts of this structure can be changed by the user. See the KN_TASK_STATE structure at the end of this chapter.

See also: **create_task**, Chapter 2

```
void delete_task_handler(task_ptr);
```

Data Type		Parameter
void	*	task_ptr

Description

The **delete_task_handler** is a user-supplied procedure that the Kernel invokes whenever it deletes a task. When you initialize the Kernel (via the **initialize** system call), you indicate which procedure the Kernel should use by placing a pointer to the procedure in the KN_CONFIGURATION_DATA_STRUC structure.

The **delete_task_handler** can also be set up using the **set_handler** system call.

When a task deletes another task, the Kernel invokes the task deletion handler after the task is removed from any scheduling queues (to prevent it from executing), but before the task state is destroyed. When a task deletes itself, the Kernel invokes the task deletion handler before removing the task from any scheduling queues. The deletion handler will typically perform additional task cleanup, either to the Kernel-maintained task state, or to any additional task state maintained by the application.

Task deletion handlers are invoked with interrupts disabled and with scheduling locked.

See also: **initialize**, **set_handler**, Chapter 2

Parameters

task_ptr

A pointer to the area containing the state of the task to be deleted. This area is supplied by the user with the **create_task** system call. Part of this area can be dereferenced using the structure KN_TASK_STATE. This is the only part of the task structure that is visible to the user. Some parts of this structure can be changed by the user. See the KN_TASK_STATE structure at the end of this chapter.

See also: **create_task**, Chapter 2

disaster_handler

```
void disaster_handler(info_ptr);
```

Data Type		Parameter
KN_DISASTER_INFO_STRUC	*	info_ptr

Description

The Kernel provides the **disaster_handler** interface to notify application programs when a catastrophic event has occurred. When you initialize the Kernel (via the **initialize** system call), you indicate which procedure the Kernel should use by placing a pointer to the procedure in the KN_CONFIGURATION_DATA_STRUC structure.

The **disaster_handler** can also be set up using the **set_handler** system call.

The Kernel provides a default disaster handler which causes an interrupt 3 when it is invoked.

Disaster handlers are invoked with interrupts disabled and scheduling locked.

See also: **initialize**, **set_handler**, Chapter 2

Parameters

info_ptr

A pointer to a structure containing information describing the current disaster. The format of this structure is as follows:

```
typedef struct {
    KN_STATUS               exception_code;
    UINT_32                 invoked_function;
    UINT_32                 addl_info;
} KN_DISASTER_INFO_STRUC;
```

Where:

exception_code The exception code associated with the current disaster. The following values are possible:

Literal	Meaning
E_STATE	A **resume_task** system call attempted to resume a task that wasn't suspended (that is, the task's suspension depth was already 0).
E_LIMIT_EXCEEDED	One of the following conditions occurred: A **send_unit** system call attempted to send a unit to a semaphore that already contained the maximum number of units (65,535). A **suspend_task** system call attempted to increase a task's suspension depth beyond the maximum (255).

invoked_function

Indicates the operation that was underway when the disaster occurred. The following values are possible:

Literal	Meaning
KN_SUSPEND_TASK_CODE	A **suspend_task** operation caused the disaster.
KN_RESUME_TASK_CODE	A **resume_task** operation caused the disaster.
KN_SEND_UNIT_CODE	A **send_unit** operation caused the disaster.

addl_info

A field reserved for expansion. It is intended to provide additional information regarding the disaster.

level_x7_handler

```
void KN_level_{M7|M15|Ø7|17|27|37|47|57|67|77}_handler( );
```

Description

The level_x7 handlers are different in two respects from the other handlers described in this chapter:

- They are supplied in the Kernel package as part of the optional PIC management module, rather than being written by the user.
- They are not called directly by the Kernel. The level_x7 handlers are a set of interrupt handlers, which means they are invoked by the processor hardware, rather than by the Kernel.

The level_x7 handlers are interrupt handlers that you can use to handle spurious interrupts. A spurious interrupt is one in which the interrupt signal does not remain asserted long enough for the PIC to determine its source. The 8259A PIC and the 82380/82370 Integrated System Peripheral PIC report spurious interrupts on level 7 of the PIC involved.

The Kernel supplies ten handlers for use in your application. The procedure **level_M7_handler** can be used to handle spurious interrupts on the master PIC. Procedures **level_07_handler** through **level_77_handler** can handle spurious interrupts on any of eight slave PICs. The **level_M15_handler** is used to set up the 1.5 level input when an 82380/82370 is used.

If your application does not connect level 7 PIC inputs to real interrupt sources, you can use these level_x7 handlers to handle spurious interrupts. These handlers merely signal EOI and return. You can install any or all of these procedures as interrupt handlers by using the **set_interrupt** system call or by using the Builder utility.

If an application uses level 7 PIC inputs for real interrupt sources, its interrupt handlers must distinguish spurious interrupts from real interrupts. To do this, the interrupt handler should invoke the **get_slot** system call, and compare the results of the call with the slot it believes it is servicing. If the **get_slot** system call returns the expected slot, then the interrupt is real; otherwise, the interrupt is spurious and an EOI should be sent.

See also: Bytes of stack used by interrupt handlers, Appendix B

```
void priority_change_handler(task_ptr);
```

Data Type		Parameter
void	*	task_ptr

Description

Normally the highest priority task is the running task, but when scheduling is stopped, the highest priority task might be one of the other ready tasks. Whenever the priority of one of the running or ready tasks changes, and that change causes the highest priority to change, the Kernel invokes the **priority_change_handler** if it is provided. The priority could change if a task calls the **set_priority** system call, or if it calls **send_units** or **receive_units** on a region.

When you initialize the Kernel (via the **initialize** system call), you indicate which procedure the Kernel should use by placing a pointer to the procedure in the KN_CONFIGURATION_DATA_STRUC structure.

The **priority_change_handler** can also be set up using the **set_handler** system call.

An example of using this handler would be to maintain a relationship between the priority of the running task and the set of interrupt slots that are masked.

Priority change handlers are invoked with interrupts disabled and with scheduling locked.

See also: **initialize**, **set_handler**, Chapter 2

Parameters

task_ptr

A pointer to the area containing the state of the task whose priority has been changed. This area is supplied by the user with the **create_task** system call. Part of this area can be dereferenced using the structure KN_TASK_STATE. This is the only part of the task structure that is visible to the user. Some parts of this structure can be changed by the user. See the KN_TASK_STATE structure at the end of this chapter.

See also: **create_task**, Chapter 2

task_switch_handler

```
void task_switch_handler(new_task_ptr);
```

Data Type	Parameter
void	* new_task_ptr

Description

The **task_switch_handler** is a user-supplied procedure that the Kernel invokes whenever a task switch occurs. When you initialize the Kernel (via the **initialize** system call), you indicate which procedure the Kernel should use by placing a pointer to the procedure in the KN_CONFIGURATION_DATA_STRUC structure.

The **task_switch_handler** can also be set up using the **set_handler** system call.

Whenever the Kernel switches the running task, it invokes the task switch handler. The handler is invoked in the context of the "old" task (the task giving up the processor). A pointer to the new running task is supplied as a parameter to the handler. A pointer to the state of the old running task can be obtained from the public variable KN_CURRENT_TASK.

