TEXAS INSTRUMENTS

# TMS320C80 (MVP) Multitasking Executive

## User's Guide

1995 Digital Signal Processing Products

Texas Instruments

Printed in U.S.A., March 1995

SPRU112A

User's Guide

# TMS320C80 (MVP) Multitasking Executive

1995

# *TMS320C80 (MVP) Multitasking Executive User's Guide*

SPRU112A
March 1995



Printed on Recycled Paper

## IMPORTANT NOTICE

Texas Instruments (TI) reserves the right to make changes to its products or to discontinue any semiconductor product or service without notice, and advises its customers to obtain the latest version of relevant information to verify, before placing orders, that the information being relied on is current.

TI warrants performance of its semiconductor products and related software to the specifications applicable at the time of sale in accordance with TI's standard warranty. Testing and other quality control techniques are utilized to the extent TI deems necessary to support this warranty. Specific testing of all parameters of each device is not necessarily performed, except those mandated by government requirements.

Certain applications using semiconductor products may involve potential risks of death, personal injury, or severe property or environmental damage ("Critical Applications").

TI SEMICONDUCTOR PRODUCTS ARE NOT DESIGNED, INTENDED, AUTHORIZED, OR WARRANTED TO BE SUITABLE FOR USE IN LIFE-SUPPORT APPLICATIONS, DEVICES OR SYSTEMS OR OTHER CRITICAL APPLICATIONS.

Inclusion of TI products in such applications is understood to be fully at the risk of the customer. Use of TI products in such applications requires the written approval of an appropriate TI officer. Questions concerning potential risk applications should be directed to TI through a local SC sales office.

In order to minimize risks associated with the customer's applications, adequate design and operating safeguards should be provided by the customer to minimize inherent or procedural hazards.

TI assumes no liability for applications assistance, customer product design, software performance, or infringement of patents or services described herein. Nor does TI warrant or represent that any license, either express or implied, is granted under any patent right, copyright, mask work right, or other intellectual property right of TI covering or relating to any combination, machine, or process in which such semiconductor products or services might be or are used.

# Read This First

## *About This Manual*

The TMS320C80 MVP (multimedia video processor) is Texas Instruments first single-chip multiprocessor DSP (digital signal processor) device. The MVP contains five powerful, fully programmable processors: a master processor (MP) and four parallel processors (PPs). The MP is a 32-bit RISC (reduced instruction set computer) with an integral, high-performance IEEE-754 floating-point unit. Each PP is an advanced 32-bit DSP; thus, in addition to having similar processing capabilities as conventional DSPs, each PP has advanced features to accelerate operation on a variety of data types.

The MVP supports a variety of parallel-processing configurations, which facilitates a wide range of multimedia and other applications that require high processing speeds. Applications include image processing, two- and three-dimensional and virtual reality graphics, audio/video digital compression, and telecommunications.

This manual describes the MVP multitasking executive software. The multitasking executive software runs on the MP and provides local control of on-chip parallel-processing tasks to present a uniprocessor-like interface to the world. The kernel has two primary components: a kernel and a software interface. The kernel consists of a software library of user-callable functions that provide basic program control, and intertask communications and synchronization. Through the software interface, tasks on the MP issue commands to the PPs. This manual provides information about the multitasking executive software features, operation, and interprocessor communications; it also includes a list of task error codes.

## *Information About Cautions*

> **This is an example of a caution statement.**
>
> **A caution statement describes a situation that could potentially damage your software or equipment.**

Please read each caution statement carefully.

## *Related Documentation From Texas Instruments*

The following books describe the TMS320C80 MVP and related support tools. To obtain a copy of any of these TI documents, call the Texas Instruments Literature Response Center at (800) 477–8924. When ordering, please identify the book by its title and literature number.

***TMS320C80 Multimedia Video Processor Data Sheet*** (literature number SPRS023) describes the features of the ’C80 device and provides pinouts, electrical specifications, and timings for the device.

***TMS320C80 Multimedia Video Processor (MVP) Technical Brief*** (literature number SPRU106) provides an overview of the ’C80 features, development environment, architecture, and memory organization.

***TMS320C80 (MVP) C Source Debugger User’s Guide*** (literature number SPRU107) describes the ’C80 master processor and parallel processor C source debuggers. This manual provides information about the features and operation of the debuggers and the parallel debug manager; it also includes basic information about C expressions and a description of progress and error messages.

***TMS320C80 (MVP) Code Generation Tools User’s Guide*** (literature number SPRU108) describes the ’C80 code generation tools. This manual provides information about the features and operation of the linker and the master processor (MP) and parallel processor (PP) C compilers and assemblers. It also includes a description of the common object file format (COFF) and shows you how to link MP and PP code.

***TMS320C80 (MVP) Master Processor User's Guide*** (literature number SPRU109) describes the 'C80 master processor (MP). This manual provides information about the MP features, architecture, operation, and assembly language instruction set; it also includes sample applications that illustrate various MP operations.

***TMS320C80 (MVP) Parallel Processor User's Guide*** (literature number SPRU110) describes the 'C80 parallel processor (PP). This manual provides information about the PP features, architecture, operation, and assembly language instruction set; it also includes software applications and optimizations.

***TMS320C80 (MVP) System-Level Synopsis*** (literature number SPRU113) contains the 'C80 system-level synopsis, which describes the 'C80 features, development environment, architecture, memory organization, and communication network (the crossbar).

***TMS320C80 (MVP) Transfer Controller User's Guide*** (literature number SPRU105) describes the 'C80 transfer controller (TC).This manual provides information about the TC features, functional blocks, and operation; it also includes examples of block write operations for big- and little-endian modes.

***TMS320C80 (MVP) Video Controller User's Guide*** (literature number SPRU111) describes the 'C80 video controller (VC). This manual provides information about the VC features, architecture, and operation; it also includes procedures and examples for programming the serial register transfer (SRT) controller and the frame timer registers.

## *If You Need Assistance. . .*

| If you want to. . . | Do this. . . |
|---|---|
| Request more information about Texas Instruments Digital Signal Processing (DSP) products | Write to:<br>Texas Instruments Incorporated<br>Market Communications Manager, MS 736<br>P.O. Box 1443<br>Houston, Texas 77251–1443 |
| Order Texas Instruments documentation | Call the TI Literature Response Center:<br>**(800) 477–8924** |
| Ask questions about product operation or report suspected problems | Call the DSP hotline:<br>**(713) 274–2320**<br>**FAX: (713) 274–2324** |
| Report mistakes in this document or any other TI documentation | Fill out and return the reader response card at the end of this book, or send your comments to:<br>Texas Instruments Incorporated<br>Technical Publications Manager, MS 702<br>P.O. Box 1443<br>Houston, Texas 77251–1443<br><br>Electronic mail: **comments@books.sc.ti.com** |

# Contents

# Figures

# Tables



# Examples

Chapter 1

# Introduction to the Multitasking Executive

The multitasking executive is a software system that executes on a TMS320C8x MVP (multimedia video processor) single-chip multiprocessor device. The purpose of the executive is to give application programs convenient and efficient access to the processing capabilities of the ’C8x.

This chapter presents the general features and capabilities of the executive and describes the hardware and software environment in which it is used. The chapter is organized into the following sections:

**Topics**

## 1.1 Overview of the Multitasking Executive

The multitasking executive software runs on the MVP's master processor (MP) and provides local control of on-chip parallel processing tasks to present a uniprocessor-like interface to the outside world.

The executive has two primary components:

- A kernel that consists of a library of functions that are called from tasks, and
- A software interface through which tasks on the MP issue commands to the MVP's parallel processors (PPs).

Kernel functions facilitate multitasking, intertask communications, and the allocation of kernel-managed data structures (such as messages). Chapter 2, *The Kernel*, explains the kernel's theory of operation, and Chapter 3, *Kernel Functions*, describes its user-callable functions. Chapter 4, *Internode Message Passing*, explains how the kernel's messaging capabilities support communications with external processors. The MP-resident command-interface software communicates with a command interpreter that runs on the PP. The protocol and structure of the PP command interface are described in Chapter 5, *Parallel Processor Command Interface*.

The executive provides your software applications with efficient and convenient access to the MVP's powerful hardware features. At the same time, the executive is designed to be flexible and unobtrusive. It does not burden you with unnecessary restrictions or impose overhead for features that are not used by applications. The executive provides all necessary support for multitasking and intertask communications, while keeping code size small and execution time short. By far the largest component of the executive is the kernel. In the current release, the kernel functions occupy less than 11K bytes of optimized, compiled C code. A typical application system may use only a subset of these functions. Kernel functions that perform time-critical communications and task-switching operations have been streamlined to reduce execution time.

The multitasking kernel that runs on the MP is message-based; MP-resident tasks use messages both to communicate with each other and to communicate with a general-purpose host processor. A message consists of a fixed-length header followed by a variable-length body. MP-resident tasks can also communicate with other tasks by signaling them through semaphores. Interrupt service routines typically use signals, rather than messages, to inform MP tasks of interrupt events.

A typical MP task is a server routine that processes a stream of requests from a client process that runs on a host processor. The host could be a UNIX workstation, for example. An MP-resident server task might perform graphics, image processing, or video compression/decompression operations requested by a client.

As defined within the kernel, a task is a program that can run concurrently with other programs. A task can allocate memory or can share resources such as PPs with other tasks. If the program is written to be reentrant, two or more tasks can be instances of the same program. A task is scheduled for execution by the task scheduler routine within the kernel. The program on which the task is based is written in the form of a C function that can be called from the task scheduler.

Intertask communications and synchronization are accomplished with messages and semaphores. A task can wait at a port for a message, and it can wait at a semaphore for a signal. The length of a message is application-dependent. In a multiprocessor system, a task can send a message to a port located in another processor node.

A task can wait on multiple events simultaneously. The task monitors the status of a group of up to 32 ports and semaphores in any combination. The arrival of a message or signal at any port or a semaphore in the group is an event that causes the task to be scheduled to execute again. The kernel's event mechanism serves a purpose similar to that of the UNIX *select* system call or the Mach *port set* facility.

The executive allows higher priority tasks to preempt lower priority tasks during interrupts and during calls to kernel functions that send messages or signals. The executive supports round-robin scheduling among tasks of equal priority.

Most code for the executive is written in the C language. The C compiler for the MVP's MP can be configured to generate either big- or little-endian code. The user-callable functions within the kernel are designed to be called either by C programs or by assembly language programs that follow the C conventions regarding function calls.

The release package for the MVP multitasking executive includes source code as well as object code. The source code is included not only to help developers better understand how the executive operates, but also to serve as a starting point for developers who need to customize the executive to meet their specific requirements.

The source code for the executive is organized into uniformly small modules, each of which performs a single, well-specified operation. The code is also structured to ease the task of porting the kernel and other portions of the executive to a host or other processor that communicates with an MVP device. To this end, the bulk of the code is written in standard C, with assembly code and system-dependent features kept to a minimum and isolated in a few modules.

## 1.2 MVP Processing Environment

The MVP is a single-chip device that contains a master processor (MP) and one or more parallel processors (PPs). The number of PPs depends on the MVP device version. Each PP is an advanced digital signal processor (DSP) core that can be programmed to perform a variety of high-speed operations on image data. The executive runs on the MP, which is a state-of-the-art, general-purpose RISC (reduced instruction set computer) processor with fast floating-point hardware. The MP coordinates the work done by the other processors within the MVP. The MP and PPs are tightly coupled and communicate with each other through a number of on-chip RAM modules. The processors access the RAM modules in parallel through a high-speed crossbar switching network.

A host processor typically uses the MVP as a coprocessor to perform computing-intensive video, imaging, audio, and graphics operations. These are some of the applications that the MVP is designed to accelerate:

- Compression and decompression of video and audio signals for video conferencing and multimedia
- Image warping, filtering, enhancement, and feature recognition
- Optical character recognition and document capture
- 2-D and 3-D graphics

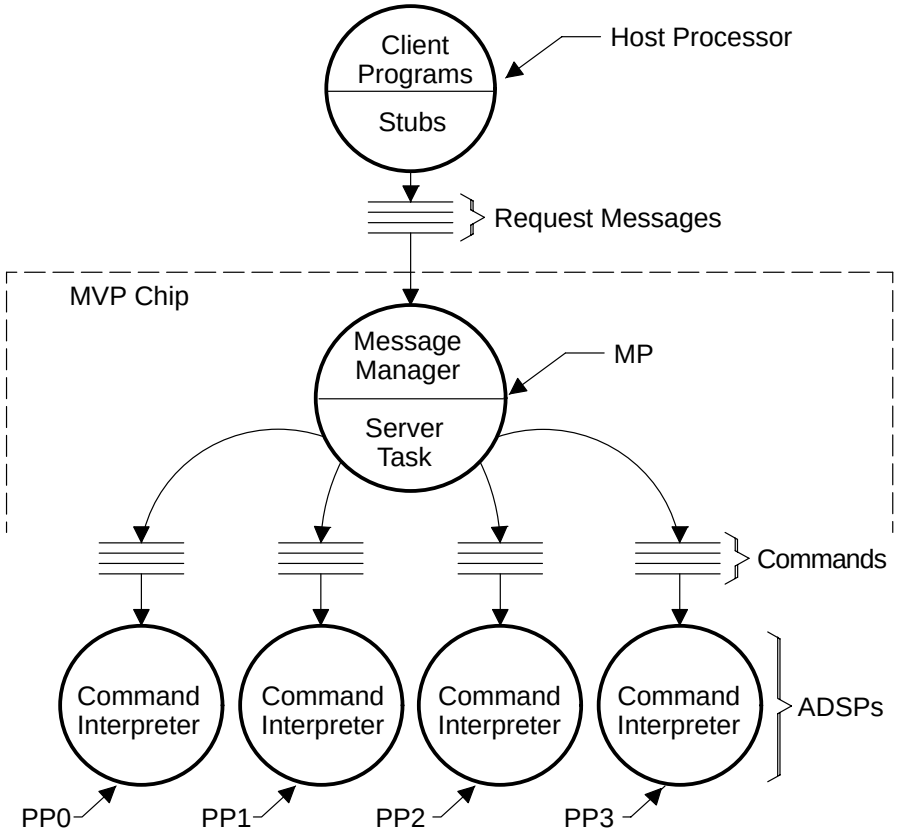
Application programs typically execute on a host processor and off-load processing work to MVP-resident server tasks. Because the MVP's MP is a general-purpose RISC processor, application programs can also be written to execute on an MVP configured as a standalone processor.

In a typical system configuration, an MVP communicates with a host processor over a parallel bus. Alternatively, an MVP-based terminal may communicate with one or more remote hosts over a network. In either instance, the MVP's MP handles all communications with client processes on the host. The MVP's PPs have no direct communication with the host.

### 1.2.1 Flow of Data and Control

Figure 1–1 shows how the processing load for a typical series of host requests can be distributed among the processors on the MVP chip to achieve faster execution. On the host processor, stub routines convert function calls from client programs into request messages. The requests are transmitted to the MP to be serviced. The figure shows a queue of such requests at the interface between the host and MP. An MP-resident message manager routes the requests to the appropriate server tasks on the MP.

Figure 1–1. Flow of Data and Control



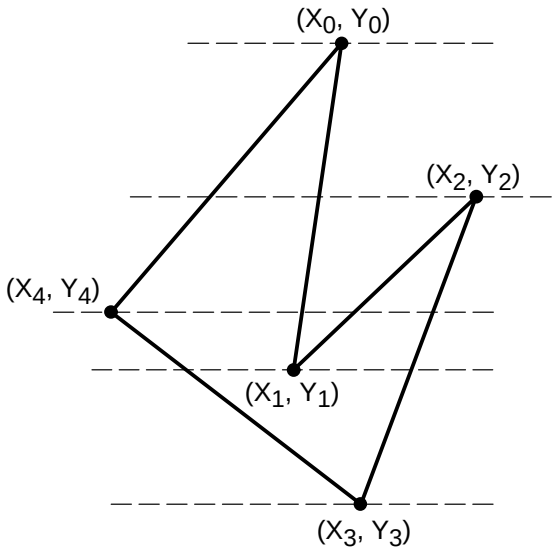

The MP, in turn, off-loads a portion of its processing load by issuing commands to the PPs. Figure 1–1 shows the commands queued at the inputs to the MVP's on-chip parallel processors. Each PP is an advanced DSP with a powerful instruction set for accelerating the execution of imaging and graphics operations. Running on each PP is a command-interpreter program. The PPs are capable of executing commands in parallel with each other. In Figure 1–1, the host, MP, and PPs form a three-stage pipeline for speeding up the execution of host programs.

Your job as an applications programmer is to partition a complex calculation among the MP and PPs. Based on the number of PPs available in the target MVP device, you can break the algorithm into several pieces that map onto the available processors. If you assume that the number of PPs is fixed at the time the program is compiled, you define this mapping to be static. Some algorithms, however, can lend themselves to dynamically adapting the mapping based on the number of PPs available at runtime.

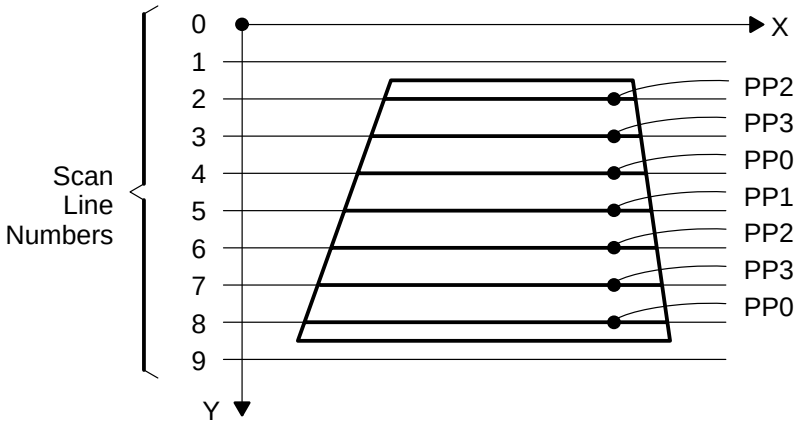
As an example of dynamic mapping, consider Figure 1–2. In this example, an application requires that a client program on the host send requests to a graphics server that runs on the MP. If the MP receives a request from the host to fill a polygon, for instance, it can partition the polygon into some number of trapezoids and then delegate to the PPs the job of actually filling the trapezoids. Figure 1–2 (a) shows how the algorithm might partition a polygon into several nonoverlapping trapezoidal regions by making horizontal cuts, shown as dashed lines in the figure, through each of the vertices. If only a single PP is available, that PP has to do all the work required to fill each trapezoid. One way to partition a trapezoid between two PPs is to have one PP fill the spans that fall on odd scan lines, and the other PP fill the spans on the even lines. If four PPs are available, as shown in the example of Figure 1–2 (b), each PP fills the spans on every fourth scan line. While the PPs are busy filling one trapezoid, the MP can calculate the shape of the next trapezoid.

Figure 1–2. Partitioning a Polygon

(a) The MP cuts a polygon into trapezoids.




(b) The PPs fill one of the trapezoids.



### 1.2.2 Master Processor (MP) Tasks

Any host processor to which the MVP is connected is likely to have a multitasking operating system. The executive must also support multitasking to allow requests from multiple client programs on the host to be serviced concurrently. For instance, the executive may be used in a system in which each of several client programs on the host communicates with a different server task on the MP.

Of course, multitasking is not always required. In some special applications, all client programs may communicate with a single server task on the MP. To handle the more general case, however, the executive must support concurrent communications with multiple servers.

The mechanisms that permit the MP to be shared among multiple tasks are contained within the executive. The executive defines a task as a single thread of execution. Two or more tasks may be instances of the same reentrant program. A task owns permanently allocated resources such as a private data area. A task may also share certain resources (PPs, for example) with other tasks.

Limited mechanisms are available to protect an MP task from other tasks that are badly behaved. A badly behaved task may, for example, write to areas of memory to which it has no access rights. Access rights are enforced primarily through software protocols rather than through hardware mechanisms. All MP-resident tasks share the same address space. The executive makes no distinction between user and supervisor execution modes. If a task is not well behaved, it can cause other tasks on the MVP to crash.

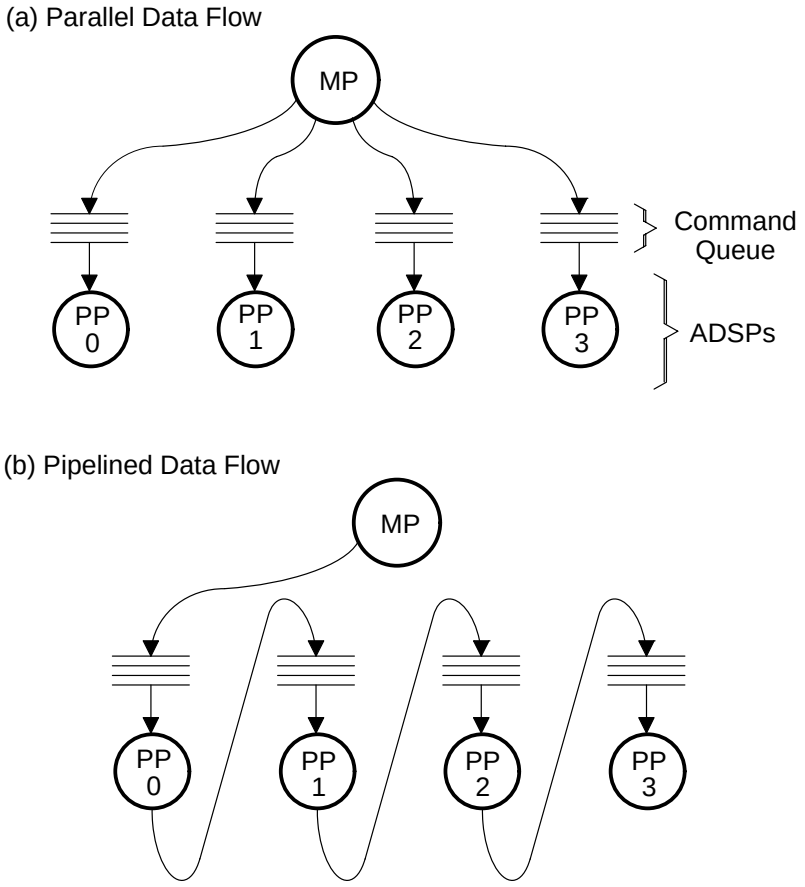
The kernel routines perform numerous checks for errors in the arguments they receive from user programs. The intent of providing these checks is not to facilitate recovery from potentially fatal runtime errors, but rather to identify software problems during the debugging phase of application code development. Before releasing production software that incorporates the kernel, some developers may elect to disable the kernel's software error-checking to eliminate unnecessary runtime overhead.

In the current implementation, the kernel routines and all server tasks that run on the MP are statically linked to form a single, executable module that is downloaded to the MVP. Future implementations may allow individual tasks to be downloaded at runtime and linked dynamically.

### 1.2.3 Parallel Processors (PPs)

The PPs can be configured in software to support a variety of processing flows, as shown in Figure 1–3. Figure 1–3 (a) shows a group of four PPs that are configured to process commands from the MP in parallel. In Figure 1–3(b), the PPs are configured to form a pipeline in which the MP issues commands to PP0, PP0 issues commands to PP1, and so on. Hybrids of these two approaches are also possible. For example, PP0 can be configured to pass commands to PPs 1, 2, and 3.

Figure 1–3. Configuring the PPs




Each command queue shown in Figure 1–3 is located in an on-chip RAM module that is accessible both to the processor that issues the commands and to the processor that interprets the commands. The protocol for transmitting a command is implemented in software, but depends on the control hardware associated with the MVP's crossbar switch to automatically connect each processor to the RAM module it addresses during an access.

At any time, a PP can be owned by at most one MP-resident task. A typical MP task treats its PPs as coprocessors to which it offloads some portion of its processing work. The PPs are the primary resources that are shared among tasks running on the MP.

The server software that executes on the PPs is single-threaded and is less complex than the multitasking system that runs on the MP. It may, in fact, be little more than a simple command interpreter. The software interface between the MP and a PP is embedded within the application software and is largely independent of the MVP executive. For this reason, the structure of this interface may vary significantly from one application to the next. Section 5.4, *An Example Configuration*, has an example of a PP command interface that illustrates the principles involved in its design.

Upon acquiring control of a PP, an MP task may have to build and initialize any PP-resident data structures that it requires to support a specific command protocol. An MP task should own a PP for sufficient time to overcome the overhead of reconfiguring the PP for use as a coprocessor by the task.