The task switch handler typically performs changes such as changing the set of interrupt sources that are disabled. The task switch handler should not change the set of ready tasks.

Task switch handlers are invoked with interrupts disabled and with scheduling locked.

See also: **initialize**, **set_handler**, Chapter 2

Parameters

new_task_ptr

A pointer to the area containing the state of the task that will be the next running task. This area is supplied by the user with the **create_task** system call. Part of this area can be dereferenced using the structure KN_TASK_STATE. This is the only part of the task structure that is visible to the user. Some parts of this structure can be changed by the user. See the KN_TASK_STATE structure at the end of this chapter.

See also: **create_task**, Chapter 2

Exported Values

KN_CURRENT_TASK

A pointer to the task state of the current running task. For compatibility with future versions of the Kernel, KN_CURRENT_TASK should be used only in places where a parameterless procedure is valid. For example, structures should not be directly based on KN_CURRENT_TASK. Instead, KN_CURRENT_TASK should be assigned to the base of appropriate structures.

KN_TASK_STATE Structure

KN_TASK_STATE is a structure describing the state of a task. It is used in the Kernel handler procedures **create_task_handler**, **delete_task_handler**, **priority_change_handler**, and **task_switch_handler**.

The KN_TASK_STATE structure can overlay a task state to provide access to the individual fields. This structure has the format shown below. The fields from `link` through `IO_map_base` correspond to fields in the TSS. The remaining fields are specific to the Kernel.

Only the fields `ESPi`, `SSi` (where i = 0 through 2), `CR3_reg`, `LDT_reg`, `TRAP_reg`, `IO_map_base`, and `task_slice` can be written by applications. If applications attempt to write other fields, the results are undefined.

```
typedef struct {
    KN_SELECTOR         link;
    UINT_16             link_h;
    UINT_32             ESP0;
    KN_SELECTOR         SS0;
    UINT_16             SS0_h;
    UINT_32             ESP1;
    KN_SELECTOR         SS1;
    UINT_16             SS1_h;
    UINT_32             ESP2;
    KN_SELECTOR         SS2;
    UINT_16             SS2_h;
    UINT_32             CR3_reg;
    UINT_32             EIP_reg;
    UINT_32             EFLAGS_reg;
    UINT_32             EAX_reg;
    UINT_32             ECX_reg;
    UINT_32             EDX_reg;
    UINT_32             EBX_reg;
    UINT_32             ESP_reg;
    UINT_32             EBP_reg;
    UINT_32             ESI_reg;
    UINT_32             EDI_reg;
    KN_SELECTOR         ES_reg;
    UINT_16             ES_h;
    KN_SELECTOR         CS_reg;
    UINT_16             CS_h;
    KN_SELECTOR         SS_reg;
    UINT_16             SS_h;
    KN_SELECTOR         DS_reg;
    UINT_16             DS_h;
    KN_SELECTOR         FS_reg;
    UINT_16             FS_h;
    KN_SELECTOR         GS_reg;
    UINT_16             GS_h;
    KN_SELECTOR         LDT_reg;
    UINT_16             LDT_h;
    UINT_16             TRAP_reg;
    UINT_16             IO_map_base;
    KN_TOKEN            task token;
    UINT_32             task slice;
    UINT_16             dynamic priority;
    UINT_16             static_priority;
    KN_FLAGS            flags;
} KN_TASK_STATE;
```

KN_TASK_STATE Structure

Where:

link	A back link to the previous TSS.
link_h	A reserved field in the TSS.
ESP0	ESP register for privilege ring 0 operation.
SS0	SS register for privilege ring 0 operation.
SS0_h	A reserved field in the TSS.
ESP1	ESP register for privilege ring 1 operation.
SS1	SS register for privilege ring 1 operation.
SS1_h	A reserved field in the TSS.
ESP2	ESP register for privilege ring 2 operation.
SS2	SS register for privilege ring 2 operation.
SS2_h	A reserved field in the TSS.
CR3_reg	CR3 register.
EIP_reg	EIP register.
EFLAGS_reg	EFLAGS register.
EAX_reg	EAX register.
ECX_reg	ECX register.
EDX_reg	EDX register.
EBX_reg	EBX register.
ESP_reg	ESP register.
EBP_reg	EBP register.
ESI_reg	ESI register.
EDI_reg	EDI register.
ES_reg	ES register.
ES_h	A reserved field in the TSS.
CS_reg	CS register.
CS_h	A reserved field in the TSS.

SS_reg SS register. Tasks that use the message passing module should not change this descriptor.

SS_h A reserved field in the TSS.

DS_reg DS register.

DS_h A reserved field in the TSS.

FS_reg FS register.

FS_h A reserved field in the TSS.

GS_reg GS register.

GS_h A reserved field in the TSS.

LDT_reg Selector for the LDT. By default, the new task's LDT is the same as its parent task's. To give the task a different LDT, you should assign memory for the LDT, set up an LDT descriptor in the GDT, and place a selector for that descriptor in the `LDT_reg` field.

LDT_h A reserved field in the TSS.

TRAP_reg Trap bit (bit 0 of the low-order byte).

IO_map_base Offset to the start of the I/O permission map from the base of the TSS.

task_token A token for the task.

task_slice The total number of clock ticks in the task's time slice. Once changed, this value becomes effective the next time the task receives a new time slice.

dynamic_priority
The current dynamic priority of the task. This field is equal to the static priority field unless the task's priority has been adjusted because of region ownership, in which case it is equal to the adjusted priority. The dynamic priority of tasks is used in scheduling the processor.

static_priority The current static priority of the task. This field gives the priority of the task if priority adjustment due to regions is ignored.

KN_TASK_STATE Structure

flags A KN_FLAGS whose bit structure specifies the following attributes of the new task:

Idle task — Indicates whether the task is the idle task.

Literal	Meaning
KN_IDLE_TASK_MASK	Mask for this field of the flag.
KN_IDLE_TASK	The task is the idle task.
KN_NOT_IDLE_TASK	The task is not the idle task.

Initial state — The initial state of the task.

Literal	Meaning
KN_INITIAL_TASK_STATE_MASK	Mask for this field of the flag.
KN_CREATE_READY	Create the task in the ready state.
KN_CREATE_SUSPENDED	Create the task in the suspended state.

***

CONFIGURATION AND INITIALIZATION 5

This chapter discusses how to specify configuration parameters, how to make optional Kernel modules available in the application, and how to initialize the Kernel. Using optional Kernel modules is described in the *Installation and User's Guide*

Optional Modules

To include an optional module in your system, you must call one of that module's system calls in your application. When you bind your application to the Kernel, and the application references a primitive in an optional module, that module is bound into the system. If an optional module is not referenced by the application, that module is not bound with the system. Many of the optional modules must be initialized with a system call.

The optional modules include:

- Device Manager Modules. These modules consist of managers for the 8259A PIC, the 8254 PIT, the Numeric Coprocessor, the 82380/82370 Integrated System Peripheral, and the 82530 USART driver.
- Message Passing Support Module. This module supports Multibus II message passing.
- Interconnect Space Support Module. This module provides interfaces for applications to access interconnect space on Multibus II systems.
- Memory Management Module. This module manages free space.
- Descriptor Table Management Module. This module provides system calls that manage descriptor tables.
- Exchanges. This module includes mailbox management and semaphore management.