In certain instances, however, the overhead of configuring a PP can be reduced significantly when two or more cooperating tasks are able to share a PP in a configuration that is common to both. The extent to which this sharing is feasible depends on the application.

You must write your application software to explicitly specify the ways in which a particular algorithm can be mapped to a variety of PP configurations. The MVP executive itself does not perform any analysis of the algorithm to detect opportunities for parallelism and pipelining.

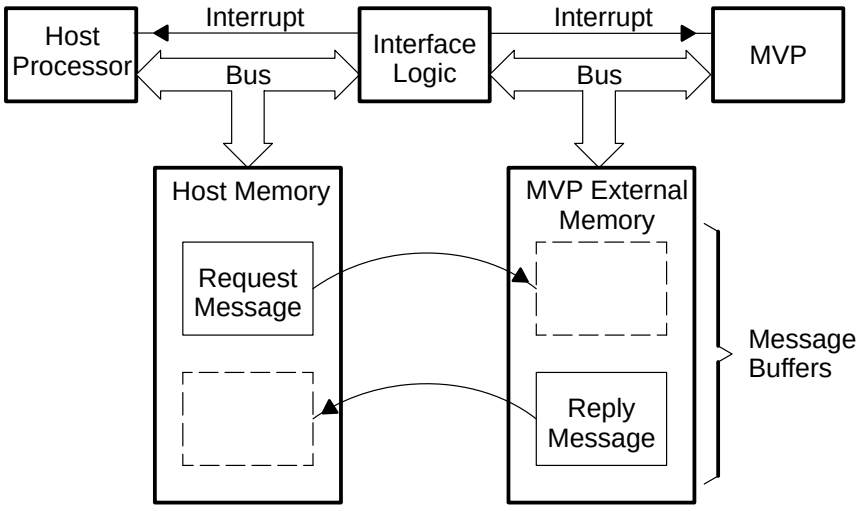
## 1.3 Host Communications

As shown in Figure 1–4, physical communication between the host and MP takes place through message buffers located in the MVP's external (off-chip) memory. If the host and MVP are connected by means of a parallel bus interface, as in this example, the message buffers may be located in a block of dual-ported memory that is accessible to both processors. The host issues a request by copying the request into a message buffer, interrupting the MP, and giving the MP a pointer to the message. The message pointer can be passed through a register contained in the interface logic at the top of Figure 1–4, for example. If the host's request message requires a reply, the MP constructs the reply in a message buffer, interrupts the host, and hands the host a pointer to the message.

A system might use any one of the following mechanisms to physically transport data from a host processor to the MVP's external memory:

- A block of dual-ported RAM that can be accessed by both the host and the MVP
- Bus-to-bus interface logic that allows the host to DMA-transfer data from its memory to the MVP's external memory
- A network over which one or more remote hosts send messages to the MVP

Figure 1–4. Interface Between Host and MVP



Regardless of the method used to transfer the message, the end result is a copy of the host's message that resides in a buffer in the MVP's external memory. The executive accepts this message from the host and delivers it to the appropriate MP server task for processing. The executive simply transfers a pointer to the message — it does not copy the message again.

The messages handled by the executive have two primary components:

- A header that specifies the message's destination address
- A message body that contains an operation code and arguments

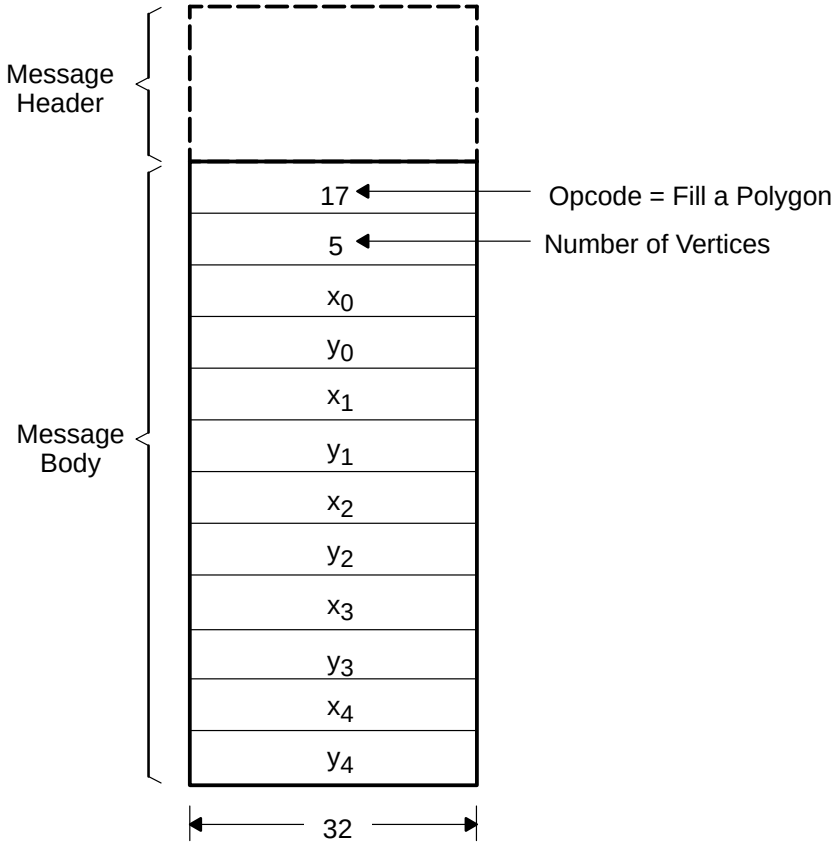
The header contains the information necessary to direct the message to the appropriate server task. The header size is fixed, but the length of the message body may vary depending on the application. The header and message body occupy contiguous locations in the message buffer.

A sample message for a graphics application is shown in Figure 1–5. This message is a request to fill the polygon shown in Figure 1–2 (a). The header is accessed only by the kernel, but the message body is interpreted by the graphics server task.

- The first word in the message body contains the value 17, which, for this example, is the operation code for a request to fill a polygon.
- The second word contains the value 5, which is the number of vertices in the polygon.
- The remainder of the message body contains the x and y coordinates of vertices 0, 1, 2, 3, and 4.

In general, the length and structure of a message body depend on the application.

Figure 1–5. Sample Message



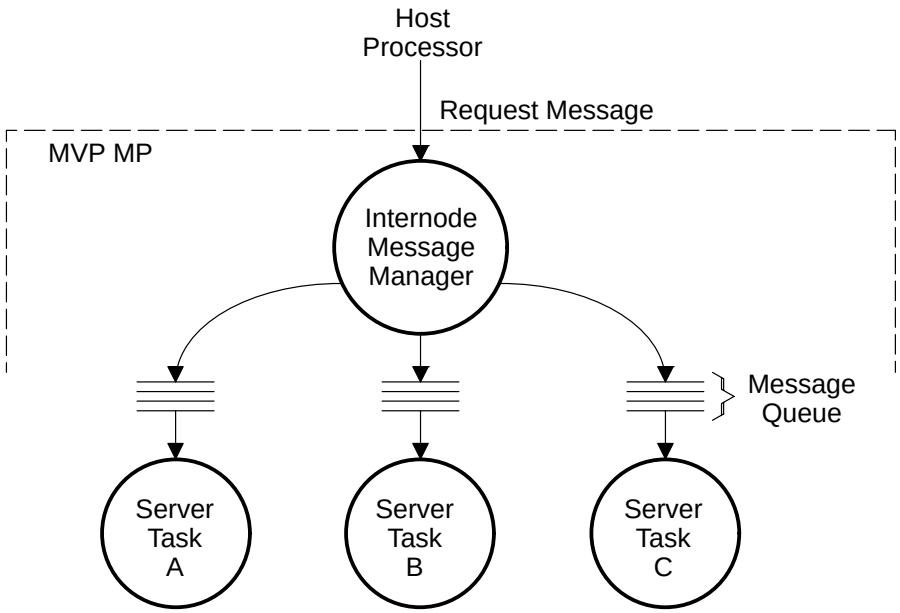

Executing on the MP are one or more server tasks that accept requests from the client programs that run on the host processor. Client programs can communicate with the MP either through a remote-procedure-call mechanism, or by explicitly constructing and sending request messages.

In Figure 1–1, the request messages that are sent from the host processor to the MP are shown forming a single queue. Figure 1–6 shows that the messages actually form multiple queues. Each queue in Figure 1–6 contains the messages sent to a particular server task on the MP. The host sends each message by:

1) Copying the message into a buffer in the MVP's external memory,
2) Loading a pointer to the message into a register in the interface, and
3) Interrupting the MP.

In response to the interrupt, a special MP-resident task, the internode message manager, retrieves the message pointer and routes the message to the MP server task to which it is addressed.

Figure 1–6. Internode Message Manager




If a server task needs to reply to a host request, it routes the reply through the internode message manager. The message manager responds by loading a pointer to the reply message into a register in the interface hardware and interrupting the host, which retrieves the reply. The operation of the internode message manager is described in greater detail in Chapter 4, *Internode Message Passing*.

### 1.3.1 Message-Buffer Allocation by the Host

The example configuration in Figure 1–4 consists of a general-purpose host processor connected to an MVP through a parallel interface. The host passes messages to the MVP through buffers located in a portion of the MVP's external memory that the host also can access. Rather than requiring the MVP to allocate each message buffer needed by the host, the host is responsible for allocating its own buffers.

The host allocates its message buffers from a block of the MVP's external memory that the host itself manages as a private storage heap. At system initialization time, the MVP grants to the host the ownership rights to this block.

To each message allocated from this block, the host assigns a return address (called a **reclamation port ID**) to which the message buffer is to be sent when the message is discarded. This ensures that each message buffer that the host sends to the MVP is eventually returned to the host so it can be reclaimed for later use.

In the event that the host sends request messages to the MVP faster than the MVP can service them, the host may temporarily deplete its supply of message buffers. When this occurs, the host must wait for the MVP to return one or more message buffers to the host before the host can allocate any new message buffers. This process is inherently self-limiting and prevents the MVP memory from being overrun with messages from the host.

### 1.3.2 Remote Procedure Calls

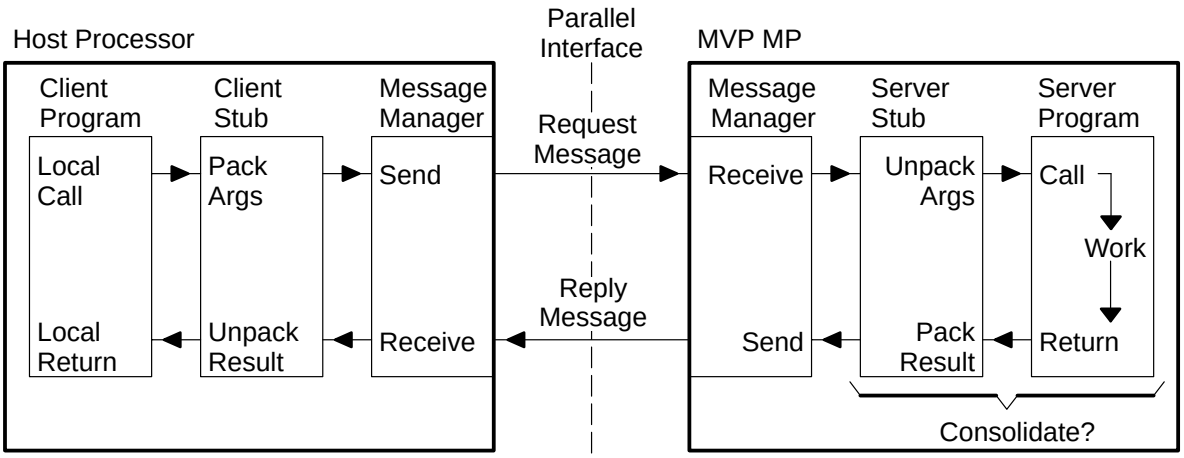
Remote procedure calls (RPCs) provide convenient access to the MVP's processing power while hiding system-dependent details from client programs that run on the host. To a client, an RPC is indistinguishable in form from a standard function call. The call, however, is intercepted by a stub routine on the host that packs the function arguments and operation code into a message that it sends to the MVP. The MVP interprets the message, executes the request, and, if necessary, sends a reply message to the host. A reply is needed if the function called by the client returns a value, for example.

Figure 1–7 shows a model for host-to-MVP communications that is similar to the one proposed in the original paper on RPCs by A.D. Birrell and B.J. Nelson. At the top left corner of the figure, an RPC originates as a standard function call made by a client program that runs on the host processor. Instead of executing the function locally, however, a stub routine intercepts the call and packs the operation code and argument values into a request message. The stub routine passes the message to the host's message manager, which transmits the message over the parallel interface to the MVP. Assuming that the original function call returns one or more values, the client program on the host must wait for its message manager to receive the reply message before proceeding.

On the right side of Figure 1–7, the MP-resident message manager receives the request message from the host processor and hands the message to the local stub routine. The stub unpacks the arguments from the request message and generates a standard function call to the server routine that actually processes the request. When the server routine returns, the local stub packs the result into a reply message that it passes to the message manager. The message manager transmits the reply message over the parallel interface to the host processor.

On the left side of Figure 1–7, the host-resident message manager receives the reply message and passes it to the local stub routine. The stub routine unpacks the result and executes the return from the client program's original function call.

Figure 1–7. Remote Procedure Call




As an option, the right side of the model can be made more efficient by consolidating the server routine and its associated stub routine. With this modification, the combined stub-and-server routine accepts the packed request message as its argument, and generates a packed reply message as its result. This eliminates one layer of overhead from the RPC model.

The fact that procedure calls are executed remotely by the MVP should be transparent to client programs on the host. This requires that the logical sequence of operations seen by the client must be the same as if the calls were executed on the host processor. For example, if a client program draws a blue polygon in a display window and then draws a red polygon in the same window, the red polygon must be drawn on top of the blue polygon wherever the two polygons overlap. In general, the sequence in which a series of commands is sent to an MVP server by a host application program must be preserved.

No particular ordering, however, needs to be maintained between two independent command streams sent to the MVP by two unrelated client programs on the host.

The X Window System™ is an example of an RPC server system that facilitates the pipelining of client requests. Where possible, the X protocols for request messages have been carefully defined to avoid generating replies from the server. This reduces execution time by allowing a series of requests (to draw lines, polygons, and so on) to be pipelined through the system.

When the host does, finally, issue a request that requires a reply, the pipeline is temporarily stalled while the host waits for one or more return values from the server.

The worst-case scenario for a client-server connection is one in which all requests require replies, and execution becomes strictly serial. Under these conditions, a new request can be issued only after a reply has been received to the preceding request.

For the sake of efficiency, a request from the host to the MVP should not be defined to require a reply, except when a reply is necessary to the request's functionality. This principle also applies to the commands that the MP issues to the PPs. A reply is usually necessary only if the requested operation returns one or more values to the client. One exception is a reply that is needed for synchronization; that is, to verify that the server has finished processing all previous requests.

### 1.3.3 Multiprocessor Support

The MVP executive supports the passing of messages between a host processor and an MVP device. In fact, the executive has even broader communications capabilities that allow it to support message passing across multiprocessor systems that consist of multiple MVP devices, for example. In this type of system, an identical copy of the executive runs on each MVP's MP, which can exchange messages with the other MVPs to which it is connected.

The medium for exchanging messages between two processor nodes could be a serial communications link or a shared block of memory in a dual-ported RAM. The means by which messages are transmitted from the memory of one processor to the memory of another are device-dependent and are supported in device drivers outside the kernel of the executive. The kernel provides the addressing mechanisms needed to route messages between processor nodes.

A multiprocessor system can combine MVP devices with other types of processors. In this type of system, you can port the kernel of the MVP executive to the other processors to facilitate a more symmetrical software interface between the processors and the MVP. Porting is straightforward because the kernel is written primarily in C. The machine-dependent portions of the code are modular and relatively small.

Chapter 2

# The Kernel

The kernel of the MVP multitasking executive is a software library of user-callable functions that provide basic program control, as well as intertask communication and synchronization. The kernel serves as a foundation on which to build sophisticated system services.

This chapter describes the functionality and theory of operation of the kernel. It describes not only the user interface, but also the kernel's internal structure.

**Topics**

## 2.1 User-Callable Functions

The kernel's primary capabilities include:

- Message passing
- Semaphore control
- Multiple-event monitoring
- Exception and error notification
- Private-context support
- Multitasking

To facilitate modular system design and present a clearly defined interface, additional kernel-supported system capabilities are implemented outside the kernel. These capabilities might typically include:

- Interrupt handling
- Dynamic memory allocation
- Timers and clocks
- Resource management
- File I/O control

Kernel services are provided to you in the form of a library of functions that can be linked with applications software. In the current implementation, an object library containing the compiled kernel functions is linked with the task programs to form a single load module. Future versions of the kernel may support features such as dynamic linking of task-specific software at runtime.

All of the kernel's user-callable functions begin with the prefix *Task* (for example, *TaskSendMsg*). Table 2–1 through Table 2–7 divide the kernel's user-callable functions into several groups, according to their functionality.

The message functions in Table 2–1 allow tasks to communicate through messages. Tasks exchange messages through ports. Messages and ports can be allocated and deallocated. Multiple tasks can send messages to the same port, and multiple tasks can wait at the same port for messages. In a multiprocessor system, the kernel supports the transmission of messages from one processor to another. Both blocking and nonblocking functions are available for receiving messages. The length of a message is application-dependent.

Table 2–1. Message Functions

| Function Name | Description | See Page |
|---|---|---|
| TaskAcceptMsg | Accept a Message Without Waiting | EX:3-5 |
| TaskAllocMsg | Allocate a Message | EX:3-7 |
| TaskClosePort | Close a Port | EX:3-13 |
| TaskFreeMsg | Free a Message Buffer | EX:3-17 |
| TaskGetMsgSize | Get the Size of a Message | EX:3-21 |
| TaskGetReplyPort | Get a Message's Reply Port | EX:3-24 |
| TaskInitMsg | Initialize a Message | EX:3-27 |
| TaskOpenPort | Open a Port | EX:3-31 |
| TaskReceiveMsg | Receive a Message | EX:3-35 |
| TaskReclaimMsg | Reclaim a Message | EX:3-36 |
| TaskRelayMsg | Relay a Message | EX:3-37 |
| TaskResizeMsg | Resize a Message | EX:3-39 |
| TaskRouteMsg | Route a Message | EX:3-41 |
| TaskSendMsg | Send a Message | EX:3-43 |
| TaskSetMsgRoute | Set a Message-Routing Port | EX:3-44 |
| TaskSetReplyPort | Set a Message's Reply Port | EX:3-47 |
| TaskValidatePort | Validate a Port ID | EX:3-52 |

The semaphore functions in Table 2–2 operate on counting semaphores. Semaphores can be allocated and deallocated. Multiple tasks can signal the same semaphore and multiple tasks can wait at the same semaphore for signals. Semaphores are used to synchronize tasks both to external events and to each other. They also coordinate the sharing of resources among tasks.

Table 2–2. Semaphore Functions

| Function Name | Description | See Page |
|---|---|---|
| TaskCheckSema | Check a Semaphore Without Waiting | EX:3-11 |
| TaskCloseSema | Close a Semaphore | EX:3-14 |
| TaskOpenSema | Open a Semaphore | EX:3-32 |
| TaskResetSema | Reset a Semaphore | EX:3-38 |
| TaskSignalSema | Signal a Semaphore | EX:3-48 |
| TaskWaitSema | Wait at a Semaphore for a Signal | EX:3-55 |
| TaskValidateSema | Validate a Semaphore ID | EX:3-53 |

The multiple-event functions listed in Table 2–3 allow a task to monitor multiple events simultaneously. The kernel recognizes two types of events:

- A **message event**, which is the arrival of a message at a port
- A **signal event**, which is the arrival of a signal at a semaphore

Associated with each task is an event register containing 32 event flags. Each flag can be bound to a port or semaphore to monitor the availability of a message or signal. A task can wait on a set of selected events; the first of these events to occur makes the task ready to execute.

Table 2–3. Multiple-Event Monitoring Functions

| Function Name | Description | See Page |
|---|---|---|
| TaskBindPort | Bind a Port to an Event Flag | EX:3-9 |
| TaskBindSema | Bind a Semaphore to an Event Flag | EX:3-10 |
| TaskGetFreeEvents | Get the Free Event Flags | EX:3-19 |
| TaskPollEvents | Poll the Event Flags | EX:3-33 |
| TaskUnbindPort | Unbind a Port From an Event Flag | EX:3-50 |
| TaskUnbindSema | Unbind a Semaphore From an Event Flag | EX:3-51 |
| TaskWaitEvents | Wait for a Selected Event | EX:3-54 |

The exception and task error functions in Table 2–4 support exception handling and monitor task errors during kernel calls. An **exception** is the software analog of a hardware interrupt. Just as an interrupt sent to a processor invokes an interrupt service routine, an exception in a task invokes an exception handler. **Task errors** are errors that occur within a task during a call to a kernel function. When a kernel function call fails, the calling task can interrogate the kernel to determine the precise nature of the error.

Table 2–4. Exception and Task Error Functions

| Function Name | Description | See Page |
|---|---|---|
| TaskClearExcep | Clear Exceptions | EX:3-12 |
| TaskGetExcep | Get the Pending Exceptions | EX:3-18 |
| TaskInstallHandler | Install an Exception Handler | EX:3-29 |
| TaskGetLastError | Get the Last Task Error | EX:3-20 |
| TaskRaiseExcep | Raise an Exception | EX:3-34 |
| TaskSelectExcep | Select Exceptions | EX:3-42 |

The private-context functions in Table 2–5 allow libraries and groups of related functions to maintain their own private contexts within the tasks that use them. A typical task is an application program that calls functions in one or more libraries. Each library may need to keep track of certain state information that it would prefer to hide from the program and from the other libraries used by the program. The private-data mechanism not only protects the private data for the sake of the library; it also spares the application program from the inconvenience of having to incorporate implementation-specific library parameters into its own visible context.

Table 2–5. Private-Context Functions

| Function Name | Description | See Page |
|---|---|---|
| TaskAddFuncList | Add to the Init- and Exit-Lists | EX: 3-6 |
| TaskAllocPrivate | Allocate a Word of Private Data | EX: 3-8 |
| TaskGetPrivate | Get a Private-Data Word | EX: 3-23 |
| TaskSetPrivate | Set a Private-Data Word to a Value | EX: 3-46 |

The multitasking and interrupt servicing functions in Table 2–6 support concurrent program execution. Task scheduling is priority-based and higher-priority tasks automatically preempt lower-priority tasks. A task can be suspended and resumed and can yield the processor to another task of the same priority. An ISR (interrupt service routine) executes within the context of the task it interrupts.

Table 2–6. Multitasking and Interrupt-Servicing Functions

| Function Name | Description | See Page |
|---|---|---|
| TaskCreate | Create a Task | EX: 3-15 |
| TaskExit | Exit the Calling Task | EX: 3-16 |
| TaskGetPriority | Get the Calling Task's Priority | EX: 3-22 |
| TaskGetTask | Get the Task ID | EX: 3-25 |
| TaskResume | Resume a Suspended Task | EX: 3-40 |
| TaskSetPriority | Set the Calling Task's Priority | EX: 3-45 |
| TaskSuspend | Suspend a Task | EX: 3-49 |
| TaskYield | Yield to Another Task | EX: 3-56 |

The miscellaneous functions in Table 2–7 initialize the kernel following power-up, provide information about the calling task, and allow you to install your own memory allocation functions.

Table 2–7. Miscellaneous Functions

| Function Name | Description | See Page |
|---|---|---|
| TaskGetTaskArg | Get the Task Argument | EX:3-26 |
| TaskInitTasking | Initialize the Kernel | EX:3-28 |
| TaskInstallMalloc | Install the Memory Allocation Functions | EX:3-30 |

## 2.2 IDs for Ports, Semaphores, and Tasks

The kernel assigns an ID (identifier) to each kernel resource—port, semaphore, and task—at the time that the resource is opened or created. Application programs use the ID to refer to the resource in subsequent kernel calls. The ID is deleted when the port or semaphore is closed or the task exits.

Ports, semaphores, and tasks are referred to collectively as **kernel resources**. The kernel maintains a separate, fixed-length table to keep track of each resource type, as opposed to maintaining a single large table in which all resource types are merged.

The length of a resource table determines the maximum number of resources of a particular type that can be open simultaneously. When a resource is closed, the kernel removes it from the table, freeing a slot in the table that can be used for a new resource.