Subsystem support consists of the gate-based interface. This module consists of a library (*mux.lib*) to be bound with an application when that application exists in a different subsystem than the Kernel. There is a corresponding file to be bound with the Kernel. You must explicitly bind these library files.

Configuration Data Structures

One of the first things the application must do is to set up configuration data structures for the Kernel and for any device managers you have included. These structures are then used as input to initialization system calls. Configuration data structures for the optional modules include:

- An configuration structure for the system debugger, if used (**initialize_RDS**)
- An interrupts data structure for any PICs
- A timer data structure for any PITs
- An interconnect data structure
- A data structure for the message passing module
- The **initialize_console** primitive must be used for 82530 initialization.

There are also handler interfaces that you can supply. The four task management extension handlers include:

- create_task_handler
- delete_task_handler
- task_switch_handler
- priority_change_handler

In addition, you can supply a disaster handler. These handlers are configured into the system with the data structures that you supply to the **initialize** primitive.

Once these data structures are assembled or compiled, they are part of the object code which can then be bound to the Kernel.

Each configuration data structure must be declared. The data structures are not modified and so may reside in ROM.

Configuration Structure for initialize_RDS System Call

Figure 5-1 shows an RDS configuration data structure using the C language. The information sets fields in the KN_RDS_STRUC structure used in the **initialize_RDS** system call. Only three fields are set in the structure; the other fields are set to zero in the default configuration.

```
KN_RDS_STRUC                    rds_config;

        rds_config.MPC_port_separation          = 4;
        rds_config.IC_base_address              = 0x30;
        rds_config.IC_port_separation           = 4;
```

Figure 5-1. Example Configuration of the initialize_RDS Structure

Configuration Structure for initialize System Call

Figure 5-2 shows a Kernel configuration data structure using the C language. The values used are from the BIST example. The information to provide the configuration data structure includes:

time_slice The time interval of the time slice.

real_time_fence The priority value at which the real-time fence is set.

priority The priority of the initial task.

task_creation_handler_ptr
A pointer to a procedure to be used as the task creation handler.

task_deletion_handler_ptr
A pointer to a procedure to be used as the task deletion handler.

task_switch_handler_ptr
A pointer to a procedure to be used as the task switch handler.

priority_change_handler_ptr
A pointer to a procedure to be used as the priority change handler.

disaster_handler_ptr A pointer to a procedure to be used as the disaster handler. (If this pointer is NIL, the Kernel executes an INT 3 instruction when a disaster occurs.)

task_creation_handler_flags

task_deletion_handler_flags

task_switch_handler_flags

priority_change_handler_flags

disaster_handler_flags
Flags for each of the above procedures.

```
KN_configuration_data_struc      config_data

config_data.time_slice                          = 50;
config_data.real_time_fence                     = 256;
config_data.priority                            = 100;
config_data.task_creation_handler_ptr           = 0;
config_data.task_creation_handler_flags         = 0;
config_data.task_deletion_handler_ptr           = 0;
config_data.task_deletion_handler_flags         = 0;
config_data.task_switch_handler_ptr             = 0;
config_data.task_switch_handler_flags           = 0;
config_data.priority_change_handler_ptr         = 0;
config_data.priority_change_handler_flags       = 0;
config_data.disaster_handler_ptr                = 0;
config_data.disaster_handler_flags              = 0;
```

Figure 5-2. Kernel Configuration Data Structure for initialize System Call

Configuration Structure for initialize_PICs

To configure the interrupts data structure for the 8259A PIC(s) or the 82380/82370 Integrated System Peripheral, set up a master PIC data structure. Then set up data structures for any slave PICs that are being used. Finally, establish data structures for those PICs not being used. The fields of the PIC data structure include the following, as shown in Figure 5-3:

port_address	The first I/O port used to program the PIC.
port_separation	The distance between consecutive I/O ports used to program the PIC.
first_slot	The highest priority (numerically lowest) interrupt slot controlled by the PIC. (If a slave PIC is not present, its `first_slot` field should contain the value 0.)
sources_map	A bit map indicating which PIC inputs are connected to interrupt sources; bit 0 corresponds to input 0, etc.
type	The type of PIC.
mode	Edge or level mode.

NOTE

If you have an iSBC 386/120, or iSBC 386/133 board with a Master PIC and slave PIC, you must initialize the slave PIC even if you do not use it.

See also: **initialize_PICs**, Chapter 2

```
KN_PIC_CONFIGURATION_STRUC        picp;

        /* Master PIC */

        picp[0].port_address              = 0xC0;
        picp[0].port_separation           = 2;
        picp[0].first_slot                = 56;
        picp[0].sources_map               = 1;
/* If you have an 8259A use the following: */
        picp[0].type                      = KN_8259A_PIC;
/* If you have an 82380/82370 Integrated System Peripheral, use
        the following: */
        picp[0].type                      = KN_82380_PIC;
        picp[0].mode                      = KN_EDGE_MODE;
/* Slave PIC not used but initialized anyway.  Note
        that if you have an 82380/82370 Integrated System Peripheral,
        the slave PIC is 2.*/
        picp[8].port_address              = 0xC4;
        picp[8].port_separation           = 2;
        picp[8].first_slot                = 120;
        picp[8].sources_map               = 0;
/* If you have an 8259A use the following: */
        picp[8].type                      = KN_8259A_PIC;
/* If you have an 82380/82370 Integrated System Peripheral, use the
        following: */
        picp[8].type                      = KN_82380_PIC;
        picp[8].mode                      = KN_EDGE_MODE;
/* Those PICs which are not being used */
        picp[1].first_slot                = 0;
        picp[2].first_slot                = 0;
        picp[3].first_slot                = 0;
        picp[4].first_slot                = 0;
        picp[5].first_slot                = 0;
        picp[6].first_slot                = 0;
        picp[7].first_slot                = 0;
```

Figure 5-3. Example Configuration of an Interrupts Data Structure

Configuration Structure for initialize_PIT

The Kernel supplies two optional PIT modules, one for the 8254 and one for the 82380/82370. You may supply your own PIT manager module.

To use the optional 8254 PIT module or the 82380/82370 Integrated System Peripheral to provide the clock ticks required by the Kernel, a PIT manager module must be initialized and available for use. To configure the PIT, set up a data structure with the following information, shown in Figure 5-4:

port_address The first I/O port used in programming the PIT.

port_separation The distance between I/O ports used in programming the PIT.

in_frequency The input frequency to the PIT expressed in kHz.

slot The interrupt slot used by the PIT.

type The type of PIT.

timer_out Which timer on the PIT is to be used.

See also: initialize_PIT, Chapter 2

```
KN_PIT_CONFIGURATION_STRUC      PIT;

        PIT.port_address          = 0xD0;
        PIT.port_separation       = 2;
        PIT.in_frequency          = 1250;
        PIT.slot                  = 56;
        PIT.type                  = KN_8254_PIT;
        PIT.timer_out             = 0;
```

Figure 5-4. Example Configuration of a Timer Data Structure

Configuration Structure for initialize_NDP

The Kernel provides an optional manager for the Numeric Coprocessor. Figure 5-5 provides an example for configuring a numeric coprocessor. To configure the coprocessor, set up a data structure with the following information:

ndp_type The type of coprocessor.

flags Specifying whether or not the Numeric Coprocessor manager should initialize the coprocessor save areas in tasks.