In the current implementation of the kernel, the storage for resource tables is allocated statically. The default table lengths are specified in C header file *task.h*, which is included in the executive software release. If you anticipate your resource usage growing beyond these fixed limits, you should specify larger table sizes to ensure that your applications code works under full loading.

As shown in Figure 2–1, each slot in a resource table is a pointer to a resource of the appropriate type. If a table slot is unused, it contains a null pointer value. The structure for a port, semaphore, or task descriptor begins with a standard resource header consisting of a 32-bit *link* field and a 32-bit *id* field. The *link* field points to the next resource in a linked list of resources of the same type. The *id* field contains the resource's ID.

Figure 2–1. A Kernel Resource Table

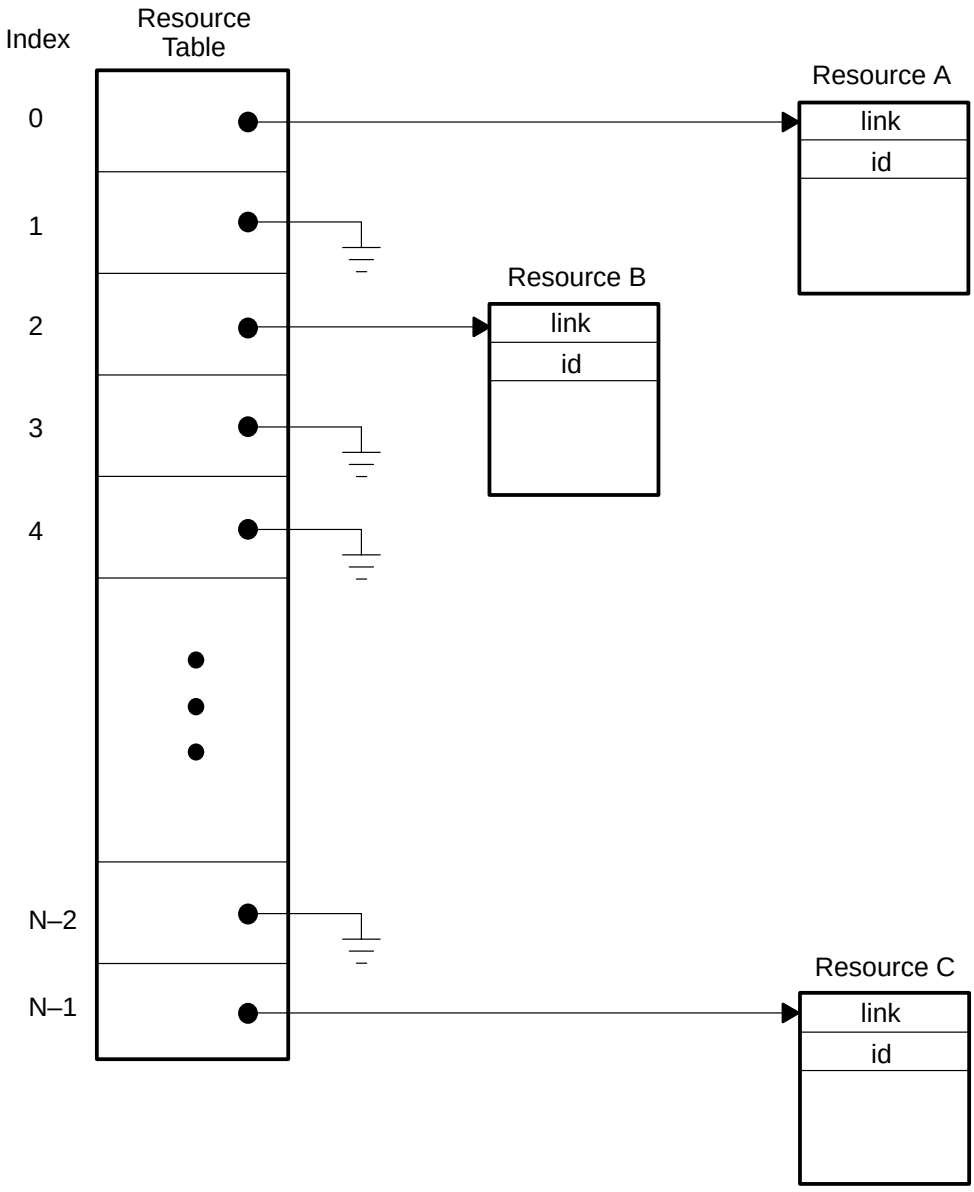
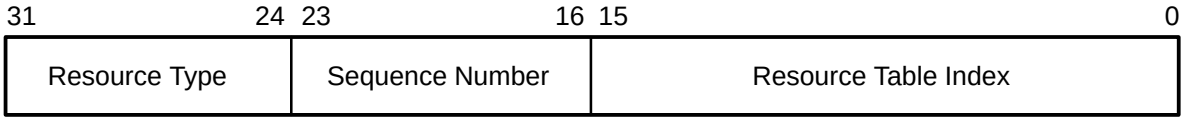



Figure 2–2 shows the format for a resource ID. The ID is 32 bits long and consists of a 16-bit table index, 8-bit sequence number, and 8-bit type code. The table index is the index into the appropriate resource table, as indicated in Figure 2–1. The sequence number is a semiunique number assigned to each successive resource added by the kernel. The type code indicates whether the resource is a port, semaphore, or task.

Figure 2–2. Format for Kernel Resource ID




Resource Type:

| Code | Meaning |
|---|---|
| –1 | Port (Local Node) |
| –2 | Semaphore |
| –3 | Task |
| 0 to 127 | Global Port's Processor Node |

### 2.2.1 Node-Local and Node-Global Port IDs

A port can reside either in the local processor node or in a foreign node. As indicated in Figure 2–2, a port ID can be expressed in one of two formats: node-local or node-global. The 8-bit resource type field in a node-local port ID is set to a value of –1 to indicate that the port is local. In a node-global port ID, the MSB is a 0, and the remaining 7 bits specify a node number in the range 0 to 127. This range effectively limits the number of nodes in a multiprocessor system to 128.

A task can use a node-global port ID to send a message not only to a foreign port but to a local port as well — as long as the local node number is known. Node-local port IDs are typically more convenient to use for local message transfers, however, because they do not require that the local node number be specified explicitly.

The kernel does not define node-global versions of semaphore and task IDs. This means that message passing is the only internode communications mechanism that the kernel supports directly.

### 2.2.2 Sequence Numbers

The sequence number in Figure 2–2 is useful for distinguishing successive uses of the same table index by different resources. Each time the kernel adds a new resource to one of its tables, it increments the sequence number before inserting it into the ID for the new resource.

Without the sequence number, the resource type and table number fields in Figure 2–2 are sufficient to ensure that each current resource has a unique ID. The sequence number increases the likelihood that IDs are unique over the entire history of the system from the time of power up.

For example, if a port is opened in slot 17 and later closed, the slot becomes available for a new port. The only portion of the identifier that distinguishes the ID for the old port from that of the new port is the sequence number.

This system is not entirely foolproof since the sequence number is only eight bits and begins to repeat after 256 IDs have been generated. The primary role of the sequence number, however, is to help identify bugs during development, and an 8-bit sequence number makes undetected ID errors extremely unlikely.

### 2.2.3 Null ID Value

The value –1 is reserved to indicate a null ID. For this reason, a resource table index is not permitted to have the value 65 535. This limits the maximum length of a resource table to 65 535, rather than 65 536 slots.

In fact, statically allocating a table this large is costly in terms of the storage required. Practical considerations typically limit table lengths to smaller values.

### 2.2.4 Generating New Resource IDs

The *TaskOpenPort* and *TaskOpenSema* functions open new ports and semaphores; the *TaskCreate* function creates new tasks. Each of these functions assigns to a new resource an ID that is used to refer to the resource in subsequent kernel calls.

When opening a new kernel resource, the kernel allows you to use either of the following methods to assign a 32-bit ID to the resource:

- Let the kernel automatically generate a valid ID for you.
- Specify your own ID, which the kernel accepts only after verifying that it is valid.

By following the format shown in Figure 2–2, you can specify your own IDs for certain public (or well known) ports, semaphores, and tasks. These can be specified as compile-time constants that are known *a priori* to all programs that need to use them. The kernel accepts a user-supplied ID as valid only if its type field is correct and its index corresponds to an empty slot in the resource table. It always accepts any value specified for the sequence number.

On the other hand, tasks may need to keep certain resources private, and a convenient way to do this is to allow the kernel function (*TaskOpenPort*, *TaskOpenSema*, or *TaskCreate*) to generate the ID for you. If you specify a value of –1 for the ID, the function automatically generates and returns a valid ID. To do this, the function searches the appropriate resource table for the next available slot. It also increments the sequence number.

When the function generates an ID for you, it begins its search for the next available slot at the high end of the resource table. In other words, if the table contains *n* slots, the function searches the slots in the order *n*–1, *n*–2,...,0. This search pattern is based on the assumption that if you choose to generate any of your own IDs, you will probably prefer to select your indices from the low end of the table: 0, 1, and so on.

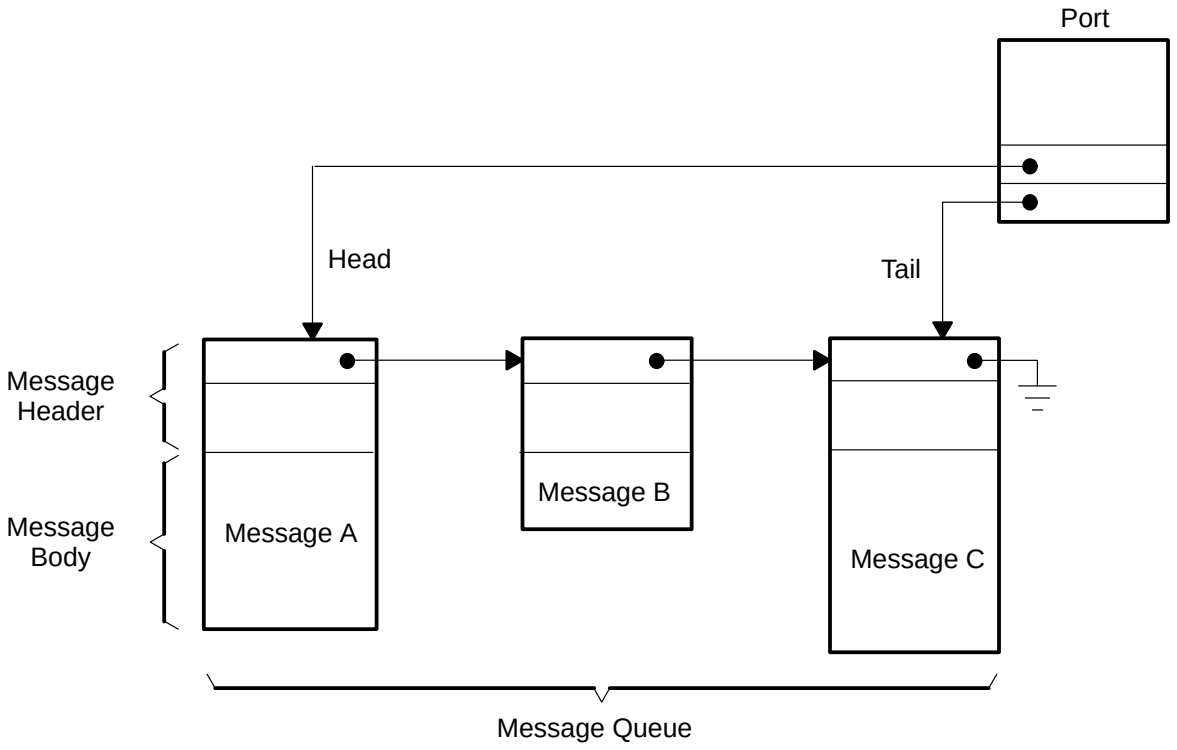
## 2.3 Messages and Ports

Messages are the basis both for intertask communications on the MP and for communications with intelligent devices external to the MVP. A message is a data structure that consists of a fixed-length header followed immediately by a body whose length is application-dependent. The minimum size for a message is 0 bytes; this is a message that consists of a message header only. The maximum message size is 32 768 bytes.

Messages are sent through ports rather than directly from one task to another. Multiple tasks can send to and receive from the same port. In a multiprocessor system, a task can receive a message only from a local port (that is, a port located in the local processor node) but can send a message to a port in another processor node.

A port is a queue of messages. Figure 2–3 shows a port at which three messages are queued. In this example, the queue is a linked list of three messages terminated by a null pointer. The port contains pointers to the head and tail of the message queue. A message sent to the port is appended to the tail of the queue, and a message received from the port is removed from the head of the queue.

Figure 2–3. Three Messages Queued at a Port

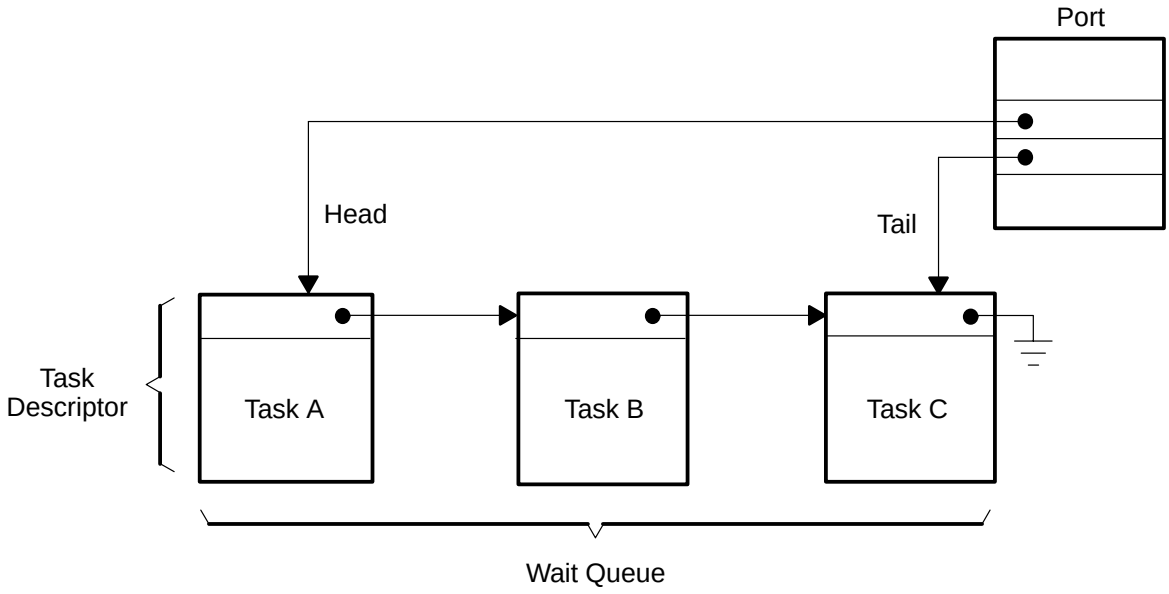


A series of messages sent from a task to a local port are queued at the port in precisely the order in which they were sent. Whether this is the case for internode messages is implementation-dependent.

If a task attempts to receive a message from a port but the message queue is empty, the task can wait at the port until a message arrives. Figure 2–4 shows a wait queue consisting of three tasks that are waiting for messages. The wait queue in this example is a linked list of three task descriptors terminated by a null pointer. The port contains pointers to the head and tail of the wait queue.

A task in the wait queue is blocked from further execution until a message becomes available for it to receive. Messages are given to tasks on a first-come-first-served basis, regardless of the tasks' relative priorities. When a new task begins waiting at the port, its task descriptor is appended to the tail of the wait queue. When a message arrives, a task descriptor is removed from the head of the queue; the task is given the message and is scheduled to start execution as soon as the processor becomes available.

Figure 2–4. Three Tasks Waiting at a Port



### 2.3.1 Message Structure

As shown in Figure 2–5, a message consists of a message header and a message body that reside in contiguous areas of memory. The message's fixed-length header is used by the kernel but is transparent to application programs. The length of the message body is application-dependent, as is the information it contains.

To make the header structure transparent to application programs, all kernel routines accept and return pointers to message bodies, rather than to message headers. Given a message body pointer, a kernel function accesses the header structure at a negative offset from the body.

Logical messages can be distinguished from the physical buffers in which they reside. A physical buffer is the block of memory allocated to contain a message, whereas the logical message is the information contained in the buffer. The same physical buffer may be used repeatedly to send a succession of logical messages. In a loosely coupled multiprocessor system, a logical message sent to a foreign node may have to be copied from one physical buffer to another one or more times.

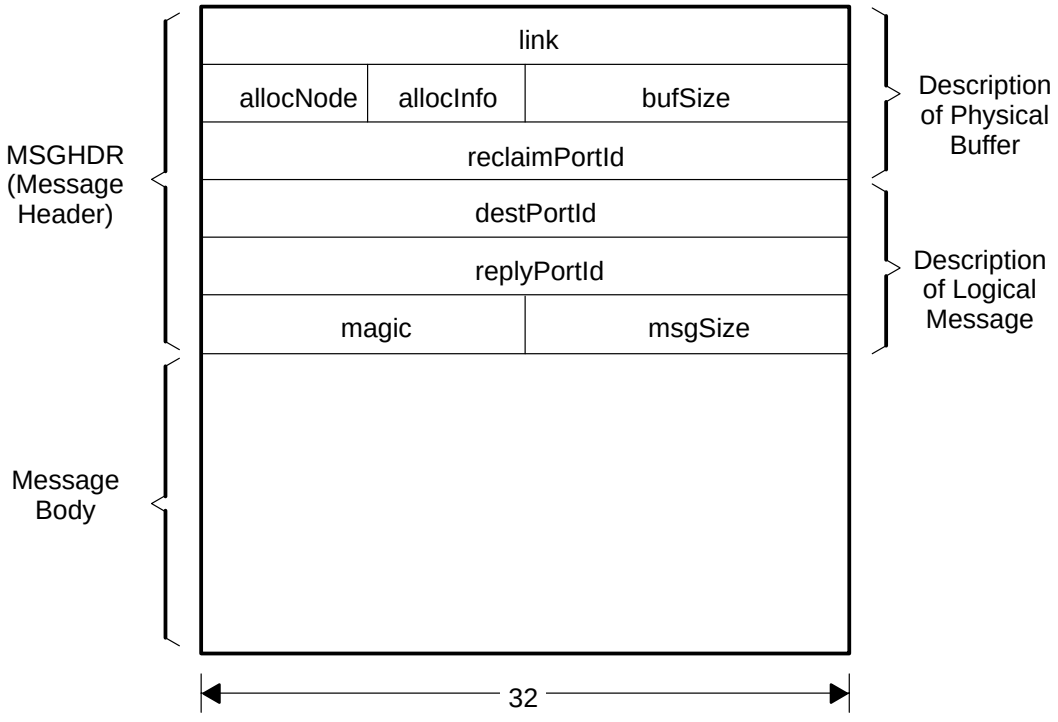
The MSGHDR (message header) structure in Figure 2–5 contains information describing both the physical buffer and the logical message it contains. The physical buffer is described by the following fields:

- The 32-bit *link* field contains a pointer to the next message in a linked list of messages in queue at a port.
- The 8-bit *allocNode* field contains the number of the processor node in which the message buffer was allocated. This field may contain a value of –1 to indicate that the node in which the message currently resides is also its allocation node.
- The 8-bit *allocInfo* field contains implementation-specific information about the manner in which the storage for the message was allocated.
- The 16-bit *bufSize* field specifies the size in bytes of the physical buffer in which the message resides. This buffer contains the message header, message body, and any additional storage that was allocated but is unused by the current message. The header always resides at the very beginning of the buffer.
- The 32-bit *reclaimPortId* field identifies the message's reclamation port, which is the port to which the message buffer is returned after the logical message it contains has been discarded. A task uses this field to maintain its own pool of message buffers.

The logical message is described by these fields:

- The 32-bit *destPortId* field identifies the message's destination port. This field is loaded only when a message is transmitted across a multiprocessor from one processor node to another.
- The 32-bit *replyPortId* field identifies the message's reply port. If a message requires a reply, the reply message is sent to this port.
- The 16-bit *magic* field contains a reserved value that is the same for all messages. The kernel does not accept a message as valid unless its header contains this value.
- The 16-bit *msgSize* field contains the length in bytes of the logical message body. The logical message may not occupy the entire physical buffer allocated for the message.

Figure 2–5. Message Structure




Note that the fields in Figure 2–5 are shown according to big-endian conventions. In other words, the *allocNode* field precedes the *allocInfo* field, and so on.

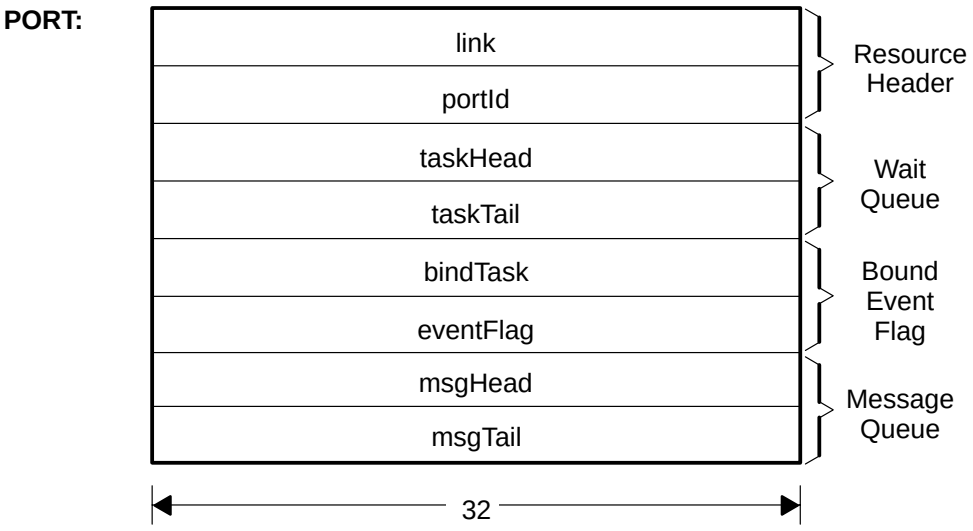
Including the reply port ID in the message header is convenient but may not be strictly necessary. The application software could store the reply port ID somewhere in the message body instead. However, when the message is sent to a port in a foreign processor node, the kernel automatically converts the *replyPortId* field from node-local to node-global format if necessary. If the reply port were not contained in the header, the application program itself would be responsible for ensuring that the reply port ID was in node-global format before the message left the sending node. Having the kernel handle this automatically is simpler and less error prone.

### 2.3.2 Port Structure

Figure 2–6 shows the structure of a port. The fields are defined as follows:

- The 32-bit *link* field contains a pointer to the next port structure in a linked list of ports.
- The 32-bit *portId* field contains a port ID that uniquely identifies the port.
- The 32-bit *taskHead* and *taskTail* fields contain pointers to the task descriptors at the head and tail of the wait queue. If the wait queue is empty, *taskHead* contains a null pointer value, and *taskTail* is undefined.
- The 32-bit *bindTask* field contains a pointer to the task descriptor containing the event flag to which the port is bound. If the port is not bound to a task, this field contains a null pointer value.
- The 32-bit *eventFlag* field is a mask containing all 0s except for a single bit with a value of 1 that indicates the position of the bound event flag in the event register.
- The 32-bit *msgHead* and *msgTail* fields contains pointers to the message headers at the head and tail of the message queue. If the message queue is empty, *msgHead* contains a null pointer value, and *msgTail* is undefined.

Figure 2–6. Structure of a Port




## 2.3.3 Message Allocation

The *TaskAllocMsg* function allocates a message of the specified size, initializes its header, and returns a pointer to the message body. Depending on the implementation, a task may be allowed to allocate a message while any heap storage remains available in memory. This is the case if the heap storage from which messages are allocated is the same as that from which the standard C function, *malloc*, allocates memory.

As an alternative, a task can directly allocate the storage for the message buffer itself. The task then calls the *TaskInitMsg* function to initialize the header area at the beginning of the buffer, and the function returns a pointer to the message body. The purpose of this function is to allow communications software to locate message buffers in specific areas in memory (for example, within a block of dual-ported memory) to facilitate communications with external processors.

In a tightly coupled multiprocessor system, two or more processors can directly access the same message buffer. The storage for a message can be deallocated, however, only by the processor that originally allocated the message.

### 2.3.4 Message Buffer Pools and Reclamation Ports

Instead of dynamically allocating a buffer from the heap for each new message, a task can maintain a pool of message buffers. When the task needs to send a message, it allocates a buffer from the pool. When the receiving task is ready to discard the message, it calls the *TaskReclaimMsg* function to send the message buffer back to its reclamation port so that it can be reused. All the message buffers in a pool are typically the same size and share the same reclamation port.