Literal	Meaning
KN_387_HANDLER_MASK	A mask for this field of the flag.
KN_387_NO_HANDLER	Don't use the default 387 initialization handler for this task.
KN_387_DEFAULT_HANDLER	Use the default 387 initialization handler for this task.

See also: initialize_NDP, Chapter 2

```
KN_NDP_CONFIGURATION_STRUC NDP;

        NDP.ndp_type        = KN_387_NDP;
        NDP.flags           = KN_387_NO_HANDLER;
```

Figure 5-5. Example Configuration for a Numeric Coprocessor

Configuration Structure for initialize_interconnect

Configuration information is necessary for accessing interconnect space. Figure 5-6 provides an example for configuring interconnect space. The fields of the interconnect configuration data structure include:

address_port — I/O port used to specify the address for interconnect operations.

data_port — The port used to read and write data in interconnect operations.

port_separation — The separation between ports used to access interconnect space.

See also: initialize_interconnect, Chapter 2

```
KN_INTERCONNECT_STRUC      interconnect;

        interconnect.address_port        = 0x30;
        interconnect.data_port           = 0x3C;
        interconnect.port_separation     = 4;
```

Figure 5-6. Example Configuration for an Interconnect Data Structure

Configuration Structure for initialize_message_passing

The message passing module must be initialized before being used. In order to initialize it, you must provide the following configuration information:

- Information about the Message Passing Coprocessor: its interrupt slot, its I/O port address, and the device configuration values.
- Information about the DMA device: its I/O port address and channel usage.
- Internal task priorities: priorities for tasks inside the message passing module.

See also: **initialize_message_passing**, Chapter 2

In addition, the message passing module requires some memory to use as working storage for initialization. The amount of memory depends upon a number of parameters controlled by the application. The exact size is a function of the configuration data and a number of constants. This size can be obtained with the primitive **KN_mp_working_storage_size**.

For detailed explanations of the fields in the message passing configuration data structure, see the **initialize_message_passing** primitive. The values shown in Figure 5-7 are for the iSBC 386/120, 386/133, and 486/125 boards except as noted. These values are reasonable default values.

NOTE

The `DMA_duty_cycle` value, in conjunction with the length of application messages, affects the interrupt latency of the processor. If the minimum number of clock cycles between DMA burst accesses is zero, then for the duration of a solicited message transfer, the ADMA or the 82380/82370 Integrated System Peripheral can keep the CPU from gaining access to the local bus.

See also: *Microcomputer Components SAB 82258 Advanced DMA Controller for 16-bit Microcomputer Systems (ADMA)*

```
KN_MP_CONFIGURATION_STRUC                config;

/* Data Link Configuration */
config.max_protocol_id                          = 4;
config.number_data_chain_elements               = 16;
config.PSB_MPC_port                             = 0;

/*  this value is true if using the iSBC boards */
config.MPC_config                               = 0x08a;
/* this value is true in 32_bit MPC mode */
config.MPC_int_slot                             = 58;

/* this value is true if using the iSBC boards */
config.MPC_duty_cycle                           = 0;

/* this value is true if using the iSBC boards */
config.delay_scale                              = 0;

/* set si counter operation is disabled */
config.si_failsafe_timer                        = 0x80;

/* set so counter operation is disabled */
config.so_failsafe_timer                        = 0x80;

/* DMA Configuration */
config.DMA_port                                 = 0x0200;
/* following value = TRUE if 386/133 or 486/125 Burst mode DMA
support is needed */
config.auxiliary_DMA_ support                   = FALSE;
config.auxiliary_DMA_port                       = 0x0300;
/* this value is true if using the iSBC boards */
config.in_channel                               = 2;
config.out_channel                              = 3;
config.data_link_task_priority                  = 4;
config.data_link_rcv_queue_size                 = 32;
config.number_of_data_retries                   = 0;
/* Transport Configuration */
config.transport_task_priority                  = 5;
config.transport_mbx_queue_size                 = 64;
config.attached_mbx_hash_table_size             = 16;
config.transaction_hash_table_size              = 64;
config.number_attached_mbxs                     = 16;
```

Figure 5-7. Example Configuration for Message Passing

Configuration Structure for initialize_console

Configuration information is necessary for initializing the console for Kernel standard I/O. Figure 5-8 provides an example for configuring the console. The fields of the console configuration data structure include:

type	The type of the I/O device. KN_82530A_USART supports Channel A, and KN_82530B_USART supports Channel B.
data_port	Data port address for the I/O device.
control_port	Control port address for the I/O device.
clock_frequency_hi	High word of the clock frequency of the I/O device.
clock_frequency	Low word of the clock frequency of the I/O device.
baud_rate	Character transmission rate allowed by the I/O device.
interrupt_level	This parameter is not used.

```
KN_CONSOLE_CONFIGURATION_STRUC    console_config;

      console_config.type               = KN_82530A_USART;
      console_config.data_port          = 0x86;
      console_config.control_port       = 0x84;
      console_config.clock_frequency_hi = 0;
      console_config.clock_frequency    = 0x4b0000;
      console_config.baud_rate          = 9600;
      console_config.interrupt_level    = 0;
```

Figure 5-8. Example Configuration for Console Configuration Structure

Kernel Initialization

The Kernel provides an explicit initialization entry point which you must invoke before invoking any other Kernel system calls. You must then invoke the entry points of any device managers you are using after the Kernel has been initialized. The Kernel expects the processor to be in protected mode when **initialize** is called. Refer to the examples to see how initialization is done.

To initialize the Kernel you must:

1. If you will be using the debugger, initialize RDS.

   ```
   status = KN_initialize_RDS(configuration_ptr);
   ```

2. Make sure interrupts are disabled before initializing the Kernel.

3. Supply two parameters to the Kernel initialization entry point:

   a. A work area in memory (used to create a Kernel task out of the caller) of at least size KN_TASK_SIZE bytes. Add KN_387_SAVE_AREA_SIZE if the numeric coprocessor manager will be used.

   b. A configuration data structure of type KN_CONFIGURATION_DATA_STRUC.

4. Call **KN_initialize**:

   ```
   task_token =   KN_initialize(&config_data, area_ptr,
                                idle_task_area_ptr);
   ```

5. If a real interrupt source is not connected to a level 7 input, set up the spurious interrupt handler for that input. The first parameter is the IDT slot number for the **level_x7_handler**. (If an interrupt handler is not set up to handle spurious interrupts, the Kernel uses a default handler.)