The *TaskAllocMsg* and *TaskInitMsg* functions allow a reclamation port to be associated with each message buffer. When the message is discarded, its buffer is automatically sent to the reclamation port, whose ID is stored in the message header.

A message's reclamation port must reside in the processor node in which the storage for the message was allocated. In a tightly coupled multiprocessor system, a shared memory buffer containing a message may be passed from one processor node to another by simply exchanging pointers. If a message is discarded in a node other than the one in which the message buffer originated, the buffer may have to be transmitted across one or more node boundaries to reach its reclamation port. The kernel is designed to perform this routing automatically.

Maintaining a pool that contains a finite number of message buffers ensures that a task that sends messages faster than they are received does not exhaust heap storage. Once the sending task has emptied its message pool, it waits for one of its message buffers to return to the reclamation port before it sends another message.

The *TaskReclaimMsg* function returns a message buffer to the heap only if no reclamation port was assigned when the message was originally allocated. Another function, called *TaskFreeMsg*, returns a message to the heap regardless of whether the message has a reclamation port. This function provides a means for a task to deallocate the messages in a message pool that it no longer requires.

A valid reclamation port must be specified for every message initialized by the *TaskInitMsg* function. Otherwise, if the message is passed to the *TaskReclaimMsg* function to be reclaimed, that function will be able neither to reclaim nor deallocate the storage for the message.

### 2.3.5 Routing Ports

The kernel provides the addressing mechanisms needed in a multiprocessor system to direct a message from a task in one processor node to a port in another processor node. A task calls the *TaskRouteMsg* function to send a message to a port in a foreign processor node. The function loads the destination port ID into the message's header before the message leaves the node it was sent from.

The interface between a pair of processor nodes is managed on each side by an **internode message manager**. The message manager is a task that manages communications between the local node and the foreign node on the other side of the interface. Messages traveling in both directions across a particular interface are routed through the associated message manager.

The kernel checks the destination port of each message that might be bound for a foreign port. If the destination lies in a foreign node, the kernel automatically routes the message through a specially designated local port. This port is the local **routing port** for all messages bound for that particular node. The internode message manager waits for a message to arrive at the routing port for a particular node. When one arrives, the message manager receives the message and transmits it through the interface hardware to the processor node on the other side. This node may or may not be the message's final destination.

The kernel keeps the information on all the local routing ports in its **routing table**. The table is of some fixed length *n* and contains the routing ports for nodes 0 through *n*–1. Each entry in the table is the 32-bit port ID of a local routing port and the index into the table is the destination node number.

If the routing port for a particular node is not known, the corresponding table entry is set to –1, the null ID value. When the kernel begins running following power up, it sets all routing table entries to –1. Each subsequent call to the *TaskSetMsgRoute* function loads the routing table entry for a particular node with the ID of a local routing port. A task should not attempt to send a message to a port in a foreign node until the table entries for that node and for the local node have both been loaded.

If a message is sent to a foreign processor node and the message's reply port ID is specified in node-local format, the kernel automatically converts the reply port ID to node-global format before queueing the message at its routing port. In other words, if an internode message originates in a node *n*, the eight MSBs of the reply port ID in the message header are changed from –1 to *n* before the message leaves the originating node. (Refer to Figure 2–2).

The *allocNode* field is treated in similar fashion if the original message buffer can be accessed directly by another processor, in which case the logical message does not need to be copied to another buffer. The kernel checks the *allocNode* field to see if it is expressed in node-local format; if so, it overwrites the field with the local processor's node number before placing the message in its routing port.

Only a message redirected by the kernel should be sent to a routing port. Application programs should avoid sending messages directly to routing ports.

As long as communications are restricted to local ports, application programs need not specify any routing-table information. In this case, all the destination port IDs specified to the *TaskRouteMsg* function must be given in node-local format.

### 2.3.6 Message Copying

An internode message manager should avoid message copying where possible. If a message is sent between two tasks in the same processor node, only a pointer needs to be exchanged. Transferring a pointer is more efficient than copying the entire message from one buffer to another. Similarly, if a message is sent between two tightly coupled processors through a block of shared memory, only a pointer needs to be exchanged.

On the other hand, if a message is sent between two loosely coupled processor nodes, it must be copied from one processor's memory to the other's. In this case, the information that is contained in the logical message that is copied must be carefully distinguished from the physical buffers themselves.

As shown in Figure 2–5, the parameters located in the first half of the message header describe the message's physical storage buffer, while those in the second half describe the logical message. When a message is copied from one buffer to another, the message header in each buffer already contains the parameters describing the physical storage. Only the logical-message portion of the header is copied.

A message buffer may be larger than is necessary to hold the logical message that it contains. This could happen for either of these reasons:

- The kernel's message allocator may allocate storage in multiples of some minimum block size (for example, the cache subblock size).
- A task may reuse the same message buffer to send a series of logical messages of different sizes.

Each message header specifies both the logical-message length and the buffer length. The logical-message length must be known to copy the entire message but avoid copying any extraneous data that follows the message in the buffer. The physical-buffer length is needed if the buffer is to be used again for another message — the buffer's size sets an upper limit on the size of the logical message that it can contain.

The *TaskResize* function is called each time a message buffer is to be loaded with a new logical message that might have to be copied. The function does two things. First, it determines whether a new logical message will fit into a particular message buffer. Second, it updates the header with the size of the new logical message.

### 2.3.7 Message Disposal

As mentioned earlier, the purpose of the *TaskReclaimMsg* function is to help a receiving task discard a message that it either no longer needs or is not prepared to deal with. The function attempts to send the message buffer to its reclamation port. If that fails, it deallocates the message.

The *TaskRelayMsg* function assists internode message managers in dealing with messages transmitted from foreign processor nodes. The function relays a message to the next leg of its journey toward its destination port. If an error prevents the message from reaching its destination, the function attempts to retrieve the buffer containing the message by sending it to its reclamation port. If that fails, it deallocates the message. The function makes no attempt to notify the sender of the message when a transmission error occurs. Such notification, if required for your application, must be provided by a higher-level protocol built upon the kernel's messaging primitives.

The *TaskReclaimMsg* and *TaskRelayMsg* functions are designed to quietly dispose of messages on behalf of the calling task. They neither post task errors nor return status values. This is in recognition of the likelihood that the task may be handing off a message it received from another source and knows little or nothing about. If the task were forced to assume responsibility for a corrupted message under these circumstances, it might be incapable of dealing with it. Furthermore, these functions give the receiving task a means of disposing of a message that the task itself might suspect is corrupted.

Internode message passing is only as reliable as the underlying hardware. The kernel can be viewed as providing an unreliable data packet service. If you require higher reliability, you may need to build higher-level protocols upon the kernel's internode message-passing mechanisms.

## 2.4 Semaphores

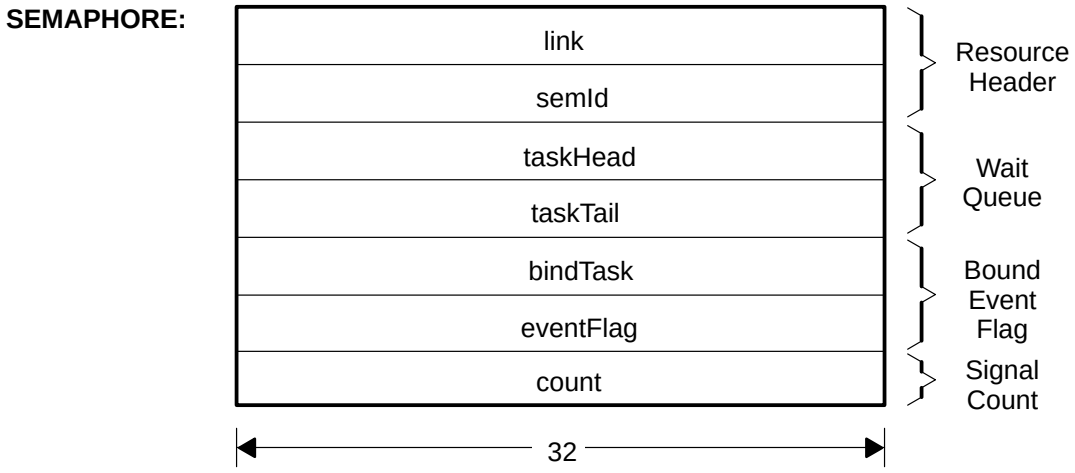
The kernel provides **signal** and **wait** operations on counting semaphores. These are useful for intertask synchronization and for coordinating the sharing of resources among tasks. Multiple tasks can signal the same semaphore, and multiple tasks can wait on the same semaphore.

A task calls the *TaskSignalSema* function to signal a semaphore. This increments the semaphore's signal count by 1.

A task calls *TaskWaitSema* to wait at a semaphore for a signal. If the semaphore's signal count is nonzero, the *TaskWaitSema* function decrements the signal count by 1 and returns immediately with the signal. Otherwise, the task waits until a signal arrives at the semaphore.

The structure of a semaphore is shown in Figure 2–7. The first six fields are identical in function to those of the port structure shown in Figure 2–6. The port's head and tail pointers to the message queue are replaced, however, with a *count* field that contains the current signal count.

Figure 2–7. Structure of a Semaphore



### 2.4.1 Signals

Similarly to the way that messages are queued at ports instead of at tasks, signals are sent to semaphores rather than directly to tasks. A signal is similar to a message with a zero-length message body: the only information conveyed by a signal is the fact that it has arrived. Other information, such as the source of the signal, may be inferred from the context, but this information is not directly available from the signal itself.

In contrast to a port, which must queue a list of unreceived messages to avoid losing the information they contain, the simple counter at the bottom of Figure 2–7 is sufficient to record the arrival of signals at a semaphore.

Signals can be viewed as low-cost alternatives to messages. A signal involves less processing overhead and does not require the allocation of a message buffer. It can be used in place of a message in instances in which the receiving task requires no information other than the signal itself.

In a multiprocessor system, messages can be sent from one processor node to another. Signals, however, can be exchanged only between tasks that execute on the same processor.

Signals are generally a more convenient means than messages for interrupt service routines (ISRs) to notify tasks of interrupts. While a message conveys more information than a signal, a signal has the advantage that it does not require the allocation of any resources. A request to allocate a message buffer can fail if insufficient storage is available or if another task is currently operating on the heap from which messages are allocated. To avoid the possibility of such failures, ISRs typically use signals rather than messages to notify tasks of interrupts.

### 2.4.2 Resource Management

Semaphores are useful in managing access to shared resources. Tasks running on the MP may need to share off-chip resources, such as peripheral devices and external memory. On-chip MVP resources that might need to be shared among tasks include the PPs (parallel processors) and the MP's (master processor's) floating-point accumulators and vector pointers.

Example 2–1 shows how the ANSI standard C functions *malloc* and *free* can be enhanced for use in a multitasking environment. A semaphore is used to ensure mutually exclusive access to the internal linked-list structures that manage the heap. If two or more tasks were allowed concurrent access to these structures, the heap might become corrupted.

Example 2–1. Multitasking Versions of Memory-Allocation Functions *malloc* and *free*

```
/*
 * Multitasking Versions of Memory Management Functions
 */
#include <stdlib.h>
#include <task.h>

long mallocSemaId;

void myMemInit(void)
{
   mallocSemaId = TaskOpenSema(–1, 1);
}

void *myMalloc(size_t size)
{
   void *ptr;
   TaskWaitSema(mallocSemaId);
   ptr = malloc(size);
   TaskSignalSema(mallocSemaId);
   return (ptr);
}

void myFree(void *ptr)
{
   TaskWaitSema(mallocSemaId);
   free(ptr);
   TaskSignalSema(mallocSemaId);
}
```

The *myMemInit* function in Example 2–1 initializes the multitasking memory management routines following system power-up. The function opens a semaphore, called *mallocSemaId*, that is used to manage heap access in the other two functions, *myMalloc* and *myFree*. It sets the semaphore's initial signal count to 1. The *myMalloc* and *myFree* functions wait on the semaphore to obtain the signal before accessing the heap structure. After the heap is updated, the respective function signals the semaphore to allow another task to access the heap. Mutual exclusion is ensured by the fact that only one task can possess the signal at a time.

The kernel function *TaskInstallMalloc* allows you to install your own versions of *malloc* and *free*. The kernel uses these routines to manage its storage requirements for message buffers, task descriptors, stacks, and so on. You may want to use the functions shown in Example 2–1 as models for these routines.

The *TaskInitTasking* function installs the *malloc* and *free* functions from the standard C library as defaults. The *myMemInit* function in Example 2–1 should be called only once — just after the call to *TaskInitTasking* and before any calls to *TaskResume*. Note that the call to *TaskOpenSema* allocates a semaphore structure by calling the *malloc* function that was installed by *TaskInitTasking*. This is perfectly safe as long as the call is made before any tasks (other than the default task) begin executing.

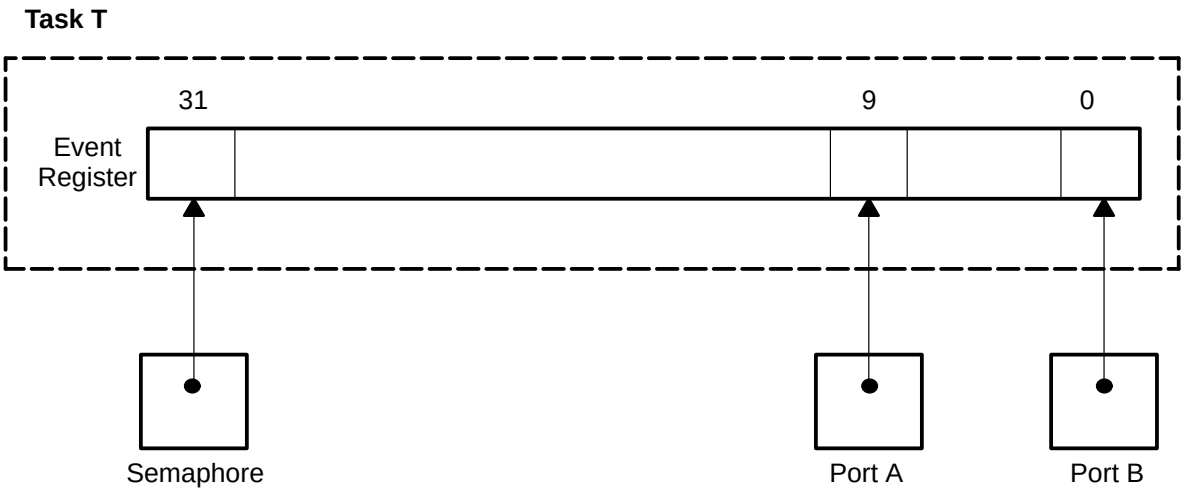
Note that the default task, which is described in subsection 2.8.3, cannot rely on semaphores to provide mutual exclusion. A call to the *TaskWaitSema* function from the default task always returns immediately regardless of the semaphore's signal count at the time of the call. For this reason, if the default task calls the allocation routines in Example 2–1, it risks corrupting the heap structure.

## 2.5 Monitoring Multiple Events

A task can simultaneously monitor any combination of up to 32 signal and message events. A signal event is the arrival of a signal at a semaphore. A message event is the arrival of a message at a port. When a task calls the *TaskWaitEvents* function to wait on a set of events, the first of these events to occur causes the task to be scheduled to execute again. Alternately, a task can call the *TaskPollEvents* function to poll its event flags.

As shown in Figure 2–8, each task descriptor contains a 32-bit event register. Each bit of this register is an event flag that can indicate the instantaneous status of a port or semaphore to which it is bound. In this example, a semaphore is bound to event flag 31, and ports A and B are bound to event flags 9 and 0, respectively. Event flag 31 is 1 any time the semaphore contains a nonzero signal count and is 0 at all other times. Event flags 9 and 0 are 1 if ports A and B, respectively, contain messages and are 0 otherwise. The event flags are updated instantaneously to reflect status changes in the semaphore and ports.

Figure 2–8. Two Ports and a Semaphore Bound to Event Flags Within Task T




The value of a free (or unbound) event flag is always 0.

A task binds a port or semaphore to one of its event flags by calling the *TaskBindPort* or *TaskBindSema* function, respectively. An individual event flag can be bound to, at most, either one port or one semaphore at a time. Similarly, a port or semaphore can be bound to only a single event flag.

Event flags provide information on the status of ports and semaphores, but have no effect on the operation of ports and semaphores. The binding between an event flag and a port or semaphore does not restrict other tasks' access to the port or semaphore.

In a typical application, a bound port or semaphore is accessed only by the task to whose event flag it is bound. While such a restriction may be adhered to voluntarily by the application, the kernel does not enforce it.

The value returned from a call to the *TaskWaitEvents* function is a snapshot of the event flags at the moment that one of the selected events was detected. The return value indicates the status of all 32 event flags, whether they are selected or not. The fact that a task was waiting on an event flag at the time that the flag was activated by a message or signal event does not preclude a second task from retrieving the message or signal before the first task can respond. Meanwhile, the event flags continue to accurately follow status changes in the ports and semaphores to which they are bound.

A task waiting on an event flag that is bound to a port receives no notification if a second task is actually waiting at the port at the time that a message arrives at the port. In other words, the event flag remains 0 during the transaction. This is because the kernel, in fact, delivers the message directly to the second task instead of to the port, which remains empty. Similarly, a task waiting on an event flag bound to a semaphore receives no notification if a second task is waiting at the semaphore at the time a signal arrives.

The meanings assigned to particular event flags, ports, and semaphores are application-dependent. For example, a task that manages a peripheral device may elect to assign a semaphore S and event flag 31 to interrupts from that device. A server task may elect to assign a port P and an event flag 0 to requests from clients. Such conventions, however, are established by applications for their own convenience and are not dictated by the kernel functions.

## 2.6 Exceptions and Task Errors

The kernel provides two means of notifying tasks of abnormal conditions:

- **Task Errors** that occur within a task during a call to a kernel function
- **Exceptions** to notify a task of exceptional conditions that may require special handling outside the normal program flow

### 2.6.1 Task Errors

A task error is an error detected by the kernel during a call to a kernel function. It is called a task error because it occurs within the context of the calling task. The calling task may or may not need to deal with the error before it can perform any further processing, but other tasks are not directly affected and can continue processing.

An example of a task error is an invalid port ID supplied as an argument to a kernel function.

When a task error occurs within a kernel function, the function returns a value that indicates that the function failed. For example, the *TaskAllocMsg* function returns a null pointer value instead of a valid message pointer. Before returning, the function also loads an error code into the calling task's descriptor. The calling task can interrogate the kernel by calling the *TaskGetLastError* to obtain the error code. If more than one task error has occurred since the task was created, only the code for the latest error is preserved. Appendix B, *Task Error Codes*, presents a list of task error codes.

Once a kernel function has returned after detecting a task error, the kernel takes no further action. The calling task is responsible for taking any corrective action that may be required.

## 2.6.2 Exceptions

An exception is an exceptional condition that occurs within the context of a particular task. The exception can be handled either by the task itself or by an exception handler associated with the task. Some typical exceptions are:

- Typing CONTROL C to abort a task in progress,
- A divide-by-zero error, and
- An illegal instruction.

The kernel's exception mechanism is similar in function to the UNIX *signal* facility.

Software exceptions are analogous in many respects to hardware interrupts. Similarly to the way an interrupt request can be signaled to a processor, an exception can be raised in a task. A task can mask selected exceptions similarly to the way a processor can mask interrupts. Also, just as a processor contains an interrupt-pending register, a task descriptor contains a 32-bit **exception register** in which each bit is an **exception flag** that reflects the status of a particular exception. Exception flags are numbered 0 to 31, where 0 is the LSB. The kernel sets an exception flag to 1 to indicate that the corresponding exception is pending.

The kernel provides the mechanisms needed to implement exception handling, but it does not define the meaning of any exception flag. Application programs that execute under the kernel may need to follow a common set of conventions for use of the 32 exception flags.

The *TaskRaiseExcep* function raises an exception in the specified task. This function can be called from the target task, from another task, or from an interrupt service routine. Because each flag represents a different exception source, several exceptions can be pending simultaneously within the same task. This means that if several exceptions occur before a task can respond to the first exception, none of the exceptions is lost.

The target task can detect the occurrence of an exception in one of two ways:

- The program can poll the exception register.
- An **exception handler** can be invoked automatically.

The drawback to polling is that the main task program must accept the overhead of polling its exception register frequently or else risk excessively delaying its response to critical occurrences. An exception handler avoids the overhead of polling.

When an exception is raised in the task, the kernel invokes the handler automatically, similarly to the way an interrupt traps to an ISR.

An exception handler is a function that can optionally be installed in a task to handle exceptions. Only a single exception handler can be installed in each task. In other words, the task's one handler must be prepared to deal with any exception that can occur within the task.

The exception handler operates as an extension to the task and executes in the task's environment. The handler always runs before the task processes past the point at which the exception occurred. If the calling task raises an exception in itself, the handler executes immediately, before the return from the call to *TaskRaiseExcep*. If another task is running when the exception is raised, the exception handler runs as soon as the target task begins executing again.

An exception handler runs at the priority of the task with which it is associated (see subsection 2.8.1). When an exception is raised within a task that is not currently executing, the scheduling of that task's next execution is not affected. In other words, the exception neither accelerates nor delays the next execution of the affected task.

Just as a processor can selectively mask interrupts, a task can call the *TaskSelectExcep* function to select the set of exceptions that invoke its exception handler. The *TaskInstallHandler* function specifies an initial mask at the time that the handler is installed in the task.

If necessary, an exception handler can call the *TaskSuspend* or *TaskExit* function on behalf of the task.

## 2.7 Private Contexts

A typical task is an application program that calls functions in several software libraries. Associated with each library may be some implementation-specific state information that the library maintains for each task in which it is used. This information represents the library's private context for the task. One way a library might keep track of all of its private contexts is to build a table or linked list that contains an entry for each task. However, this approach is inconvenient and inefficient because it requires that all libraries contain mostly redundant implementations of software to maintain their private contexts.

The kernel provides a more convenient set of facilities for managing private contexts. A private context can be automatically initialized in each task when the task is created. When the task exits, cleanup is performed. A library function always has direct access to the library's context within the current task, regardless of the level of function-call nesting.

### 2.7.1 Private-Data Words

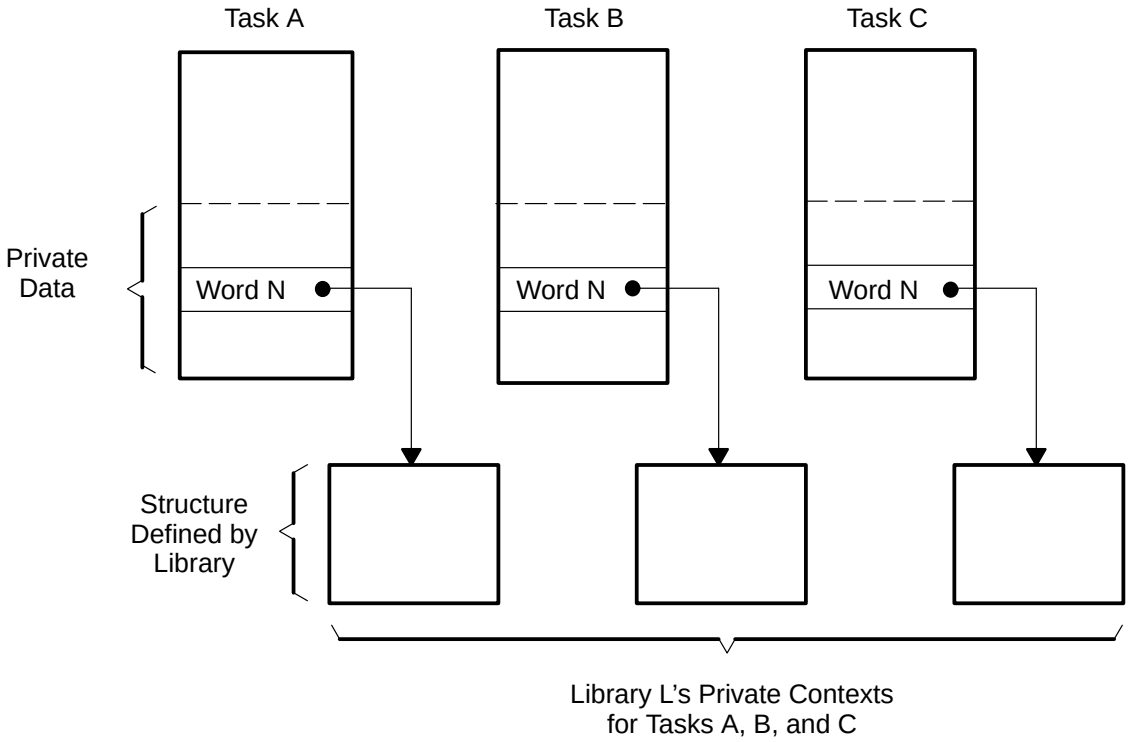
Every task descriptor contains an array of private-data words, each of which is used to maintain the private context of one of the libraries called by the task. If a single private-data word is not large enough to store all the private context data required by a library, the library's initialization routine can allocate a buffer of the necessary size and load a pointer to the buffer into the private-data word.

When a library allocates a private-data word, it actually reserves a word in the task-descriptor structure itself. Because an instance of this structure is associated with each task in the system, the number of words of physical storage allocated depends on the number of tasks in existence.

During system initialization, each library allocates its private contexts by calling the *TaskAllocPrivate* function, which returns a private-data index N. Note that this is not the index to a single private-data word in a single task. Rather, it is an index that identifies private-data word N in every task in the system. When a library routine subsequently refers to private-data word N, it accesses that particular word in the currently running task.

Figure 2–9 shows a library L's private contexts for tasks A, B, and C. L has previously allocated private-data word N. It now owns that word in each task in the system. Because L's private context is too large to fit into a single word, L has allocated storage for its private context within each task and has loaded a pointer to that structure into private-data word N within that task. Each time the task calls one of L's functions and that function requires access to L's private context, the function has immediate access to that context through the task's private-data word N.

Figure 2–9. Private Contexts




The *TaskGetPrivate* and *TaskSetPrivate* functions provide the ability to read from and write to, respectively, the private-data word identified by an index N.

### 2.7.2 Init-List and Exit-List

Each time a new task is created, the private contexts for that task must be allocated and initialized. When a task exits, any storage allocated to hold private-context information must be freed.

The kernel maintains a list of application-dependent functions that execute each time it creates a new task and another list of functions that execute each time a task exits. The two lists are called the init-list and exit-list, respectively. Immediately after the *TaskInitTasking* function initializes the kernel, both lists are empty. You can add your own functions to these lists by calling the *TaskAddFuncList* function.

The kernel executes all the functions in the init-list each time a new task is created and just prior to beginning execution of the task function itself. The init-list functions are executed within the context of the newly created task. They are executed in precisely the order in which the functions were added to the init-list by successive calls to the *TaskAddFuncList* function.

The kernel executes all the functions in the exit-list each time a task exits by either calling *TaskExit* or returning directly from the task function. The exit-list functions are executed within the context of the task that is about to exit. They are executed in an order that is precisely opposite to that in which the functions were added to the exit-list by successive calls to the *TaskAddFuncList* function.

No arguments are passed directly to the functions in the init- and exit-lists. These functions, however, can access the task's private contexts by calling *TaskGetPrivate*. The functions can access the task argument by calling *TaskGetTaskArg*.

### 2.7.3 An Example

Example 2–2 shows how the ANSI standard C functions *rand* and *srand* can be enhanced for use in a multitasking environment. A program in a single-threaded environment can specify an initial seed value to the *srand* function. Successive calls to *rand* then generate precisely the same sequence of pseudo-random numbers each time the program is run. Achieving the same repeatability in a multitasking environment, however, may require an implementation such as that shown Example 2–2.

The *myRand* and *mySRand* functions are multitasking versions of the *rand* and *srand* functions, respectively. The *myRand* function retrieves the calling task's current seed value and specifies this value to the *srand* function. This ensures that the new seed generated by the call to *rand* is based on the same task's previous seed, independent of calls to the *rand* and *srand* functions from other tasks. Inside *myRand*, the calls to *TaskWaitSema* and *TaskSignalSema* ensure that the calls to *srand* and *rand* are indivisible; without this protection, another task might alter the seed value between the calls to *srand* and *rand*.

The semaphore used inside *myRand* is identified by the variable *randSemaId*. The task's latest seed value is stored in the private-data word whose index is contained in the variable *randPrivate*. These variables are assigned values by the *setupRand* function, which is called only once following the call to *TaskInitTasking*. Thereafter, the global variables *randSemaId* and *randPrivate* remain constant and can be accessed by all tasks.

Each time it creates a new task, the kernel calls *initRand* to initialize the private-context information needed by the random-number routines. The *initRand* function is added to the kernel's init-list by the call to *TaskAddFuncList* inside the *setupRand* function. Note that this call does not add a corresponding *exitRand* function to the kernel's exit-list. An exit-list function might have been necessary to deallocate the buffer containing the private context for the random-number functions, except that no such buffer is needed. The only private-context data maintained by these functions is the seed number, which happens to fit in the private-data word itself.

Example 2–2. Multitasking Versions of Random-Number Functions *rand* and *srand*

```
/*
 * Multitasking Version of Random–Number Functions
 */
#include <stdlib.h>
#include <task.h>

#define INITIAL_SEED   12345

long randSemaId, randPrivate;

void initRand(void)
{
    TaskSetPrivate(randPrivate, INITIAL_SEED);
}

void setupRand(void)
{
    randSemaId = TaskOpenSema(–1, 1);
    randPrivate = TaskAllocPrivate();
    TaskAddFuncList(initRand, NULL);
}

int myRand(void)
{
    unsigned int seed;

    seed = (unsigned int)TaskGetPrivate(randPrivate);
    TaskWaitSema(randSemaId);
    srand(seed);
    seed = rand();
    TaskSignalSema(randSemaId);
    TaskSetPrivate(randPrivate, (void *)seed);
    return (seed);
}

void mySRand(unsigned int seed)
{
    TaskSetPrivate(randPrivate, (void *)seed);
}
```

## 2.8 Multitasking

The kernel provides the mechanisms required to perform multitasking. A task is a program element that is scheduled to execute concurrently with other tasks. Associated with each task is a program counter that advances as the processor progressively executes the program element. One or more tasks can be instances of the same reentrant program element. A task can access system resources such as the MVP's on-chip parallel processors that may be shared with other tasks.

The *TaskCreate* function creates a new task. The task is initially suspended and must be resumed by another task before it can run. This means that the *TaskCreate* function always returns before the new task begins executing. Any initialization of the new task, such as installing an exception handler, can be performed while the new task is still suspended.

When a task is created, the kernel allocates a task descriptor and a stack. The task descriptor contains information such as the task's current state, priority, event flags, exception flags, and the context data that is saved during a task switch. Each task has its own stack on which to push arguments, allocate local environments, and return values during function calls and interrupts.

The program code for a task is written in the form of a standard C function. The kernel begins executing the task by calling this function, which is referred to as the *task function*. During the call, a single pointer value is passed to the function as an argument. This is called the *task argument*. The meaning of this argument is application-dependent, but it typically points to a block of storage that holds a structure containing parameters needed by the task.

A task voluntarily terminates itself either by returning from the task function or by calling the *TaskExit* function.

### 2.8.1 Priority-Based Scheduling and Preemption

Task scheduling is based on priorities. A task is assigned a priority in the range 0 to 31, with larger numbers representing higher priorities. Level 0 is the lowest (least urgent) priority, and level 31 is the highest (most urgent).

A task is assigned an initial priority at the time that it is created. The task can dynamically change its priority by calling *TaskSetPriority*.

The execution of a task running at a priority less than 31 is automatically preempted when a task of higher priority that was previously blocked becomes ready to execute. This could happen, for example, if the second task is waiting at a port at the time that the first task sends a message to that port. It could also happen if an interrupt service routine signals a semaphore at which the second task has been waiting.

Multiple tasks can have the same priority. If more than one task of the same priority is ready to execute, these tasks are executed on a first-come-first-served basis. When a previously blocked task becomes ready to execute, it must wait in line for the processor behind any other ready tasks of the same priority.

A group of two or more tasks of the same priority can share the processor in round-robin fashion. The currently executing task can voluntarily yield the processor to the next task in the group by calling the *TaskYield* function. Also, a form of time-slicing can be implemented by generating periodic timer interrupts that invoke the *TaskYield* function.

Tasks waiting at a port or a semaphore are served on a first-come-first-served basis regardless of their relative priorities.

### 2.8.2 Blocking Calls

A task that is not ready to execute is said to be blocked. A task can be blocked either by being suspended or by voluntarily choosing to wait for an event such as the arrival of a message or signal.

A suspended task is blocked from further execution until another task resumes the suspended task. As mentioned above, a newly created task is initially suspended. The *TaskSuspend* function also suspends a task.

The kernel includes three additional blocking functions: *TaskReceiveMsg, TaskWaitSema,* and *TaskWaitEvents.*

- The *TaskReceiveMsg* function receives a message from a port but waits at the port if it is empty.
- The *TaskWaitSema* function waits at a semaphore for a signal.
- The *TaskWaitEvents* function waits for the first of a set of selected events to occur.

Nonblocking versions of these three functions are also available: *TaskAcceptMsg*, *TaskCheckSema*, and *TaskPollEvents*. A task uses these functions to avoid having to wait.

- The *TaskAcceptMsg* function accepts a message from a port if a message is available but does not wait if the port is empty.
- The *TaskCheckSema* function accepts a signal from a semaphore if the semaphore's signal count is nonzero but does not wait if the count is 0.
- The *TaskPollEvents* function returns immediately with a snapshot of the calling task's event flags.

### 2.8.3 Default Task

At least one task is always executing. The *default task* executes any time no other task is ready to run. The kernel never allows this task to wait for a signal event or a message event. The default task cannot be suspended, and it cannot exit. The kernel defeats any attempt to suspend, terminate, or otherwise block the default task.

The kernel creates the default task during the call to *TaskInitTasking* that initializes the kernel. Following system power up, this function must be called once and only once before any other kernel functions can be called.

Following system reset, the reset routine begins executing the compiler's standard boot routine, which sets up the default C environment and then calls *main*. At this point, only the single-threaded program defined by the C compiler is executing. Multitasking begins only when the initial program calls the kernel's *TaskInitTasking* function. Upon return from the *TaskInitTasking* function, the kernel has transformed the initial program into the default task. The default task continues to use the stack set up by the C boot routine. The kernel assigns an initial priority of 0 to the default task.

The default task can create other tasks, send messages, and signal semaphores. Its event flags can be bound to ports and semaphores. The default task's primary functions are:

- Initialize the system environment following power-up
- Perform background processing any time no other task is ready to execute

A number of kernel functions exhibit special behavior when called from the default task:

- The *TaskGetTask* function returns a value of –1 in place of a valid task ID.
- The *TaskExit* function ignores the task's request to exit and returns immediately.
- The *TaskWaitEvents* function never waits; it returns immediately with a snapshot of the current event flags.
- If *TaskReceiveMsg* or *TaskWaitSema* is called when the message queue is empty or the signal count is 0, respectively, the function posts a task error and returns immediately with a null pointer or a value of 0.

While the default task is executing at a particular priority level, another ready task with the same priority is not allowed to run until the default task calls *TaskYield* to explicitly yield the processor to the other task. A task with a priority lower than that of the default task never executes. For these reasons, a typical system maintains the priority of the default task at 0 and assigns higher priorities to all other tasks.

### 2.8.4 Ready Queue

All tasks that are ready to execute are contained in a priority-sorted list called the **ready queue**. Any task not in the ready queue is, by definition, blocked. Conceptually, the ready queue is a linked list of task descriptors that are sorted in order of decreasing priorities, with the higher-priority tasks toward the head of the list.

The ready queue is, in fact, implemented as an array of 32 linked lists — one list per priority level. This structure makes insertion operations more efficient. For simplicity, however, assume that the ready queue is a logically equivalent structure: a single linked list containing all ready tasks of all priorities.

At the head of the ready queue is the task that is currently executing. When a task calls a kernel function, the kernel knows that the calling task must reside at the head of the ready queue.

When a new task that was previously blocked is added to the ready queue, it is inserted into the queue in front of all tasks of lower priority and behind all tasks of equal or higher priority. When the ready queue contains two or more tasks of the same priority, the tasks are granted access to the processor on a first-come-first-served basis.

When the currently executing task yields the processor, the kernel removes the task from the head of the ready queue and inserts it back into the queue behind any other tasks of the same priority. The task that moves to the head of the queue begins to execute next. If no other task in the ready queue has the same priority as the yielding task, that task remains at the head of the ready queue and continues executing.

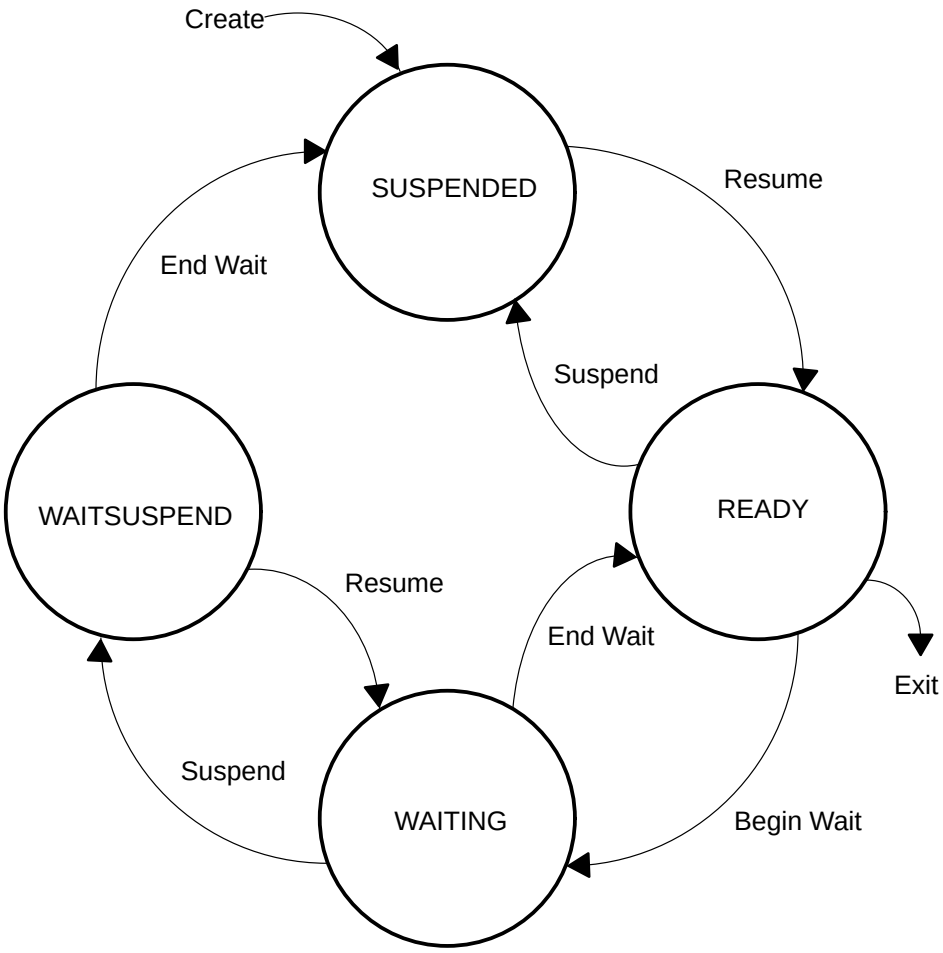
A task can change its priority dynamically by calling *TaskSetPriority*. When a task calls this function to change its priority, the kernel removes the task from the head of the ready queue and inserts it back into the ready queue ahead of any tasks of the same or lower priority. This insertion scheme allows the task to continue executing whenever possible. The task is preempted only if it lowers its priority below that of another task in the ready queue.

The ready queue is never empty. It always contains the default task.

## 2.8.5 Task States

As shown in Figure 2–10 a task is always in one of four states: READY, WAITING, WAITSUSPEND, or SUSPEND.

Figure 2–10. Kernel State Transition Diagram




A task in the READY state is ready to execute and is in the ready queue. The currently executing task is always the task at the head of the ready queue. No separate state distinguishes the currently executing task from the other tasks in the ready queue.

A task in the WAITING state is waiting for a message event or a signal event to occur. A message event is the arrival of a message at a port. A signal event is the arrival of a signal at a semaphore.

The kernel contains three functions that can cause a task to wait: *TaskReceiveMsg*, *TaskWaitSema*, and *TaskWaitEvents*. The *TaskReceiveMsg* function causes the task to wait at a port until a message arrives. The *TaskWaitSema* function causes a task to wait at a semaphore until a signal arrives. The *TaskWaitEvents* function allows a task to simultaneously monitor multiple message and signal events; the first of these events to occur causes the task to become ready to execute again.

A task in the SUSPENDED state is suspended from execution until another task causes it to resume. A task can suspend itself or another task by calling the *TaskSuspend* function. Once a task is suspended, it remains suspended indefinitely or until another task calls *TaskResume* on behalf of the suspended task.

When a task is created, its initial state is SUSPENDED. It will not begin execution until another task causes it to resume.

If the target task is in the WAITING state at the time another task attempts to suspend it, the state of the target task changes to WAITSUSPEND rather than to SUSPENDED. This state indicates that suspension is pending and will occur as soon as the event for which the task is waiting occurs. When the awaited event occurs, the task's state changes to SUSPENDED rather than to READY. If another task calls *TaskResume* to resume a task that is in the WAITSUSPEND state, the target task's state changes back to WAITING.

The behavior of a task waiting for a message event or a signal event is the same regardless of whether the task is in the WAITING or the WAITSUSPEND state. The only difference in behavior occurs when the awaited event occurs: a task in the WAITING state changes to the READY state, but a task in the WAITSUSPEND state changes to the SUSPENDED state. A task contained in a wait queue at a port or semaphore can be in either the WAITING or the WAITSUSPEND state. Ports and semaphores treat all waiting tasks identically, regardless of which of these two states they are in.

A task always exits from the READY state. No task can terminate another task. A task terminates itself by calling the *TaskExit* function or by returning from the task function.

## 2.8.6 Scheduling Examples

Two examples of task scheduling are shown in Figure 2–11 and Figure 2–12. Each box in the figures represents a task descriptor, and the task's priority is shown within the box. Assume that the default task has a priority of 0 and permanently resides at the tail of the ready queue.

Figure 2–11 shows an example of a task that waits for an event to occur. In (a), Task A, which sits at the head of the ready queue, is the currently executing task. Task A calls the *TaskWaitEvents* function to wait for an event. In (b), the *TaskWaitEvents* function has removed Task A from the ready queue to await the event. This allows Task B to move to the head of the ready queue and begin executing. Because Task A is waiting on its own event flags, it does not enter a wait queue as it would if it were waiting at a port or a semaphore. When the event that Task A is waiting for occurs, it becomes ready to execute again. In (c), the kernel inserts Task A back into the ready queue ahead of all tasks of lower priority, but behind all tasks of greater or equal priority. This places Task A in line behind Task B, which delays Task A's further execution until Task B either yields or is blocked.

Figure 2–11. A Task Waits for an Event

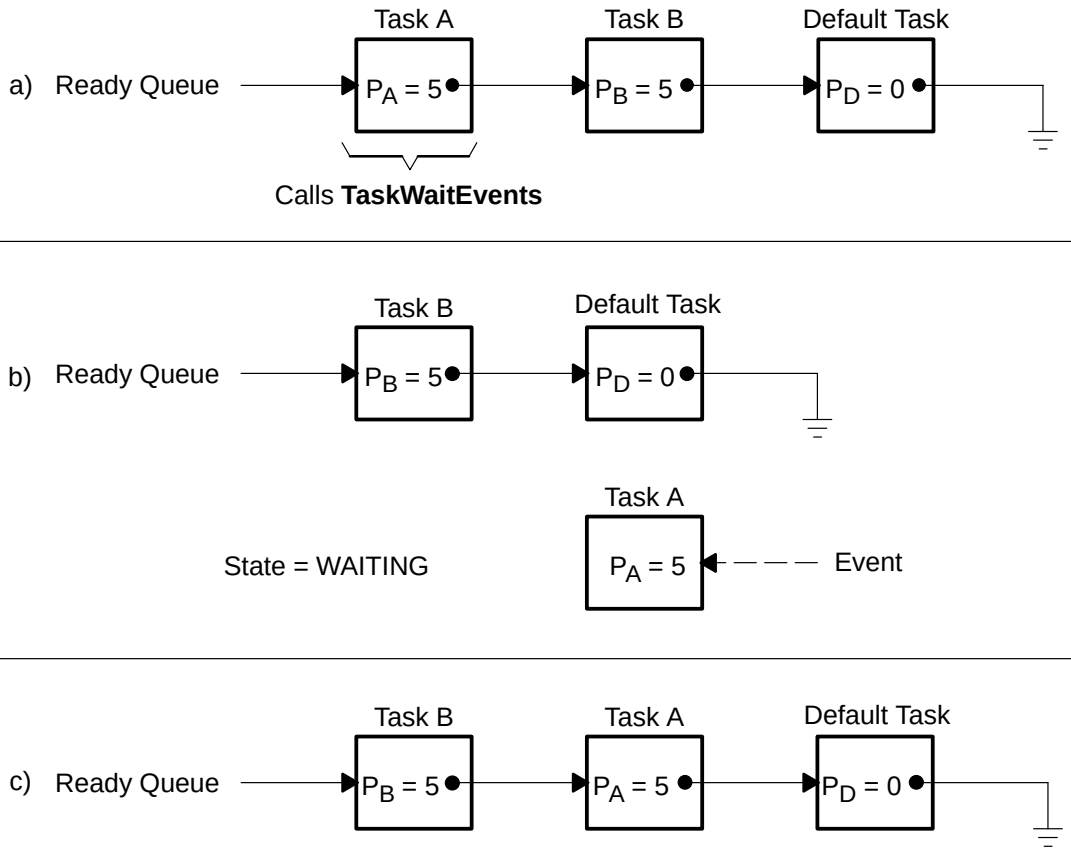
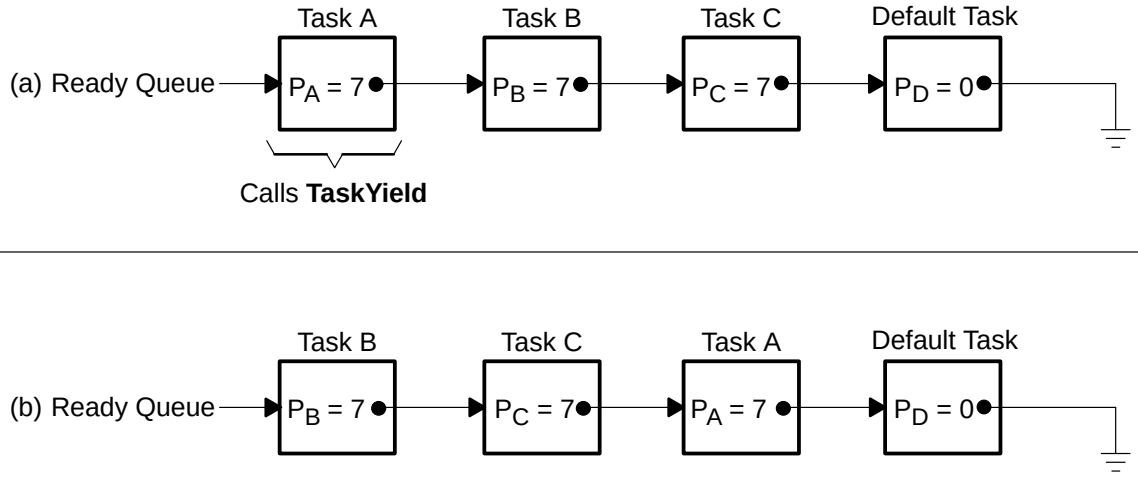


Figure 2–12 shows an example of a task that voluntarily yields the processor to another task of equal priority. In (a), Task A, which sits at the head of the ready queue, is the currently executing task. Task A calls the *TaskYield* function to yield the processor to another task of the same priority if one is ready. In this example, Tasks B and C, both of which have priorities equal to that of Task A, are ready to execute. In (b), the *TaskYield* function has removed Task A from the head of the queue and inserted it back into the queue behind Tasks B and C. Task B is now at the head of the ready queue and immediately begins executing. The execution of Task A continues only after Tasks B and C have executed.

Figure 2–12. A Task Yields to Another of Equal Priority



### 2.8.7 Task Exit

A task can terminate itself either by calling *TaskExit* or by returning directly from the task function that was called when the task was created. Either method has the same effect.

When a task exits, its task descriptor and stack are both deallocated, and its task ID is deleted from the kernel's internal resource tables.