   ```
   KN_set_interrupt(63, KN_level_M7_handler);
   KN_set_interrupt(127, KN_level_77_handler);
   ```

   Note: if an 82380/82370 is used, the 1.5 level input must be set up as follows:

   ```
   KN_set_interrupt(58, KN_level_M15_handler);
   ```

6. Call any device managers, optional modules, or other items that need to be initialized. The following system calls must be invoked before using their respective modules or functions:

```
KN_initialize_PICs(picp);      /* interrupt use */
KN_initialize_PIT(&PIT);       /* timer use */

/* 8253Ø init (iSBX 354 board) for console I/O */
KN_initialize_console(configuration_ptr);

initialize_stdio();            /* stdio library use */

/* interconnect space use */
KN_initialize_interconnect(&interconnect);

KN_initialize_NDP(NDP_ptr);    /* math coprocessor use */

/* message passing use */
working_storage_size =
                               KN_mp_working_storage_size(con
fig_ptr);
KN_initialize_message_passing(config_ptr,
                                working_storage_ptr);
```

7. If you have included the 8254 PIT in your system, you must start the clock with a separate call:

```
KN_start_PIT(interval);
```

8. If the application is in a subsystem different from the Kernel's, issue the system call to initialize the subsystem:

```
KN_initialize_subsystem(configuration_ptr);
```

9. Enable interrupts and start application processing.

***

EXCEPTION CODES A

Only a subset of Kernel system calls return exception codes. The Kernel does not validate parameters, so the number of calls that generate exceptions is restricted. This appendix lists the calls that return exception codes, those that indicate exceptional conditions by returning a null pointer, and those that do not return any exception code.

This appendix lists the exception codes in numerical order by hexadecimal value, and describes all exception codes in alphabetical order.

The format of the returned exception code is:

Data Type	Parameter
KN_STATUS	status

where KN_STATUS is a UINT_32 value.

Classification of System Calls

System calls can indicate a problem by returning exceptions, by returning null pointers, or by invoking a disaster handler. Many system calls do not return any status information.

Table A-1 lists the system calls that return exceptions.

Table A-1. System Calls that Return Exceptions

System Call	System Call
KN_attach_receive_mailbox	KN_receive_unit
KN_cancel_dl	KN_send_data
KN_cancel_tp	KN_send_dl
KN_initialize_RDS	KN_send_priority_data
KN_receive_data	KN_send_tp

Table A-2 lists system calls that return null pointers as an indication that the call did not succeed.

Table A-2. System Calls that Return Null Pointers

System Call	System Call
KN_create_area	KN_get_slot

Table A-3 lists system calls that invoke disaster handlers as an indication that the call did not succeed.

Table A-3. System Calls that Invoke Disaster Handlers

System Call	Exception Code	Function Code
KN_send_unit	E_LIMIT_EXCEEDED	KN_SEND_UNIT_CODE
KN_suspend_task	E_LIMIT_EXCEEDED	KN_SUSPEND_TASK_CODE

Table A-4 lists system calls that do not return an exception.

Table A-4. System Calls That Do Not Return an Exception

System Call	System Call
KN_attach_protocol_handler	KN_initialize_PICs
KN_ci	KN_initialize_PIT
KN_co	initialize_stdio
KN_create_alarm	KN_initialize_subsystem
KN_create_mailbox	KN_linear_to_ptr
KN_create_pool	KN_local_host_ID
KN_create_semaphore	KN_mask_slot
KN_create_task	KN_mp_working_storage_size
KN_csts	KN_new_masks
KN_current_task_token	KN_null_descriptor
KN_delete_alarm	KN_ptr_to_linear
KN_delete_area	KN_reset_alarm
KN_delete_mailbox	KN_reset_handler
KN_delete_pool	KN_resume_task
KN_delete_semaphore	KN_send_EOI
KN_delete_task	KN_set_descriptor_attributes
KN_get_code_selector	KN_set_handler
KN_get_data_selector	KN_set_interconnect
KN_get_descriptor_attributes	KN_set_interrupt
KN_get_interconnect	KN_set_priority
KN_get_PIT_interval	KN_set_time
KN_get_pool_attributes	KN_sleep
KN_get_priority	KN_start_PIT
KN_get_time	KN_start_scheduling
KN_initialize	KN_stop_scheduling
KN_initialize_console	KN_tick
KN_initialize_interconnect	KN_token_to_ptr
KN_initialize_LDT	KN_translate_ptr
KN_initialize_message_passing	KN_unmask_slot
KN_initialize_NDP	

Numerical List of Exception Codes

System calls that return exceptions will return the code E_OK when an operation is successful.

Table A-5 lists the hexadecimal values of the exception codes that can be returned by the Kernel.

Table A-5. Exception Codes

Hex	Exception
0000H	E_OK
0001H	E_TIME_OUT
0003H	E_LIMIT_EXCEEDED
0004H	E_NONEXIST
0005H	E_NOT_CONFIGURED
0012H	E_ILLEGAL_PARAM
001BH	E_NOT_PRESENT
0025H	E_STATE
00F0H	E_TRANSMISSION
0102H	E_CANCELLED
0103H	E_FRAGMENT
0104H	E_RESOURCE_LIMIT
0105H	E_TOO_LATE
0106H	E_TRANS_ID

Table A-6 lists the MPC errors that can be returned when using the Kernel.

Table A-6. MPC Errors Returned

Hex	Exception
0010H	E_RETRY_EXPIRED
0020H	E_NO_RESOURCE
0040H	E_BUS_ERROR
0080H	E_BUS_TIMEOUT
0010H	E_SO_CANCEL
0020H	E_SO_FAIL_SAFE_EXPIRED
0040H	E_SO_RETRY_EXPIRED
0080H	E_SO_PROTOCOL
0010H	E_SI_CANCEL
0020H	E_SI_FAIL_SAFE_EXPIRED

Descriptions of Exception Codes

The descriptions which follow provide a summary of causes for each exception code. The Kernel System Calls chapter provides more detailed descriptions of the exception codes.

For more information on processor-related exceptions, refer to the *386™ DX Programmer's Reference Manual* and the *MPC User's Manual.*

E_BUS_ERROR 0040H
Processor-related exception.

E_BUS_TIMEOUT 0080H
Processor-related exception.

E_CANCELLED 0102H
A request/response transaction has been cancelled remotely (a cancel message was received in response to a requested message).

E_FRAGMENT 0103H
In a request message, the fragmentation transmission failed (a KN_SEND_NEXT_FRAGMENT message could not be satisfied or contained a fragment length of 0).

E_ILLEGAL_PARAM 0012H
Indicates the message has an invalid structure.

E_LIMIT_EXCEEDED 0003H
Indicates that the message was rejected because the mailbox was full.

E_NO_RESOURCE 0020H
Processor-related exception.

E_NONEXIST 0004H
Returned from a **receive_unit** on a semaphore or **receive_data** on a mailbox when the semaphore or mailbox is deleted.

E_NOT_CONFIGURED 0005H
RDS did not initialize.

E_NOT_PRESENT (Processor Fault) 001BH
A reference occurred to a descriptor that is not present.

E_OK 0000H

No exceptional conditions occurred. System calls that return exceptions return the code E_OK when an operation is successful.

E_RESOURCE_LIMIT 0104H

Indicates that an internal resource limit has been reached.

E_RETRY_EXPIRED 0010H

MPC-related exception.

E_SI_CANCEL 0010H

MPC-related exception.

E_SI_FAIL_SAFE_EXPIRED 0020H

MPC-related exception.