Before exiting, a task should release any shared resources it holds. For example, it should free any dynamically allocated memory it has obtained. If the task has received any messages that have not yet been sent, reclaimed, or deallocated, those messages are lost when the task exits. Similarly, if the task has waited on a semaphore but has not signaled the semaphore, the signal is lost.

One task cannot directly terminate another task. However, you can write a task's exception handler to call *TaskExit* on behalf of the task in response to a particular exception. Raising that exception causes the target task to exit.

When a task terminates in this fashion, it may not have had the chance to release any system resources that it holds. When this occurs, those resources may be unavailable until the next time the system is powered up.

### 2.8.8 Task Control Flags

Each task descriptor contains several **task control flags** that are used in conjunction with exceptions and with freeing storage space belonging to tasks that have exited.

Two of the flags, DOEXCEP and DOEXIT, are checked by the kernel after a task switch. These flags indicate whether the kernel must perform any special operations within the newly restored context before the task is allowed to continue its normal processing.

The DOEXCEP flag tells the kernel that the newly restored task must handle an exception. When, for example, Task A raises an exception in Task B by calling the *TaskSetExcep* function, that function sets Task B's DOEXCEP flag but does not cause Task A to relinquish the processor to Task B. The handling of the exception is deferred until Task B is scheduled to run again.

The DOEXIT flag tells the kernel that it must free the storage associated with the preceding task, which has just exited. When, for example, Task A exits by calling the *TaskExit* function, the function is unable to deallocate the storage for Task A's stack and task descriptor while it is still executing within the context of that task. Before scheduling the next task (Task B) to run, the function sets Task B's DOEXIT flag. This is the kernel's way of reminding itself to free Task A's storage as soon as the switch to Task B's context has been made.

Two additional flags, INEXCEP and INEXIT, indicate when a task is in the process of handling an exception or deallocating an exiting task's storage, respectively. Because the code for freeing a task or handling an exception must not be reentered a second time by the same task the INEXCEP and INEXIT flags are needed to protect these critical sections of code.

The INEXCEP flag is necessary because the kernel enables interrupts before calling an exception handler. A task can be interrupted and preempted while executing its exception handler. The kernel must ensure that when the preempted task resumes execution it does not begin executing the exception handler a second time, which would probably overwrite and corrupt the internal data of the exception handler. The INEXCEP flag prevents this type of failure from occurring.

Similarly, the INEXIT flag is necessary because the kernel enables interrupts before deallocating an exiting task's stack and task descriptor. This deallocation occurs within the context of a task other than the one that is exiting. The kernel must properly deal with a task that is interrupted and preempted while performing this deallocation. Otherwise, when the preempted task is later restored, the kernel will make the error of calling the same deallocation function a second time. Assuming that the function uses a semaphore to provide mutually exclusive access to the heap, the task will then be deadlocked, waiting in vain for its previous invocation to finish the heap operation and signal the semaphore. The INEXIT flag ensures that this type of failure does not occur.

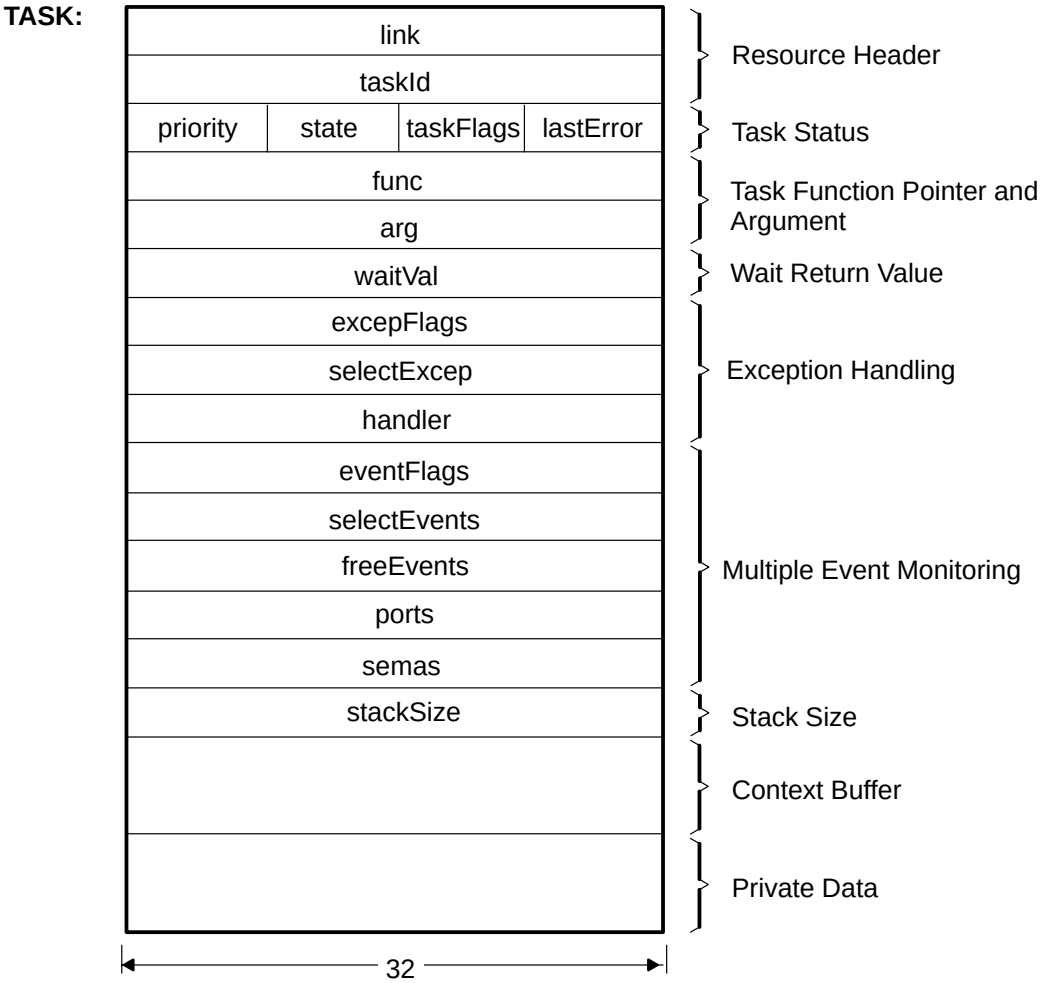
### 2.8.9 Task Descriptor

Each task is described by a structure called a **task descriptor**. Specified in the task descriptor is the task's state, priority, function pointer, argument, event flags, and exception status. It contains a buffer in which the task's internal registers are saved during a context switch. The task descriptor also contains an array of private-data words that function libraries use to maintain private contexts within the task.

The *TaskCreate* function creates a task by allocating a task descriptor and a stack. The function initializes the task descriptor, loads a pointer to the task descriptor into the kernel's internal resource tables, and assigns an ID to the task.

Figure 2–13 shows the structure of a task descriptor.

Figure 2–13. Task Descriptor Structure




- The 32-bit *link* field contains a pointer to the next task in a linked list of tasks. A task can be a part of the ready queue or part of the wait queue at a port or semaphore.
- The 32-bit *taskId* field contains the task ID assigned to the task at its creation.
- The 8-bit *priority* field indicates the task's priority. A task is assigned a priority level in the range 0 to 31, with larger numbers representing higher priorities.
- The 8-bit *state* field indicates the state of the current task. The four possible states of a task are READY, WAITING, SUSPENDED, and WAITSUSPEND.
- The 8-bit *taskFlags* field contains the task control flags. These flags facilitate exception handling and the freeing of storage associated with tasks that have exited.

- The 8-bit *lastError* field contains the code for the last task error that occurred within the task. A task error is an error that has occurred within a kernel function that was called from the task. This field is initialized to 0 at task creation.

- The 32-bit *func* field contains a pointer to the task function. This is the task's program, which is written in the form of a standard C function that accepts a single argument. A newly created task begins its execution at the entry-point address contained in this field.

- The 32-bit *arg* field contains the pointer value that is passed as an argument to the task function when the task first begins execution after being created. The meaning of this argument is application-dependent, but it is typically a pointer to a buffer that contains parameters required by the task.

- The 32-bit *waitVal* field contains the value returned by a kernel function call that requires the task to wait. These functions are *TaskReceiveMsg*, *TaskWaitEvents*, and *TaskWaitSema*.

- The 32-bit *excepFlags* field is the task's exception register. Each bit is a flag that is raised to indicate that a particular exception has occurred. Flags are numbered 0 through 31 (0 is the LSB). This field is initialized to 0 at task creation.

- The 32-bit *selectExcep* field is a mask that indicates which of the 32 possible exceptions the task's exception handler is prepared to deal with. Only a selected exception, represented by a bit value of 1, causes execution of the exception handler. This field is initialized to 0 at task creation.

- The 32-bit *handler* field is a pointer to the exception handler, which is written as a C function that accepts no arguments. A null pointer value indicates that no exception handler is installed. This field is initialized to a null pointer at task creation.

- The 32-bit *eventFlags* field is the task's event register. Each bit is a flag that can represent the status of a port or semaphore to which it is bound. A flag bound to a port has the value 1 as long as the port contains messages. A flag bound to a semaphore has the value 1 as long as the signal count is nonzero. Event flags are numbered 0 through 31 (0 is the LSB). This field is initialized to 0 at task creation.

- The 32-bit *selectEvents* field is a mask that indicates the set of events the task has selected to wait on. Only a selected event, represented by a bit value of 1, will cause the task to become ready to execute. This field is initialized to 0 at task creation. It is nonzero only while the task is actually waiting for a selected event to occur.
- The 32-bit *freeEvents* field is a mask that indicates the set of event flags that are free; that is, not bound to ports or semaphores. A free flag is represented by a bit value of 1. This field is initialized to all 1s at task creation.
- The 32-bit *ports* field is a pointer to a linked list of the ports that are bound to the task's event flags. This field is initialized to a null pointer at task creation.
- The 32-bit *semas* field is a pointer to a linked list of the semaphores that are bound to the task's event flags. This field is initialized to a null pointer at task creation.
- The 32-bit *stackSize* field specifies the size of the task's stack area in bytes.
- The context buffer is the storage area in which the task's context is saved while another task executes. The length of this buffer is system-dependent. In the current implementation, the buffer is defined to be of type *jmp_buf*, which is defined in the ANSI standard C header file, *setjmp.h*.
- The private-data area is an array of private-data words. A function library can allocate one of these words to hold a pointer to a buffer that contains the library's private context within the task. This allows a library to maintain private data separately from the contexts of the main task and of the other libraries used by the task. Each private-data word is initialized to a null pointer at task creation.

The structure in Figure 2–13 is shown according to big-endian conventions. The *priority* field precedes the *state* field, and so on.

## 2.9 Interrupt Handling

All interrupt and device handling is performed outside the kernel. To communicate with a task running under the kernel, an interrupt service routine (ISR) responding to a real-world event, such as a keyboard input or timer interrupt, must translate the event into a form that the kernel understands. The kernel understands only message events and signal events.

An ISR typically communicates with a task by signaling the task through a semaphore, rather than by sending it a message. As mentioned previously, a signal has the advantage that it requires no system resources, whereas a message requires the allocation of a buffer to contain the message. Memory allocation presents difficulties for ISRs. In a multitasking system, a task may have to wait on a semaphore to access a shared-heap structure, but an ISR must not wait. Also, an allocation request fails if insufficient storage is available. Using signals avoids these problems.

When an interrupt occurs, the local environment for the interrupt service routine is allocated from the top of the stack belonging to the task that was executing when the interrupt occurred. The ISR executes within the context of the interrupted task. For example, a call to *TaskGetTask* from an ISR returns the task ID of the interrupted task. A far as the kernel is concerned, the ISR is simply a continuation of the interrupted task and has the same priority as that task. An ISR should not disturb the environment of the interrupted task.

While an ISR executes with interrupts disabled, other interrupts may be delayed. ISRs should be as brief as possible. A typical ISR notifies one or more tasks when an external event occurs but leaves it to the tasks themselves to perform any lengthy calculations that may be required to handle the event.

### 2.9.1 Preemption During Interrupts

An ISR can cause preemption by calling a kernel function such as *TaskSignalSema* that can cause a blocked task to become ready to execute. The new task immediately preempts the interrupted task if its priority is higher. When task preemption occurs within an ISR, the state of the interrupted task is preserved on its stack. Meanwhile, the kernel function transfers control to the preempting task, which has its own stack. Only when control is eventually returned to the interrupted task does the return-from-interrupt occur. This restores the state of the interrupted task so that it can continue processing from the point of the interrupt.

The processor's global interrupt-enable flag is part of the task's state that is saved when a task is preempted. This flag is restored when the task begins executing again. The *TaskCreate* function sets a new task's global interrupt enable flag to the *enabled* state. During normal operation, a task should run with interrupts enabled.

When an interrupt occurs, however, the processor automatically disables interrupts before vectoring to the entry point of the ISR. The processor accomplishes this by clearing its global interrupt-enable flag. The ISR has the option of enabling interrupts again. To do so can be risky, though, unless the stack belonging to the interrupted task is large enough to allow the worst-case number of interrupts to pile up without overflowing. A more conservative approach is to keep interrupts disabled until the ISR returns control to the interrupted task.

In fact, a task's stack may need to be large enough to accommodate all of the following:

- The task function and any functions it calls
- The exception handler
- Two ISRs

When a task switch restores the context of a task, the kernel checks the task's control flags to determine if the task needs to: (1) handle an exception, or (2) deallocate the stack and task descriptor of another task that has just exited. Before performing either of these actions, the kernel always enables interrupts. Only when these actions are completed does the kernel restore the task's original interrupt-enable status (for example, to *disabled* if the task was executing an ISR). In the meantime, a second interrupt may pile up on the task's stack. This means that a total of two interrupts, rather than one, can pile up on the stack, even if ISRs always keep interrupts disabled. The second interrupt can be ruled out only if: (1) the task does not have an interrupt handler and (2) none of the tasks in the system ever exits.

An ISR can call most of the kernel functions while interrupts are disabled. If a task switch occurs as a result of one of these calls, the kernel remembers to disable interrupts again when a task switch later restores the ISR's context.

An ISR can call the *TaskRaiseExcep* function to raise an exception in a task. This call never causes preemption.

An ISR should avoid calling kernel functions that might cause it to wait. These functions are *TaskReceiveMsg*, *TaskWaitEvents*, and *TaskWaitSema*.

An ISR should also avoid calling kernel functions that allocate storage. These functions are *TaskAllocMsg*, *TaskCreate*, *TaskOpenPort*, and *TaskOpenSema*.

An ISR can raise its priority level to inhibit premature task preemption when the ISR makes kernel calls that may activate multiple tasks. For instance, consider an ISR that interrupts a priority-10 task to signal two tasks of priority 15 and 20, respectively. Assume that without raising its priority from that of the interrupted task, the ISR signals the priority-15 task first. The priority-15 task immediately preempts the priority-10 ISR and unnecessarily delays execution of the priority-20 task until later. A better approach is for the ISR to temporarily raise its priority, signal all the tasks, and then restore its original priority to allow preemption by the signaled task with the highest priority.

If an ISR calls a kernel function that causes preemption, that function may, in rare instances, never return control to the ISR. Accordingly, if the ISR performs any actions essential to the correct operation of the system, it should complete those actions before permitting itself to be preempted.

The one instance in which control never returns to the ISR occurs if the interrupted task, on whose stack the ISR executes, is terminated by its own exception handler. This could happen if a fatal exception is raised in the interrupted task during the preemption. Note that once the ISR has allowed itself to be preempted, the kernel resumes execution of the ISR only after the exception handler associated with the interrupted task has had an opportunity to execute. If the handler, however, terminates the interrupted task by calling the *TaskExit* function, the ISR never returns from the interrupt to restore the state of the interrupted task. This hardly matters, however, since the task itself no longer exists and the kernel has deallocated its task descriptor and stack. A problem occurs only if the failure to resume execution of the ISR prevents the ISR from completing pending actions of importance to other tasks. This potential problem is easily avoided by placing any calls to preempting kernel functions at the end of the ISR, after the ISR has completed any other actions needed to keep the system running correctly.

### 2.9.2 Round-Robin Scheduling and Time Slicing

The *TaskYield* function supports round-robin scheduling and time slicing.

As illustrated in Figure 2–12, the *TaskYield* function can be called by a task that is willing to surrender the processor to another task of the same priority if one is ready to run.

The *TaskYield* function can also be called from an ISR to time-slice the processor among a group of tasks of the same priority. After each task in the group has held the processor for its prescribed time slice, a periodic interrupt reschedules the tasks in the group to give the next task in the group a chance to execute. The ISR calls the *TaskYield* function to rotate all the ready tasks of a specified priority in round-robin fashion. The operation is similar to that shown in Figure 2–12, except that the currently executing task is forced to yield by the ISR, rather than yielding on its own.

Chapter 3

# Kernel Functions

This chapter describes the individual kernel functions in detail. The functions are listed in alphabetical order. Each listing specifies the syntax of a function call and a detailed description of the function's arguments, behavior, and return values. The chapter is organized into the following sections:

**Topics**

# 3.1 Summary of Kernel Functions

Table 3–1 contains an alphabetical list of the 52 kernel functions and a brief description of the action performed by each function.

Table 3–1. Alphabetical List of Kernel Functions

| Function Name | Description | See Page |
|---|---|---|
| TaskAcceptMsg | Accept a Message Without Waiting | EX:3-5 |
| TaskAddFuncList | Add to the Init- and Exit-Lists | EX:3-6 |
| TaskAllocMsg | Allocate a Message | EX:3-7 |
| TaskAllocPrivate | Allocate a Word of Private Data | EX:3-8 |
| TaskBindPort | Bind a Port to an Event Flag | EX:3-9 |
| TaskBindSema | Bind a Semaphore to an Event Flag | EX:3-10 |
| TaskCheckSema | Check a Semaphore Without Waiting | EX:3-11 |
| TaskClearExcep | Clear Exceptions | EX:3-12 |
| TaskClosePort | Close a Port | EX:3-13 |
| TaskCloseSema | Close a Semaphore | EX:3-14 |
| TaskCreate | Create a Task | EX:3-15 |
| TaskExit | Exit the Calling Task | EX:3-16 |
| TaskFreeMsg | Free a Message Buffer | EX:3-17 |
| TaskGetExcep | Get the Pending Exceptions | EX:3-18 |
| TaskGetFreeEvents | Get the Free Event Flags | EX:3-19 |
| TaskGetLastError | Get the Last Error Code | EX:3-20 |
| TaskGetMsgSize | Get the Size of a Message | EX:3-21 |
| TaskGetPriority | Get the Calling Task's Priority | EX:3-22 |
| TaskGetPrivate | Get a Private-Data Word | EX:3-23 |
| TaskGetReplyPort | Get a Message's Reply Port | EX:3-24 |
| TaskGetTask | Get the Task ID | EX:3-25 |
| TaskGetTaskArg | Get the Task Argument | EX:3-26 |
| TaskInitMsg | Initialize a Message | EX:3-27 |
| TaskInitTasking | Initialize the Kernel | EX:3-28 |
| TaskInstallHandler | Install an Exception Handler | EX:3-29 |
| TaskInstallMalloc | Install the Memory Allocation Functions | EX:3-30 |
| TaskOpenPort | Open a Port | EX:3-31 |
| TaskOpenSema | Open a Semaphore | EX:3-32 |
| TaskPollEvents | Poll the Event Flags | EX:3-33 |
| TaskRaiseExcep | Raise an Exception | EX:3-34 |
| TaskReceiveMsg | Receive a Message | EX:3-35 |

Table 3–1. Alphabetical List of Kernel Functions (Continued)

| Function Name | Description | See Page |
|---|---|---|
| TaskReclaimMsg | Reclaim a Message | EX:3-36 |
| TaskRelayMsg | Relay a Message | EX:3-37 |
| TaskResetSema | Reset a Semaphore | EX:3-38 |
| TaskResizeMsg | Resize a Message | EX:3-39 |
| TaskResume | Resume a Suspended Task | EX:3-40 |
| TaskRouteMsg | Route a Message | EX:3-41 |
| TaskSelectExcep | Select Exceptions | EX:3-42 |
| TaskSendMsg | Send a Message | EX:3-43 |
| TaskSetMsgRoute | Set a Message-Routing Port | EX:3-44 |
| TaskSetPriority | Set the Calling Task's Priority | EX:3-45 |
| TaskSetPrivate | Set a Private-Data Word to a Value | EX:3-46 |
| TaskSetReplyPort | Set a Message's Reply Port | EX:3-47 |
| TaskSignalSema | Signal a Semaphore | EX:3-48 |
| TaskSuspend | Suspend a Task | EX:3-49 |
| TaskUnbindPort | Unbind a Port From an Event Flag | EX:3-50 |
| TaskUnbindSema | Unbind a Semaphore From an Event Flag | EX:3-51 |
| TaskValidatePort | Validate a Port ID | EX:3-52 |
| TaskValidateSema | Validate a Semaphore ID | EX:3-53 |
| TaskWaitEvents | Wait for a Selected Event | EX:3-54 |
| TaskWaitSema | Wait at a Semaphore for a Signal | EX:3-55 |
| TaskYield | Yield to Another Task | EX:3-56 |

## 3.2 Alphabetical Reference

This section contains detailed descriptions of each of the 52 user-callable functions in the kernel. Arguments, return values, and syntax are specified for each function. The functions are listed in alphabetical order.

**Syntax**

void *TaskAcceptMsg(long *portId*);

**Description**

The *TaskAcceptMsg* function returns a message from a local port if the port contains a message but does not wait if the port is empty. This is a nonblocking version of the *TaskReceiveMsg* function described on page EX:3-35.

Argument *portId* is a port ID that identifies the local port from which the calling task accepts a message. It can be expressed in either node-local or node-global format. The port must have been previously opened.

If the port contains a message, the function removes the message from the port and returns a pointer to the message body. If the port contains no message, the function returns a null pointer. In either case, the function returns immediately without waiting.

For the sake of speed, the function does not verify that argument *portId* is a valid local port ID. The results are undefined if the argument is not valid.

**Syntax**

```
void TaskAddFuncList(void (* initFunc)(void),
                     void (* exitFunc)(void));
```

**Description**

The *TaskAddFuncList* function adds functions to the init-list and exit-list that are maintained by the kernel. The functions in these two lists are executed at task creation and task exit, respectively.

The purpose of this function is to allow software libraries and groups of related functions to initialize their private contexts within each task as it is created and to do cleanup when a task exits.

Argument *initFunc* is a pointer to the function that is to be added to the init-list, and argument *exitFunc* is a pointer to the function that is to be added to the exit-list. Each of these functions accepts no arguments and returns no value. If you want to add a function to only one of the two lists, specify a null value for the other argument to indicate that no function is to be added to the corresponding list.

The kernel executes all the functions in the init-list each time a new task is created and just prior to beginning execution of the task function itself. The init-list functions are executed within the context of the newly created task. They are executed in precisely the order in which the functions were added to the init-list by successive calls to the *TaskAddFuncList* function.

The kernel executes all the functions in the exit-list each time a task exits by either calling *TaskExit* or returning directly from the task function. The exit-list functions are executed within the context of the task that is about to exit. They are executed in an order that is precisely opposite to that in which the functions were added to the exit-list by successive calls to the *TaskAddFuncList* function.

**Syntax**

void *TaskAllocMsg(long *size*, long *portId*);

**Description**

The *TaskAllocMsg* function allocates a message of the specified size and returns a pointer to the message body. The function initializes the message header but leaves the message body uninitialized.

Argument *size* is the specified size (in bytes) of the message body. The function automatically allocates the fixed amount of additional storage required for the message header. The minimum size for a message is zero bytes; this is a message that consists of a message header only.

Argument *portId* is a port ID that identifies the message's reclamation port. This is the port to which the message buffer is to be sent when the message itself is discarded. The purpose of the reclamation port is to allow the storage for the message to be reclaimed to be used to send another message. The port must be local (that is, not located in a foreign processor node), and its ID can be expressed in either node-local or node-global format. The port must have been previously opened.

You may specify a value of –1 for argument *portId* in place of a valid port ID. This indicates that no reclamation port is to be assigned to the message. When the message is later discarded by a call to the *TaskReclaimMsg* function, that function simply deallocates the message buffer.

If any of the following errors occur, the function posts a task error and returns a null pointer value:

- The function is unable to allocate storage for a message of the specified size.
- Argument *portId* is neither a valid port ID nor –1.
- The requested message body size exceeds 32 768 bytes.

**Syntax**

long TaskAllocPrivate(void);

**Description**

The *TaskAllocPrivate* function allocates a private-data word and returns an index through which the word can be accessed.

Each task contains its own array of private-data words, each of which is identified by a particular array index value. A call to *TaskAllocPrivate* returns an index that can be used to access the corresponding private-data word in whatever task is running at the time of the access. In other words, a single call to the function allocates a private-data word in each task in the system.

The purpose of this function is to allow a library or group of related functions to maintain its own private context within each task that uses the library. By calling the *TaskGetPrivate* and *TaskSetPrivate* functions within the context of a particular task, the library can access its private context within that task.

If a single private-data word is not large enough to store all the private context data required by a library, the library's initialization routine can allocate a buffer of the necessary size and load a pointer to the buffer into the private-data word.

A typical task may consist of an application program that makes calls to functions in several software libraries. Each library may need to keep track of task-specific data that it would prefer to hide both from the application program and from the other libraries. The private-data mechanism facilitates this data hiding.

The private-data index is valid across all tasks, regardless of whether a task is created before or after the call to the *TaskAllocPrivate* function.

If the function is unable to allocate a private-data word, it posts a task error and returns a value of –1.

**Syntax**

long TaskBindPort(long *portId*, long *flagNum*);

**Description**

The *TaskBindPort* function binds a port to one of the calling task's event flags.

Argument *portId* is a port ID that identifies the local port that is to be bound. It can be expressed in either node-local or node-global format. The port must have been previously opened.

Argument *flagNum* is an event flag number and is restricted to the range 0 to 31. The function ignores all but the five LSBs of the argument.

Each task descriptor contains a 32-bit event register, each bit of which is an event flag that can be bound to a port or a semaphore. Once an event flag is bound to a port, the task can monitor the event flag to determine the status of the port at any instant. If the port contains one or more messages, the flag's value is 1. If the port is empty, the flag is 0.

An individual event flag can be bound to at most one port or semaphore at a time. Similarly, a port can be bound to at most one event flag.

If either (1) you attempt to bind a port or event flag that is already bound, or (2) argument *portId* is not a valid local port ID, the function posts a task error and returns a value of 0. Otherwise, a nonzero value is returned to indicate success.

**Syntax**

long TaskBindSema(long *semaId*, long *flagNum*);

**Description**

The *TaskBindSema* function binds a semaphore to one of the calling task's event flags.

Argument *semaId* is a semaphore ID that identifies the semaphore that is to be bound. The semaphore must have been previously opened.

Argument *flagNum* is an event flag number and is restricted to the range 0 to 31. The function ignores all but the five LSBs of the argument.

Each task descriptor contains a 32-bit event register, each bit of which is an event flag that can be bound to a semaphore (or port). Once an event flag is bound to a semaphore, the task can monitor the event flag to determine the status of the semaphore at any instant. If the semaphore contains a nonzero signal count, the flag's value is 1. If the semaphore's signal count is 0, the flag is 0.

An event flag can be bound to, at most, one semaphore (or port) at a time. Similarly, a semaphore can be bound to, at most, one event flag.

If either (1) you attempt to bind a semaphore or event flag that is already bound, or (2) argument *semaId* is not a valid semaphore ID, the function posts a task error and returns a value of 0. Otherwise, a nonzero value is returned to indicate success.

**Syntax**

long TaskCheckSema(long *semaId*);

**Description**

The *TaskCheckSema* function checks a semaphore to determine whether a signal is available but does not wait if one is not available. This is a nonblocking version of the *TaskWaitSema* function.

Argument *semaId* is a semaphore ID that identifies the semaphore that is to be checked. The semaphore must have been previously opened.

If the semaphore's signal count is nonzero, the function decrements the signal count by 1 and returns a value of 1. If the signal count is 0, the function returns a value of 0. In either case, the function returns immediately without waiting.

For the sake of speed, the function does not verify that argument *semaId* is a valid semaphore ID. The results are undefined if the argument is not valid.

**Syntax**

void TaskClearExcep(long *excepMask*);

**Description**

The *TaskClearExcep* function clears the specified exceptions in the calling task.

Argument *excepMask* is a 32-bit mask in which bits 0 to 31 (0 is the LSB) correspond to exception flags 0 to 31, respectively. A bit value of 1 in the argument indicates that the corresponding exception flag is to be cleared.

The function ignores an attempt to clear an exception that is already cleared.

**Syntax**

long TaskClosePort(long *portId*);

**Description**

The *TaskClosePort* function closes a local port that was previously opened by a call to the *TaskOpenPort* function. The port's ID is deleted, and its storage is deallocated.

Argument *portId* is the port ID of the local port that is to be closed. It can be expressed in either node-local or node-global format.

If the port contains any unreceived messages, the function attempts to direct each message toward its reclamation port; if that is not possible, it deallocates the message's storage.

If any tasks are waiting at the port, the function posts a task error in each of the tasks and schedules the task to run. When the task begins its next execution, the *TaskReceiveMsg* function returns to the task a null pointer value.

If the port is bound to an event flag in a task, the function unbinds the port. If the task has selected the event flag to wait on, the function posts a task error in the task and schedules the task to run. When the task begins executing again, the *TaskWaitEvents* function returns with a value of 0.

If argument *portId* is not the valid ID of a local port, the function posts a task error and returns a value of 0. Otherwise, the function returns a nonzero value to indicate success.

**Syntax**

long TaskCloseSema(long *semaId*);

**Description**

The *TaskCloseSema* function closes a semaphore that was previously opened by a call to the *TaskOpenSema* function. The semaphore's ID is deleted, and its storage is deallocated.

Argument *semaId* is the semaphore ID.

If any tasks are waiting at the semaphore, the function posts a task error in each of the tasks and schedules the task to run. When the task begins its next execution, the *TaskWaitSema* function returns a value of 0 to the task.

If the semaphore is bound to an event flag in a task, the function unbinds the semaphore. If the task has selected the event flag to wait on, the function posts a task error in the task and schedules the task to run. When the task begins executing again, the *TaskWaitEvents* function returns with a value of 0.

If argument *semaId* is not a valid semaphore ID, the function posts a task error and returns a value of 0. Otherwise, the function returns a nonzero value to indicate success.

**Syntax**

long TaskCreate(long *taskId*, void (**taskFunc*)(void *),
void **taskArg*, long *priority*, long *stackSize*);

**Description**

The *TaskCreate* function creates a new task by allocating a task descriptor and the stack space to be used by the task.

Argument *taskId* is the task ID. If the value of *taskId* is specified as –1, this directs the function to automatically generate a valid task ID. If a value other than –1 is specified for *taskId*, the function verifies that the value corresponds to a task ID that is both valid and available.

Argument *taskFunc* is a pointer to the task function, and argument *taskArg* is the single value that is passed to the task function as an argument. The program for each task is written in the form of a standard C function (referred to as a *task function*) that accepts a single argument and returns no value.

Argument *priority* is the task's initial priority level. Task priorities are restricted to the range 0 to 31, with larger numbers representing higher (or more urgent) priorities. The function ignores all but the five LSBs of the argument.

Argument *stackSize* specifies the size (in bytes) of the stack area to be allocated for the task.

The newly created task's initial state is SUSPENDED; it does not begin executing until a call to *TaskResume* causes it to enter the READY state (see subsection 2.8.5, *Task States*). This means that the call to *TaskCreate* always returns control to the caller before the new task has a chance to execute. Once the new task is resumed, the functions in the init-list are executed before jumping to the entry point of the task function.

If the function is either (1) unable to allocate storage for the stack or the task descriptor, or (2) unable to generate a valid task ID, it posts a task error and returns a value of –1. Otherwise, it returns a valid task ID to indicate success.

**Syntax**

void TaskExit(void);

**Description**

The *TaskExit* function exits the calling task by removing it from the head of the ready queue and freeing the storage occupied by its task descriptor and stack.

Just prior to deallocating the stack and task descriptor, the function executes the functions in the exit-list.

A task can also exit by returning directly from the task function.

**Syntax**

void TaskFreeMsg(void **msgBody*);

**Description**

The *TaskFreeMsg* function frees a message by deallocating the storage associated with the message.

Argument *msgBody* is a pointer to the message body.

In a tightly coupled multiprocessor system, a shared memory buffer containing a message may be passed from one processor node to another by simply exchanging pointers. The storage for the message can be deallocated, however, only by the processor that originally allocated the message.

Only messages allocated locally by the *TaskAllocMsg* function can be freed by *TaskFreeMsg.* If a processor calls *TaskFreeMsg* in an attempt to free a message allocated by another processor, the function can record information about the message to aid in debugging, but cannot retrieve the message's storage for reuse. The function similarly disposes of a message initialized by a call to *TaskInitMsg*.

The results are undefined if argument *msgBody* is not a valid message pointer.

**Syntax**

long TaskGetExcep(void);

**Description**

The *TaskGetExcep* function returns a 32-bit mask that indicates the status of exception flags 0 to 31 within the calling task. Bits 0 through 31 of the mask (0 is the LSB) correspond to exception flags 0 through 31, respectively. A mask bit with a value of 1 indicates that the corresponding exception is pending.

**Syntax**

long TaskGetFreeEvents(void);

**Description**

The *TaskGetFreeEvents* function returns a 32-bit mask that indicates which of the calling task's 32 event flags are free (that is, not bound to a port or semaphore). Bits 0 through 31 of the mask (0 is the LSB) correspond to event flags 0 through 31, respectively. A mask bit with a value of 1 indicates that the corresponding event flag is free.

**Syntax**

long TaskGetLastError(void);

**Description**

The *TaskGetLastError* function returns a code that identifies the last task error and clears the task error code. If no error has occurred since the last time the task error was cleared, the function returns a value of 0. A nonzero return value indicates an error.

A task error is an error that occurs within a task during a call to a kernel function. Each task descriptor contains a task error code. When the kernel creates a task, it clears the task's error code to 0. If a kernel function detects a task error, it sets the error code to a nonzero value, overwriting any previous error code. A kernel function that detects no error never modifies the stored error code. The only function that clears a task error is *TaskGetLastError*, which clears the calling task's error code back to 0.

A number of user-callable kernel functions return special values to indicate task errors. For example, when the *TaskReceiveMsg* function detects an error, it returns a null pointer value in place of a valid message pointer. This value is not the actual task error code, however. You can interrogate the kernel with the *TaskGetLastError* function to obtain the task error code that precisely specifies the nature of the error.

The error codes are defined in Appendix B, *Task Error Codes*.

**Syntax**

long TaskGetMsgSize(void **msgBody*);

**Description**

The *TaskGetMsgSize* function returns the length in bytes of the body of the specified message. A message's length is specified in its header. The header information is stored at a negative offset from the start of the message body.

Argument *msgBody* is a pointer to the message body.

The minimum valid message length is 0. A zero-length message consists of a fixed-length header and a zero-length body.

If argument *msgBody* is not a valid message pointer, the function posts a task error and returns a value of –1.

**Syntax**

long TaskGetPriority(void);

**Description**

The *TaskGetPriority* function returns the calling task's priority.

Each task is assigned an initial priority in the range 0 to 31 at the time that the task is created. Larger numbers correspond to higher (or more urgent) priorities. The *TaskSetPriority* function allows the task to dynamically change its priority.

**Syntax**

void *TaskGetPrivate(long *index*);

**Description**

The *TaskGetPrivate* function returns the contents of a private-data word that is identified by an index.

Argument *index* is the handle that identifies the private-data word. This is the value that was returned by a previous call to the *TaskAllocPrivate* function.

If argument *index* does not represent an allocated private-data word, the function returns a null pointer and posts a task error. Otherwise, it returns the value of the private-data word.

**Syntax**

long TaskGetReplyPort(void **msgBody*);

**Description**

The *TaskGetReplyPort* function returns a message's reply port ID. A message's reply port is specified in its header, which is stored at a negative offset from the start of the message body.

Argument *msgBody* is a pointer to the message body.

If the reply port is located in a foreign processor node, the port ID is in node-global format. If the reply port is local, the port ID can be in either node-local or node-global format. If no reply port is associated with the message, the function returns a value of –1 in place of a valid port ID.

If argument *msgBody* does not point to a valid message, the function posts a task error and returns a value of –1.

**Syntax**

long TaskGetTask(void);

**Description**

The *TaskGetTask* function returns the task ID of the calling task. This is the identifier that was assigned to the task when it was created.

If called from within the default task, the function returns a value of –1 in place of a valid task ID.

**Syntax**

void *TaskGetTaskArg(void);

**Description**

The *TaskGetTaskArg* function returns the calling task's task argument value. This is the value of the argument that was passed to the task at the time it was created.

A task program is written in the form of a standard C function (the **task function**) that accepts a single argument (the **task argument**). In principle, because the task function has direct access to the task argument, it can make the task argument value accessible to any function it calls by passing the value to the function as an argument. This may be inconvenient, however. The *TaskGetTaskArg* function allows a function at any level of function nesting to retrieve the task argument value directly from the current task's descriptor.

In a multitasking environment, several tasks may be instances of the same reentrant program. The use of global variables is, at best, tricky under these circumstances. The *TaskGetTaskArg* function offers a safe alternative to globals. The task argument is typically a pointer to a data structure containing task-specific information. The *TaskGetTaskArg* function makes this task-specific information globally accessible by providing direct access to the task argument from anywhere within the task.

If called from the default task, the function returns a null pointer value.

**Syntax**

void *TaskInitMsg(void **msgBuf*, long *size*, long *portId*);

**Description**

The *TaskInitMsg* function initializes a message whose storage you have allocated. The function initializes the message header and returns a pointer to the message body. The function does not initialize the contents of the message body.

The purpose of this function is to allow message buffers to be placed at user-specified locations in memory (for example, within a block of dual-ported memory) to facilitate communications with external processors.

Argument *msgBuf* is a pointer to a message buffer that you have allocated. The function places the message header at the beginning of the buffer and returns a pointer to the message body, which immediately follows the header.

Argument *size* is the length in bytes of the entire message buffer, including both header and body. (The value of argument *size* fixes the upper limit to the message size available by calling the *TaskResizeMsg* function.) The size of the resulting message body can be determined by calling the *TaskGetMsgSize* function.

Argument *portId* is a port ID that identifies the message's reclamation port. This is the port to which the message buffer is sent when the message it contains is discarded. Argument *portId* must identify a local port; that is, it must not reside in a foreign processor node. It can be expressed in either node-local or node-global format. The port must have been previously opened.

A valid reclamation port must be specified for every message initialized by the *TaskInitMsg* function. Otherwise, if the message is passed to the *TaskReclaimMsg* function as an argument, that function will be able to neither reclaim the message nor deallocate it.

If one of the following errors occurs, the function posts a task error and returns a null pointer value:

- Argument *portId* is not a valid port ID.
- The specified buffer size is not large enough to contain the fixed-size header structure.
- The resulting message body size exceeds 32 768 bytes.

**Syntax**

void TaskInitTasking(void);

**Description**

The *TaskInitTasking* function initializes the kernel of the MVP multitasking executive. The function sets the data structures within the kernel to their initial values.

The function must be called before any other kernel functions are called. The function is designed to be called only once following system power-up or reset.

On return from the call to *TaskInitTasking*, the program context from which the function was called has become the kernel's designated default task. This is the task that remains in the ready queue at all times and that executes when no other task is ready to run. The stack that is in use at the time of the call to *TaskInitTasking* is the stack that is assigned to the default task. An initial priority of 0 is assigned to the default task.

**Syntax**

long TaskInstallHandler(long *taskId*, void (* *handler*)(void),
long *selectMask*);

**Description**

The *TaskInstallHandler* function installs an exception handler in a previously created task. The handler is a user-specified function that is called each time an exception is raised in the task. The handler executes within the task's context.

Argument *taskId* is a task ID that identifies the task in which the exception handler is to be installed. Alternately, if a value of –1 is specified for this argument, the exception handler is installed in the calling task.

Argument *handler* is a pointer to the exception handler function. The handler is written as a standard C function that accepts no arguments and returns no value.

If a null pointer is specified for the handler argument, this means that no exception handler is to be installed in the task; this is the default following creation of a task.

Argument *selectMask* is a 32-bit mask that specifies the set of exceptions with which the exception handler is initially prepared to deal. Bits 0 through 31 of the argument (0 is the LSB) correspond to exception flags 0 through 31, respectively. A mask bit with a value of 1 indicates that the corresponding exception flag is selected.

The kernel invokes the handler each time one of the selected exceptions is raised in the task. If the exception occurs while the offending task is running, the exception handler begins executing immediately. Otherwise, the exception handler is invoked as soon as the task is scheduled to run and before the task begins executing again.

If argument *taskId* is neither –1 nor a valid task ID, the function posts a task error in the calling task and returns a value of 0. Otherwise, the function returns a nonzero value to indicate success.

**Syntax**

```
void TaskInstallMalloc(void *(* mallocFunc)(size_t),
                       void (* freeFunc)(void *));
```

**Description**

The *TaskInstallMalloc* function installs user-specified functions for dynamically allocating and freeing the blocks of memory storage used within the kernel.

The purpose of this function is to allow you to specify multitasking versions of the standard C functions *malloc* and *free* in the event that the versions of these functions provided in the default library do not support multitasking.

Argument *mallocFunc* is a pointer to a function that is compatible with the standard function *malloc.*

Argument *freeFunc* is a pointer to a function that is compatible with the standard function *free*.

The *TaskInitTasking* function automatically installs the *malloc* and *free* functions from the standard C library as defaults. The *TaskInstallMalloc* function permits user-defined allocation functions to be substituted for the defaults.

Any storage allocated by the kernel's default allocation functions can continue to be used safely after the defaults have been replaced by their user-defined equivalents. The results may be undefined, however, should you attempt to call the user-installed *free* function to deallocate storage that was allocated by the kernel's default *malloc* function.

The *size_t* data type in the syntax description above is defined in the standard C header file *stdlib.h*.

The four kernel functions that dynamically allocate storage are *TaskAllocMsg*, *TaskOpenPort*, *TaskOpenSema*, and *TaskCreate*.

**Syntax**

long TaskOpenPort(long *portId*);

**Description**

The *TaskOpenPort* function opens a port by reserving a port ID in the kernel's internal table space and allocating storage for the port. The value returned by the function is a port ID that is used to identify the port in subsequent kernel calls.

Argument *portId* is the port ID. By specifying the value of this argument as –1, you direct the function to automatically generate a valid port ID. Otherwise, the function verifies that the port ID that you have supplied is valid. The argument can be expressed in either node-local or node-global format.

If either (1) argument *portId* is not a valid port ID and is not –1, or (2) the function is unable to reserve space for a new port ID in the kernel's internal table space, the function posts a task error and returns a value of –1. Otherwise, the function returns a valid port ID to indicate success.

The port ID returned by the function is always in node-local format. If argument *portId* is given in node-global format, the function returns its node-local equivalent.

**Syntax**

long TaskOpenSema(long *semaId*, long *count*);

**Description**

The *TaskOpenSema* function opens a semaphore by reserving a semaphore ID in the kernel's internal table space and allocating storage for the semaphore. The value returned by the function is a semaphore ID that is used to identify the semaphore in subsequent kernel calls.

Argument *semaId* is the semaphore ID. By specifying the value of this argument as –1, you direct the function to automatically generate a valid semaphore ID. Otherwise, the function verifies that the semaphore ID that you have supplied is valid.

Argument *count* is the semaphore's initial signal count.

If either (1) argument *semaId* is not a valid semaphore ID and is not –1, or (2) the function is unable to reserve space for a new semaphore ID in the kernel's internal table space, the function posts a task error and returns a value of –1. Otherwise, the function returns a valid semaphore ID to indicate success.

**Syntax**

long TaskPollEvents(void);

**Description**

The *TaskPollEvents* function returns a mask containing the current state of the calling task's 32 event flags. This is essentially a nonblocking version of the *TaskWaitEvents* function, but unlike *TaskWaitEvents*, it requires no argument.

The return value is a 32-bit mask in which bits 0 through 31 (0 is the LSB) correspond to event flags 0 through 31, respectively. A mask bit with a value of 1 indicates that the corresponding event flag is set.

An event flag that is bound to a port indicates the instantaneous status of the port. If the port contains one or more messages, the flag is 1. If the port is empty, the flag is 0.

Similarly, an event flag that is bound to a semaphore is 1 if the semaphore's signal count is nonzero; otherwise, the flag is 0.

The value of a free (or unbound) flag is always 0.

**Syntax**

long TaskRaiseExcep(long *taskId*, long *excepNum*);

**Description**

The *TaskRaiseExcep* function posts a task error in the specified task.

Argument *taskId* is the task ID of the target task; that is, the task in which the exception is to be raised. The task must have been previously created. If a value of –1 is specified for this argument, the exception is raised in the calling task.

Argument *excepNum* is user-defined exception code. Exception codes are restricted to the range 0 to 31. The function ignores all but the five LSBs of the argument.

A task may optionally have an exception handler associated with it. If the target task has an exception handler, the kernel executes the handler as soon as possible after the exception is raised. In the case in which the calling task posts an exception in itself, the exception handler runs before the *TaskRaiseExcep* function returns from the call. Otherwise, the handler is entered immediately after the target task begins its next execution.

If the value supplied for argument *taskId* is neither –1 nor a valid task ID, the function posts a task error and returns a value of 0. Otherwise, it returns a nonzero value to indicate success.

**Syntax**

void *TaskReceiveMsg(long *portId*);

**Description**

The *TaskReceiveMsg* function receives a message from a local port. The value returned by the function is a pointer to the message body.

If a message is already queued at the port, the function removes the message from the port and returns immediately. Otherwise, the function causes the calling task to wait at the port until a message arrives.

Argument *portId* is a local port ID that identifies the local port from which the calling task is to receive a message. It can be expressed in either node-local or node-global format. The port must have been previously opened.

If a port's wait queue contains more than one task, the tasks are served messages on a first-come-first-served basis, regardless of their relative priorities.

If the port is bound to an event flag and a single message is queued at the port at the time that the function is called, the flag value changes from 1 to 0 to indicate that the port has become empty.

If either (1) the calling task is waiting at the port at the time that the port is freed, or (2) the default task calls *TaskReceiveMsg* and the message queue is empty, the function posts a task error and returns a null pointer value.

This is one of three blocking functions. The other two are *TaskWaitEvents* and *TaskWaitSema* (see pages EX:3-54 and EX:3-55, respectively).

For the sake of speed, the function does not verify that argument *portId* is a valid local port ID. The results are undefined if the argument is not valid.

**Syntax**

void TaskReclaimMsg(void **msgBody*);

**Description**

The *TaskReclaimMsg* function sends a message to its reclamation port. If no reclamation port is associated with the message, the function attempts to deallocate the buffer containing the message.

The purpose of this function is to retrieve the storage buffer containing a message that has been discarded.

Argument *msgBody* is a pointer to the message body.

In a tightly coupled multiprocessor system, two processors may send messages to each other by exchanging pointers to shared buffers. The reclamation port for a message buffer is always located in the allocation node, that is, in the processor node in which the buffer was originally allocated. If *TaskReclaimMsg* is called in a node other than the one in which the message was allocated, the function automatically routes the message to the allocation node so that it can be reclaimed or deallocated.

The results are undefined if argument *msgBody* is not a valid message pointer.

**Syntax**

void TaskRelayMsg(void **msgBody*);

**Description**

The *TaskRelayMsg* function relays a message that was sent from another processor node. The port ID of the message's destination port is contained in the message header. If the destination port lies in the local processor node, the function delivers the message directly to the port. Otherwise, it sends the message to the designated local routing port for the destination node.

Argument *msgBody* is a pointer to the message body.

The purpose of this function is provide a convenient means for a communications interface driver to dispose of incoming messages from other processor nodes. Unlike the *TaskRouteMsg* function, the *TaskRelayMsg* function does not require the caller to know anything about the message, including its destination port. Instead, the function retrieves this information from the message header.

If an error occurs in relaying the message to its destination port, the function attempts to retrieve the buffer containing the message by sending it to its reclamation port. If that fails, the function attempts to deallocate the message buffer. No attempt is made to notify the sender that an error has occurred.

The results are undefined if argument *msgBody* is not a valid message pointer.

If the local port (routing or destination) to which the message is delivered is bound to an event flag, the flag's behavior is similar to that described for the *TaskSendMsg* function (see page EX:3-43).

**Syntax**

long TaskResetSema(long *semaId*);

**Description**

The *TaskResetSema* function resets (that is, clears to 0) the signal count in a semaphore and returns the signal count's original value at the time the function was called. If the signal count is already 0, the function immediately returns a value of 0.