E_SO_CANCEL 0010H

MPC-related exception.

E_SO_FAIL_SAFE_EXPIRED 0020H

MPC-related exception.

E_SO_PROTOCOL 0080H

MPC-related exception.

E_SO_RETRY_EXPIRED 0040H

MPC-related exception.

E_STATE 0025H

If you attempt to resume a task that is not suspended, a disaster will occur and an exception code will not be returned.

E_TIME_OUT 0001H

The time specified by the time_limit parameter expired before the specified action, such as receive, was completed.

E_TOO_LATE 0105H

The request to cancel the message came too late to cancel the message.

E_TRANS_ID 0106H

Non-unique or invalid transaction ID in the message.

E_TRANSMISSION 00F0H

A Multibus II message transmission error occurred in executing this system call. Whatever was being transmitted is lost; a retry may be successful.

***

STACK REQUIREMENTS B

This appendix discusses the stack requirements of application tasks. It also discusses the stack requirements for the Kernel's internal tasks, and how to set up the internal stacks.

Stack Requirements of Application Tasks

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 B-1 and B-2 on the following pages list stack requirements of Kernel-provided interrupt handlers and Kernel system calls.

Table B-1. Bytes of Stack Used by Interrupt Handlers

Bytes	Interrupt Handler	Bytes	Interrupt Handler
124	PIT handler	68	level_37_handler
60	PIT handler, invoked while active	68	level_47_handler
228	Message passing handler	68	level_57_handler
12	NDP handler	68	level_67_handler
68	level_07_handler	68	level_77_handler
68	level_17_handler	68	level_M7_handler
68	level_27_handler	68	level_M15_handler

Table B-2. 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
40	initialize_LDT
24	initialize_NDP
16	initialize_PICs
28	initialize_PIT
68	initialize_interconnect
244	initialize_message_passing

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 Stack Protection fault.

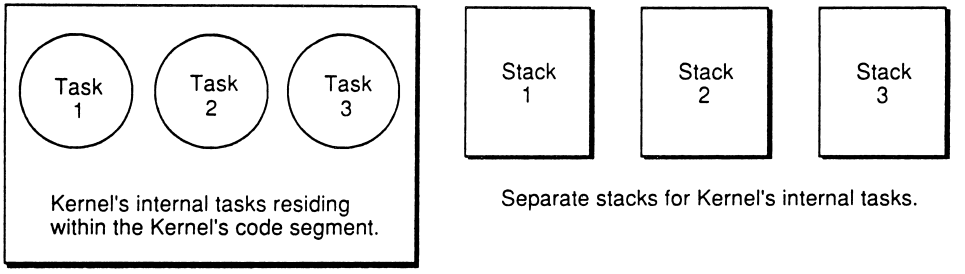
The Kernel's Internal Tasks' Stacks

The Kernel has three internal tasks, the system idle task and two message passing tasks. These reside in the Kernel's code segment and their stacks normally reside in the Kernel's data segment. The default stack size for each task is 4K bytes.

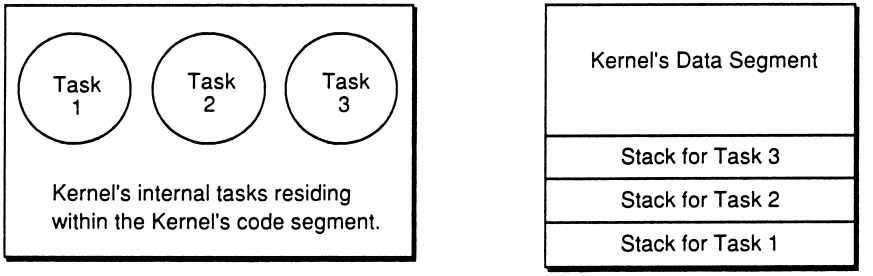
The Kernel can be used in different models of segmentation that allow different stack arrangements. Depending upon the model chosen, the user may need to, or want to, alter the default stack arrangements for these internal tasks. There are two basic arrangements possible:

- The stacks for each internal task may reside in separate segments. This is the method used in release I.2 of the Kernel. This method may be used to create expand-up stacks or expand-down stacks.
- All of the stacks may reside in the Kernel's data segment. This method allows the use of the FLAT programming model as well as other models. This is the default method used in the current version of the Kernel.

Regardless of the type of stacks chosen, the user may alter the stack size of these internal tasks. The following sections describe how to implement these stack choices and how to modify stack size. Figure B-1 illustrates the two basic task stack possibilities.



Method A: Provide three separate stacks for the Kernel's three internal tasks.



Method B: Place all of the stacks in the Kernel's data segment.

W-2603

Figure B-1. Two Methods for Creating Kernel Task Stacks

Creating and Modifying Kernel Tasks' Stacks

The following sections describe how to create the various forms of Kernel tasks' stacks and how to modify the size of the stacks. The final section provides sample code for creating task stacks in segments separate from the Kernel's data segment.

Default Stacks Within the Kernel's Data Segment

To obtain task stacks that exist within the Kernel's data segment, use the default stacks. If you use the defaults, nothing must be changed. When using default stacks, the stack size can still be altered. Changing the stack size is discussed later in this appendix.

Creating Separate Segment Stacks Using the Builder

To use the builder to create the Kernel's tasks' stacks in separate segments from the Kernel's data segment, perform the following:

1. Update the build file to include definitions for stack segments for those tasks.
2. Modify the *mp_stk.a38*, and the *idle_stk.a38* files so that the stacks are imported from the builder and the appropriate pointer variables (listed in Table B-3) are set.
3. Set the stack pointers according to the type of stack desired.
   - For expand-up stacks: stack pointer = selector of segment:(size of segment)
   - For expand-down stacks: stack pointer = selector of segment:0
4. Modify the stack size, if desired, as described later.

Table B-3. Kernel Stack Pointers

Variable Name	File
KNI_TP_TASK_STACK_PTR	mp_stk.a38
KNI_DL_TASK_STACK_PTR	mp_stk.a38
KNI_IDLE_TASK_STACK_SEG	idle_stk.a38

Modifying Stack Size

In addition to modifying the location of the Kernel tasks' stacks, the user may want to change the default 4K byte size of these stacks. Change the stack size by altering the literals in Table B-4. Their values and file locations are shown in the table.

To change the stack sizes:

1. Change the stack size literals.
2. Recompile the modules.
3. Use LIB386 to replace the existing modules in *kernel.lib* or place the new object modules ahead of *kernel.lib* in the bind sequence.

This can be done for all programming models.

Table B-4. Kernel Stack Literals

Literal Name	File	DEFAULT VALUE (UINT_32)
TP_TASK_STACK_SIZE	mp_stk.a38	1024
DL_TASK_STACK_SIZE	mp_stk.a38	1024
IDLE_TASK_STACK_SIZE	idle_stk.a38	1024

Example of Code for Separate Kernel Task Stack Segments

This section provides an example and an explanation of a build file which creates three separate stack segments for the Kernel tasks.

The default method, using the Kernel's data segments for the Kernel tasks' stacks, is discussed and illustrated in the *Installation and User's Guide* in the section on building the application. Figure B-2 contains a build file which demonstrates how to create separate stack segments for the Kernel's tasks.

In the *mp_stk.a38* file, import KNI_DL_TASK_STACK_PTR and KNI_TP_TASK_STACK_PTR segments from the builder.

In the *idle_stk.a38* file, import KNI_IDLE_TASK_STACK_SEG.

The build file listed in Figure B-2 contains five kinds of definitions, labeled CREATESEG, SEGMENT, TASK, TABLE, and MEMORY. Three of these pertain specifically to creating separate segments for stacks (CREATESEG, SEGMENT, and TABLE). The definitions for TASK and MEMORY are found in the *Installation and User's Guide* in the section on building the application.

```
test;
CREATESEG
       dl_stack_seg (SYMBOL = KNI_dl_task stack_seg),
       tp_stack_seg (SYMBOL = KNI_tp_task_stack_seg),
       idle_stack_seg (SYMBOL =
                       KNI_idle_task_stack_seg);
SEGMENT
       test            (DPL = Ø),
       test.data       (BASE = 10000H),
       dl_stack_seg    (DPL =    Ø),
       tp_stack_seg    (DPL =    Ø),
       idle_stack_seg (DPL = Ø);
TASK
initial_kernel_task
       (DPL = Ø, OBJECT = test, IOPRIVILEGE = Ø,
       NOT INTENABLED, INITIAL);
TABLE
       GDT
              (DPL = Ø,
              RESERVE = (3..63, 1024..1025),
              BASE = 20000H,
              ENTRY = (100:(test
                     dl_stack_seg,
                     tp_stack_seg,
                     idle_stack_seg))
                     ),
       IDT
              (DPL = Ø,
              RESERVE = (126..127)
              BASE = 30000H);
MEMORY
              (RESERVE = (Ø..ØFFFFH,
               0100000H..ØFFFFFFFFH));
END
```

Figure B-2. Kernel Build File Listing Separate Segments

Createseg

The CREATESEG definition instructs the Builder to create three new segments. The first two (KNI_DL_TASK_STACK_SEG and KNI_TP_TASK_STACK_SEG) are stack segments required by the message passing module if message passing and builder created stack segments are desired. If the application uses message passing system calls, include these entries in the build file exactly as listed here. If you don't use the message passing module, omit these entries.

The third segment created in Figure B-2 is the stack segment for the Kernel's idle task. When building Kernel applications, you must create this segment, specifying the public symbol name KNI_IDLE_TASK_STACK_SEG for the SYMBOL field. To create other stack segments for tasks other than the idle task, include their definitions here also. (small RAM applications cannot have separate stack segments.)

Segment

The SEGMENT definition specifies information about the individual fields in the segment descriptors. The definition in Figure B-2 provides information for all the segments in the application (test), more specific information about the application's data segment (*test.data*), information about the stack segments used by the message passing module (`dl_stack_seg` and `tp_stack_seg`), and information about the stack segment of the Kernel's idle task (`idle_stack_seg`). The definition establishes a privilege level for all the segments (DPL=0), and it supplies information about the location of the data segment. The name in the SEGMENT definition (test) is the name assigned using the BND386 NAME control.

Table

The TABLE definition establishes descriptor tables. The field:

```
100:test
```

specifies that all the descriptors in module test should be placed into the GDT, beginning at entry 100. Without this field, BLD386 creates an LDT and places the descriptors there. The entry number (100) is the recommended starting entry.

The remaining fields in the GDT definition specify other descriptors in the GDT. Because there are no numbers specified with these fields, the Builder places these descriptors immediately after the descriptors for module test. The first two (`dl_stack_seg` and `tp_stack_seg`) are required by the message passing module. Applications that use message passing facilities (and want the builder to build the stacks) must include these descriptors. Other applications should omit them. The last field (`idle_stack_seg`) specifies a descriptor for the idle task's stack segment. This entry is required for Builder-created stack segments.

***

ASSEMBLY LANGUAGE INTERFACES TO THE KERNEL C

This appendix provides information on making Kernel system calls using the assembly language interface.

Making Calls to the Kernel

Table C-1, 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 Chapter 2. 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 C-1 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 C-1 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 C-1. 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 C-1. 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 C-2.

Table C-2. 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

***

APPLICATION NOTES D

This appendix provides information on a number of topics that apply to specific application needs.

Using Address Translation Mechanisms

The Kernel puts these conditions on the use of address translation mechanisms by applications:

- Applications should not alter the mapping of the Kernel's data and code segments to physical memory.
- Mapping of any pointer parameters by an application must stay constant while the pointer is in use by the Kernel. In addition, the mapping of any pointer retained by the Kernel after the system call returns should stay constant.
- After the Kernel is given memory for constructing an object, the linear to physical mapping (that is, page tables) should not be altered while the object exists.
- The GDT must contain a segment descriptor referencing the IDT and the GDT itself. The Builder automatically creates such descriptors in slot numbers 1 and 2. See the following section.

Alias Selectors for GDT and IDT Slots

The Kernel needs to know the alias selectors for referencing the global descriptor table (GDT) and the interrupt descriptor table (IDT). For systems built with Intel tools these values are 8H and 10H respectively. These selectors are for GDT slots 1 and 2. Slot 1 contains a descriptor for the GDT itself, and slot 2 contains a descriptor for the IDT. These are considered default values, and the user does not need to configure the aliases for the GDT and IDT if GDT slots 1 and 2 contain these descriptors.

However, the selector values do not have to be 8H and 10H. When these aliases are not the default values, the user must edit the *dt_alias.a38* file, assemble it, and replace the old version in the *kernel.lib* file (or place *dt_alias.obj* before *kernel.lib* in the bind list).

Within the *dt_alias.a38* file are two public variables that determine these two alias values. They are:

```
KNI_GDT_alias_sel          (default value:   8H)
KNI_IDT_alias_sel          (default value:  10H)
TSS_alias                  (default value:  88H)
LDT_alias                  (default value:  30H)
```

NOTE

The default values for `TSS_alias` and `LDT_alias` are only required for outside product support.

I/O Permission Bit Maps

The following discussion applies only if the privilege level for the given task is less than the I/O privilege level (IOPL). Otherwise the I/O permission bit map is not used.

Intel386™ family processors selectively trap references to specific I/O addresses through the I/O permission bit map located in the TSS segment. Its size and location in the TSS segment are variable.

If the code being executed contains an I/O instruction, the code's current privilege level is first checked against its IOPL. If the IOPL for a task is lower than required by a particular I/O instruction, the processor checks the I/O permission map. Each bit in the map corresponds to an I/O port address. The processor tests all the bits that correspond to the I/O addresses spanned by an I/O operation. If any tested bit is set, the processor signals a general protection fault.

Because the I/O permission map is in the TSS segment, different tasks can have different maps. Thus, the application can allocate I/O ports to a task by changing the I/O permission map in the task's TSS.

The processor locates the I/O permission bit map by means of the `I/O_map_base` field in the fixed portion of the TSS. This field is 16 bits wide and contains the offset of the beginning of the I/O permission bit map. The value of this field can be set using a task handler. The literal KN_TSS_IO_BIT_MAP_OFFSET indicates the offset of this field relative to the TSS base. (KN_TSS_IO_BIT_MAP_OFFSET is found in *rmk_base.lit*, *rmk_base.equ*, *rmk_base.l*, and *rmk_base.par* files.)

The upper limit of the map is the same as the limit of the TSS segment. The same literal files contain another literal, KN_TASK_TSS_LIMIT, which is the offset into the task area where the given task's TSS limit field exists. The **create_task** call sets this field to 67H (that is, no I/O permission bit map).

If I/O permission bit maps are desired, perform the following:

- Set the UINT_16 value pointed to by the offset KN_TSS_IO_BIT_MAP_OFFSET to the beginning of the area where the bit map will exist:

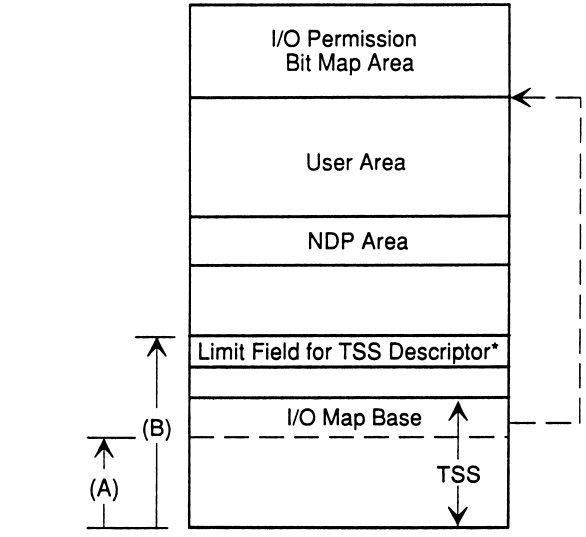
  (start addr of I/O map) minus (start addr of task area)

- Modify the UINT_16 value at the location pointed to by the offset KN_TASK_TSS_LIMIT to indicate the upper limit of the TSS, including the I/O permission bit map:

  (size of the task area in bytes) - 1

- Modify the bit map area to set the I/O privileges desired for the given task.

Figure D-1 shows the task structure, the I/O permission bit map portion of the TSS, and the offsets which allow access to the values which must be modified to show the start and limit of the I/O bit map.



Offset Values:
(A) KN_TSS_IO_BIT_MAP_OFFSET
(B) KN_TASK_TSS_LIMIT

* If I/O map is used, this field must be modified.

W-1423

Figure D-1. Task Structure

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

82380/82370 Notes

This section provides an overview to using the 82380 or 82370 Integrated System Peripheral devices and references other portions of the manual for specific discussions. The 82380 and the 82370 devices are software compatible and are referred to as the 82380/82370.

82380/82370 Functions

The 82380/82370 device provides three functions: PIT, PIC, and DMA.

An application may use either the 8254 device or the 82380/82370 but not both.

The PIT portion of the 82380/82370 device provides four timers. (There are three timers in the 8254 device.) The source code for the PIT manager is provided as part of the 82380/82370 manager.

See also: **initialize_PIT**, Chapter 2

The PIC portion of the 82380/82370 device provides three banks of controllers. The Kernel supports only banks A (master) and B (slave). The source code for the PIC manager is provided as part of the 82380/82370 manager.

See also: **initialize_PICs**, Chapter 2

The DMA portion of the 82380/82370 device provides eight channels of DMA. It works with the MPC device for message passing. The interface for the DMA function is not public. The source code provided with the 82380/82370 manager is strictly for message passing.

See also: **initialize_message_passing**, Chapter 2

Development information for the 82380/82370 is found in the *Installation and User's Guide* under the topic Binding with the 82380/82370.

82380/82370 PIC Slot Numbering

The Kernel supports banks A (master) and B (slave) of the 82380/82370 PIC functions. The numbering of these slots is consecutive beginning with the number the developer chooses for the initial slot of each bank. However, not all of the slots are accessible to the user. Figure D-2 provides an example of slot numbering and access.

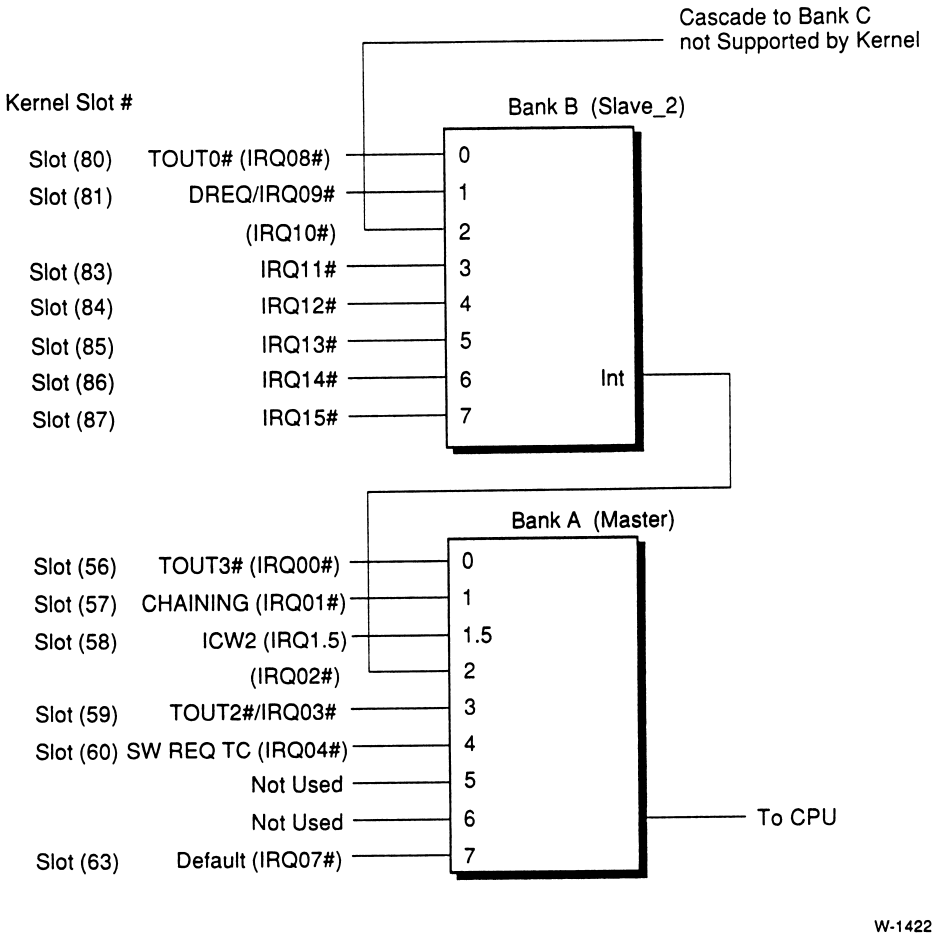


Figure D-2. Slot Ordering on the 82380

***

INDEX

D

I (continued)

K

K (continued)

L

M

M (continued)

M (continued)

N

P

R

S

S (continued)

T (continued)

T (continued)

U

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?

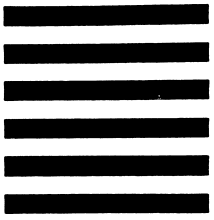
4. Did you have any difficulty understanding descriptions or wording? Where?

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