Argument *semaId* is a semaphore ID that identifies the semaphore that is to be reset. The semaphore must have been previously opened.

If the semaphore is bound to an event flag and the signal count is nonzero just before the call, the function changes the event flag from 1 to 0 to indicate that the signal count has been reset.

For the sake of speed, the function does not verify that the argument *semaId* is a valid semaphore ID. The results are undefined if the argument is not valid.

**Syntax**

long TaskResizeMsg(void **msgBody*, long *size*);

**Description**

The *TaskResizeMsg* function allows you to resize a message, subject to the physical limits of the storage buffer in which the message resides.

The purpose of this function is to make interprocessor copying of messages more efficient by copying only the messages themselves, not the undefined data at the end of the physical storage buffer that was allocated to contain the message.

Argument *msgBody* is a pointer to the message body.

Argument *size* specifies the requested new size (in bytes) of the message body. The minimum size is 0, which is the body size of a message consisting of a header only. If argument *size* exceeds the size of the available message buffer, the function resizes the message to the limits of the buffer and returns this value in lieu of the requested size. You should compare the returned value to the requested size to detect this situation.

If argument *size* is specified as –1, the function sets the message size to the maximum value that can be accommodated by the buffer and returns this value.

The function never relocates the message and never alters the contents of the message body.

If argument *msgBody* is not a valid message pointer, the function posts a task error and returns a value of –1.

**Syntax**

long TaskResume(long *taskId*);

**Description**

The *TaskResume* function resumes a suspended task and schedules it to run.

Argument *taskId* is the task ID of the task that is to be resumed. The task must have been previously created. The function verifies that the argument value is a valid task ID.

If the target task's state at the time of the call is SUSPENDED, the function changes its state to READY and inserts the task into the ready queue to await execution. If the task's state is WAITSUSPEND at the time of the call, the function changes its state to WAITING. If the task's state is READY or WAITING, the function returns immediately without taking any action. See subsection 2.8.5, *Task States,* for more information.

A task contained in a wait queue at a port or semaphore may be in either the WAITING or the WAITSUSPEND state. A port or semaphore treats all waiting tasks the same, regardless of which of these two states they are in.

If argument *taskId* is not a valid task ID, the function returns a value of 0 and posts a task error in the calling task. Otherwise, the function returns a nonzero value to indicate success.

**Syntax**

long TaskRouteMsg(void *_msgBody_, long _destPortId_);

**Description**

The *TaskRouteMsg* function routes a message to a global port; that is, a port that may either be local or reside in a foreign processor node.

The purpose of this function is to provide transparent access to ports in any processor node of a multiprocessor system.

Argument *msgBody* is a pointer to the message body.

Argument *destPortId* is the destination port ID. It may be specified either in node-global or node-local format. The port designated by the argument must have been previously opened.

If the destination port lies in the local node, the function delivers the message directly to the port. If the destination lies in a foreign processor node, the function routes the message through the designated local routing port for that node.

If the message header contains a reply port ID that is specified in node-local format, the function automatically converts the ID to node-global format before sending the message to a foreign node.

If the local port (routing or destination) to which the message is delivered is bound to an event flag, the flag's behavior is similar to that described for the *TaskSendMsg* function.

If one of the following errors occurs, the function posts a task error and returns a value of 0:

- The destination port is local, but the specified port ID is not valid.
- The destination port lies in a foreign node, but the routing table contains no entry for that node.
- Argument *msgBody* is not a valid message pointer.

Otherwise, the function returns a nonzero value to indicate success.

**Syntax**

void TaskSelectExcep(long *excepMask*);

**Description**

The *TaskSelectExcep* function selects a subset of exceptions that the calling task's exception handler is prepared to handle. Only selected exceptions can cause the exception handler to run.

Argument *excepMask* is a 32-bit mask in which bits 0 to 31 (0 is the LSB) correspond to exception flags 0 to 31, respectively. A bit value of 1 in the argument indicates that the corresponding exception flag is selected.

The default is that no exceptions are selected. This default is in effect immediately after the task is created.

You load the initial select mask for an exception handler when you call *TaskInstallHandler* to install the handler.

**Syntax**

void TaskSendMsg(void **msgBody*, long *portId*);

**Description**

The *TaskSendMsg* function sends a message to a local port.

This function supports efficient transmission of messages to local ports but does not support transmission to ports in foreign processor nodes.

Argument *msgBody* is a pointer to the message body.

Argument *portId* is a port ID that identifies the message's destination port. It can be expressed in either node-local or node-global format. The port must have been previously opened.

If one or more tasks is waiting at the port, the message bypasses the port and is delivered directly to the task at the head of the wait queue. In this case, if the port is bound to an event flag, the flag remains 0.

If no task is waiting at the port when the message arrives, the message is appended to the tail of the message queue. If the message queue was empty just before the message arrived and the port is bound to an event flag, the event flag changes from 0 to 1.

For the sake of speed, the function does not verify that argument *portId* is a valid local port ID. The results are undefined if argument *portId* is not valid or argument *msgBody* is not a valid message pointer.

**Syntax**

long TaskSetMsgRoute(long *nodeNum*, long *portId*);

**Description**

The *TaskSetMsgRoute* function adds an entry to the message-routing table, which contains the information needed to route messages to foreign processor nodes.

Argument *nodeNum* is the destination node number and is also an index into the routing table. Each table entry contains the ID of the port through which messages bound for the corresponding node are routed.

Argument *portId* is a port ID that identifies the local routing port. It can be expressed in either node-local or node-global format. The port must have been previously opened.

By specifying a value of –1 for argument *portId*, you indicate that argument *nodeNum* is the local node number. Before communicating with a foreign processor node, not only must the foreign processor's node number be entered into the routing table, but so must the local processor's node number.

As long as communications are restricted to local ports, you need not specify any routing-table information. In this case, all the destination port IDs specified to the *TaskRouteMsg* function must be given in node-local format.

If either (1) argument *portId* does not identify a valid local port or (2) the value of argument *nodeNum* exceeds the size of the routing table, the function posts a task error and returns a value of 0. Otherwise, the function returns a nonzero value to indicate success.

**Syntax**

long TaskSetPriority(long *priority*);

**Description**

The *TaskSetPriority* function sets the calling task's priority to the specified value and returns the task's previous priority.

Argument *priority* is the value to which the calling task's priority is to be changed. Priorities are constrained to the range 0 to 31, with larger numbers representing higher (or more urgent) priorities. The function ignores all but the five LSBs of the argument.

If the calling task's new priority is lower than its previous priority, the function checks to see if another task with a higher priority is ready to run. If so, the other task begins executing and the calling task is scheduled to run at a later time.

This function is designed to allow the calling task to continue to execute whenever possible. Only if a higher priority task is ready to run is the calling task preempted.

One use of this function is to allow interrupt service routines (ISRs) to inhibit task preemption by setting their priorities high. When an ISR completes, it should restore the interrupted task's original priority before returning from the interrupt. An ISR uses the stack and environment of the task that was running at the time the interrupt occurred.

**Syntax**

long TaskSetPrivate(long *index*, void **val*);

**Description**

The *TaskSetPrivate* function sets a private-data word in the calling task to a specified value.

Argument *index* is the handle that is used to identify the private-data word. This is the value returned from a previous call to the *TaskAllocPrivate* function.

Argument *val* is the value that is to be copied into the specified private-data word. The argument is cast as a pointer to void, but it may actually be an integer or a pointer to a user-defined structure.

The function verifies that argument *index* identifies a valid private-data word. If the argument is not valid, the function posts a task error and returns a value of 0. Otherwise, the function returns a nonzero value to indicate success.

**Syntax**

void TaskSetReplyPort(void **msgBody*, long *portId*);

**Description**

The *TaskSetReplyPort* function loads the specified reply port ID into the header of a message. A task that later receives the message can retrieve the reply port ID by calling the *TaskGetReplyPort* function.

Argument *msgBody* is a pointer to the message body.

Argument *portId* is the reply port ID. It can be specified either in node-global or node-local format. You can specify argument *portId* as –1 to indicate that no reply port is associated with the message.

Before the *TaskRouteMsg* function sends a message to a foreign processor node, it checks to see if the reply port ID contained in the message header is in node-local format. If so, it automatically converts the reply port ID to node-global format before routing the message to the destination port.

The reply port specified by argument *portId* is typically a local port, but it need not be. The function performs no checking to verify that the argument is a valid port ID.

**Syntax**

void TaskSignalSema(long *semaId*);

**Description**

The *TaskSignalSema* function signals a counting semaphore.

Argument *semaId* is a semaphore ID that identifies the semaphore to be signaled. The semaphore must have been previously opened.

If one or more tasks are waiting on the semaphore, the signal bypasses the semaphore and is delivered directly to the task at the head of the wait queue. In this case, if the semaphore is bound to an event flag, the flag remains 0.

If no task is waiting on the semaphore when the signal arrives, the semaphore's signal count is incremented by 1. If the signal count was 0 just before the signal arrived and the semaphore is bound to an event flag, the event flag changes from 0 to 1.

For the sake of speed, the function does not verify that argument *semaId* is a valid semaphore ID. The results are undefined if the argument is not valid.

**Syntax**

long TaskSuspend(long *taskId*);

**Description**

The *TaskSuspend* function suspends a task. A suspended task remains inactive indefinitely or until another task calls the *TaskResume* function to resume the task.

Argument *taskId* is the task ID of the task that is to be suspended. The task must have been previously opened. Alternately, the calling task can specify a value of –1 for this argument to indicate that it wants to suspend itself. If a value other than –1 is specified, the function verifies that the argument value is a valid task ID.

When the calling task suspends a task other than itself, the target task may be waiting on a port, semaphore, or event register at the time that the *TaskSuspend* call is made. Instead of immediately changing the target task's state from WAITING to SUSPENDED, the function changes its state to WAITSUSPEND. In this state, the target task continues to wait, but when the wait ends, the task's state changes to SUSPENDED instead of READY. Should a call to the *TaskResume* function be made on behalf of the target task while it is in the WAITSUSPEND state, the task's state changes back to WAITING. See subsection 2.8.5, *Task States,* for more information.

A task contained in a wait queue at a port or semaphore can be in either the WAITING or the WAITSUSPEND state. Ports and semaphores treat all waiting tasks the same way, regardless of which of these two states they are in.

If the function is called to suspend a task that is in the READY or WAITING state, the function changes the task's state to SUSPENDED or WAITSUSPEND, respectively, and returns a nonzero value to indicate success. If the function is called to suspend a task that is in the SUSPENDED or WAITSUSPEND state, the function returns immediately without taking any action and returns a nonzero value to indicate success.

In the event of (1) a task ID that is not valid and is not –1, or (2) an attempt by the default task to suspend itself, the function posts a task error and returns a value of 0.

**Syntax**

long TaskUnbindPort(long *portId*);

**Description**

The *TaskUnbindPort* function unbinds a port that has been previously bound to one of the calling task's event flags.

Argument *portId* is a local port ID that identifies the port that is to be unbound. It can be expressed in either node-local or node-global format.

If one of the following errors occurs, the function posts a task error and returns a value of 0:

- The specified port ID is not valid.
- The port is not bound to an event flag.
- The event flag to which the port is bound belongs to a task other than the calling task.

Otherwise, the function returns a nonzero value to indicate success.

**Syntax**

long TaskUnbindSema(long *semaId*);

**Description**

The *TaskUnbindSema* function unbinds a semaphore that has previously been bound to one of the calling task's event flags.

Argument *semaId* is a semaphore ID that identifies the semaphore that is to be unbound.

If one of the following errors occurs, the function posts a task error and returns a value of 0:

- The specified semaphore ID is not valid.
- The semaphore is not bound to an event flag.
- The event flag to which the semaphore is bound belongs to a task other than the calling task.

Otherwise, the function returns a nonzero value to indicate success.

**Syntax**

long TaskValidatePort(long *portId*);

**Description**

The *TaskValidatePort* function validates a port ID.

This function allows a calling task to validate a port ID before passing the ID to a function such as *TaskSendMsg* that does not validate port IDs.

Argument *portId* is the port ID to be validated. To qualify as valid, the ID must be that of a local port that has previously been opened. The port ID can be expressed in either node-local or node-global format.

If the port ID is not valid, the function returns a value of 0 but does not post a task error. Otherwise, the function returns a nonzero value to indicate that the port ID is valid.

**Syntax**

long TaskValidateSema(long *semaId*);

**Description**

The *TaskValidateSema* function validates a semaphore ID.

This function allows a calling task to validate a semaphore ID before passing the ID to a function such as *TaskSignalSema* that does not validate semaphore IDs.

Argument *semaId* is the semaphore ID to be validated. To qualify as valid, the ID must be that of a semaphore that has previously been opened.

If the semaphore ID is not valid, the function returns a value of 0, but does not post a task error. Otherwise, the function returns a nonzero value to indicate that the semaphore ID is valid.

**Syntax**

long TaskWaitEvents(long *selectMask*);

**Description**

The *TaskWaitEvents* function waits for any one of a set of events selected by the calling task to occur.

The purpose of the function is to allow the calling task to wait on multiple events simultaneously. If one of the selected events has already occurred at the time of the call, the function returns immediately. Otherwise, it waits until one of the selected events occurs.

Argument *selectMask* is a 32-bit mask that indicates which event flags the calling task has selected to wait for. Bits 0 through 31 of the argument (0 is the LSB) correspond to the calling task's event flags 0 through 31, respectively. A mask bit with a value of 1 indicates that the corresponding event flag is selected.

The value returned by the function is a snapshot of the calling task's event register at the time that a selected event occurred (or was detected). The occurrence of the event does not in any way inhibit the bound ports and semaphores from continuing to operate. By the time the calling task responds to a message event in a port or a signal event in a semaphore, another task may have received the message or signal, and the corresponding event flag may once again be 0. Similarly, an event flag that was 0 at the time of the snapshot may be 1 (due to the arrival of a message or signal) by the time the calling task responds to the first event.

Each task contains a 32-bit event register. Each bit in the register is an event flag that can be bound to a port or semaphore. A bound event flag indicates the instantaneous status of the port or semaphore to which it is bound. The value of an event flag bound to a port is 1 if the port contains at least one message but is 0 if the port is empty. Similarly, an event flag to which a semaphore is bound is 1 if the semaphore contains a nonzero signal count but is 0 otherwise.

The return value indicates the status of all 32 event flags, not just those selected by the *selectMask* argument. The value of a free (or unbound) event flag is always 0.

In the event that (1) the calling task attempts to wait on a free (unbound) event flag, or (2) the task is waiting on an event flag at the time that the port or semaphore to which it is bound is freed by another task, the function posts a task error and returns a value of 0.

This is one of three blocking functions. The other two are *TaskReceiveMsg* and *TaskWaitSema*.

**Syntax**

long TaskWaitSema(long *semaId*);

**Description**

The *TaskWaitSema* function waits at a counting semaphore for a signal. If one or more signals has already arrived at the semaphore, the function decrements the semaphore's signal count and returns immediately. Otherwise, the function causes the calling task to wait at the semaphore until a signal arrives.

Argument *semaId* is a semaphore ID that identifies the semaphore that the task is waiting on. The semaphore must have been previously opened.

If a semaphore's wait queue contains more than one task, the tasks are signaled on a first-come-first-served basis, regardless of their relative priorities.

If the semaphore is bound to an event flag and the semaphore's signal count is 1 when the function is called, the flag value changes from 1 to 0 to indicate that the signal count has gone to 0.

If either (1) the calling task is waiting at the semaphore at the time that the semaphore is freed, or (2) the default task calls *TaskWaitSema* and the signal count is 0, the function posts a task error and returns a value of 0. Otherwise, the function returns a value of 1 to indicate success.

This is one of three blocking functions. The other two are *TaskReceiveMsg* and *TaskWaitEvents*.

For the sake of speed, the function does not verify that argument *semaId* is a valid local port ID. The results are undefined if the argument is not valid.

**Syntax**

void TaskYield(long *priority*);

**Description**

The *TaskYield* function rotates a group of functions that are sharing the processor in round-robin fashion. The function can be called from a task to voluntarily yield the processor to another task of the same priority if one is ready to run. It can also be called from an interrupt service routine to implement time-slicing under the control of a periodic timer.

Argument *priority* is the priority of the tasks that are to be rotated in round-robin fashion. Valid priorities are in the range 0 to 31. If you specify a value of –1 for this argument, the calling task's priority is used. If you specify a value other than –1, all but the five LSBs of the argument value are ignored by the function.

If the ready queue contains fewer than two tasks of the specified priority, the function has no effect. Otherwise, the tasks of the specified priority form a subqueue within the ready queue. The function removes the task from the head of this subqueue and inserts it at the tail of the subqueue. The task that moves to the head of the subqueue as a result of this operation will be the next task of the specified priority to execute.

If another task of the same priority is ready to execute and either: 1) argument *priority* is specified as –1, or 2) the calling task's priority matches that given in the argument, then the other task preempts the calling task before the function returns.

Chapter 4

# Internode Message Passing

This chapter describes communications between processor nodes in a multiprocessor system. It presents examples of multiprocessor systems that incorporate MVPs. The discussion deals primarily with processors that communicate through parallel interfaces and share access to a common block of memory.

The operation of the internode message manager is explained. The message manager is a special task that executes on the MVP's master processor (MP). It is fundamentally a device driver that handles the hardware interface between processors. The message manager's job is to cooperate with the kernel to route internode messages to and from the other processor nodes in a multiprocessor system.

The parallel interface between a pair of processors must be designed to support efficient communications. The discussion includes a description of a model hardware interface that relies primarily on block data moves for speed. It is designed to reduce the number of random accesses necessary to coordinate message passing between processors.

The internode message manager is discussed in the following sections:

**Topics**

## 4.1 Overview of Internode Message Passing

A typical system consists of an MVP and a host processor connected to each other through a parallel bus interface. The role of the internode message manager in this configuration is to accept request messages from the host and deliver them to the appropriate MP-resident server tasks. The message manager also routes reply messages from these tasks back to the host. To manage the host side of the interface, a host-resident driver routine can emulate the MVP executive's message communications protocol. Alternatively, some portion of the MVP executive can be ported to the host.

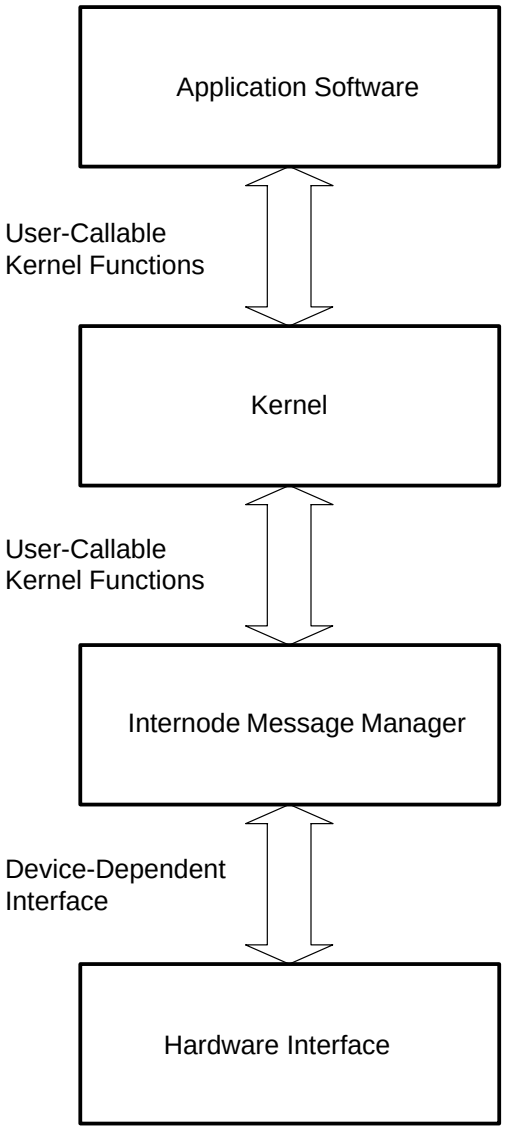
Another example of a multiprocessor is a dual-MVP system in which the MP in each MVP device runs its own private copy of the MVP executive. Running under the kernel as a task is a message manager that sends and receives messages across the interface between the two MVPs.

Figure 4–1 shows the functional software and hardware layers in the communications hierarchy of an MVP-based system. The application software, kernel, and internode message manager are all programs that run on the MP. The internode communications hardware is used to send messages to and receive messages from other processor nodes. The kernel's interfaces to the application software and to the internode message manager are both device-independent, whereas the interface to the internode communications hardware is device-dependent.

The application software in Figure 4–1 is likely to consist of a number of server tasks that run under the kernel and communicate with other processor nodes through the kernel's message-passing functions. The server tasks may receive service requests from client programs running on a host processor, for example.

The internode message manager in Figure 4–1 also interfaces to the kernel through the kernel's user-callable functions. During system initialization, the message manager instructs the kernel to open the routing ports that will be needed to locally route messages bound for foreign processor nodes. The message manager enters these ports into the kernel's message-routing table.

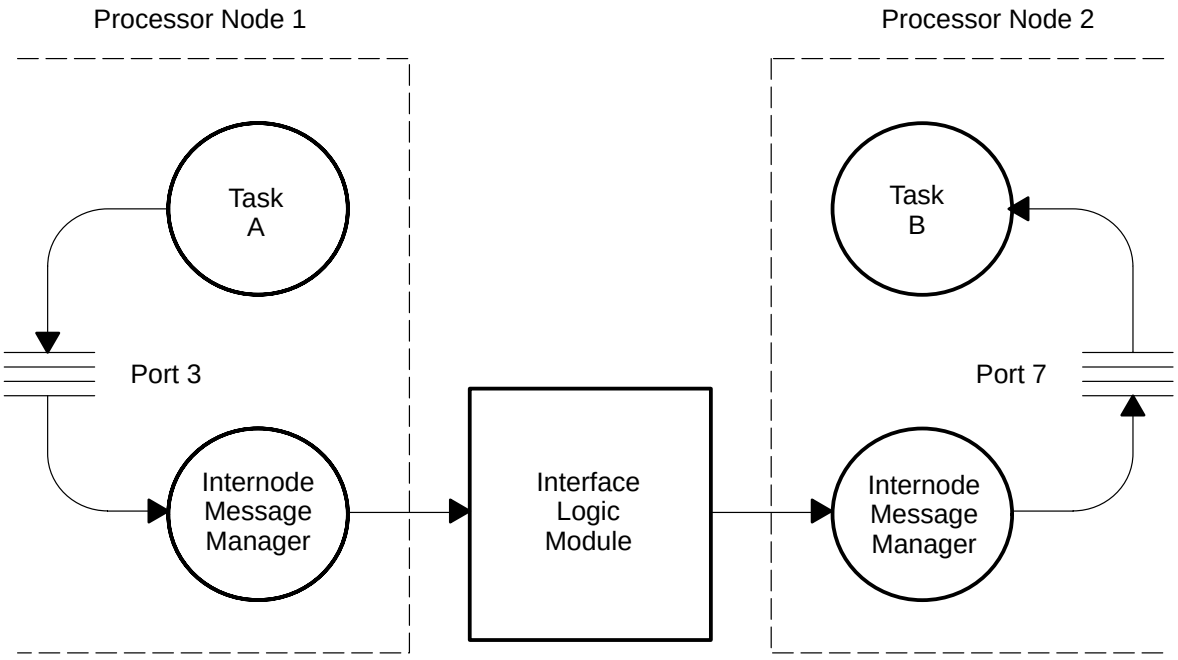
Figure 4–1. Functional Layers for Internode Message Passing



## 4.2 Internode Message Routing

Figure 4–2 shows an example of a message transfer through a parallel hardware interface. The message transfer is controlled by the interface logic module shown in the center of the figure. The internode message manager on either side of the hardware interface acts as the local device driver for the interface logic.

Figure 4–2. Message Transfer Across the Interface Between Two Processor Nodes




At the top left of Figure 4–2, Task A, which runs on processor node 1, sends a message to port 7 on processor node 2. Task B will receive the message when it arrives at the port. The first phase of the message transfer occurs in node 1:

1) Task A calls the kernel function *TaskRouteMsg* to send the message.

2) The kernel loads the destination port ID into the message header and sends the message to local port 3, which is node 1's local routing port for all messages destined for node 2.

3) The internode message manager, which has been waiting for a message to arrive at port 3, receives it and transmits it to node 2 through the interface logic in the center of Figure 4–2.

The message transfer completes in node 2 as follows:

1) Node 2's internode message manager responds to the arrival of the message by calling the kernel function *TaskRelayMsg* to relay the message on the next leg of its journey.

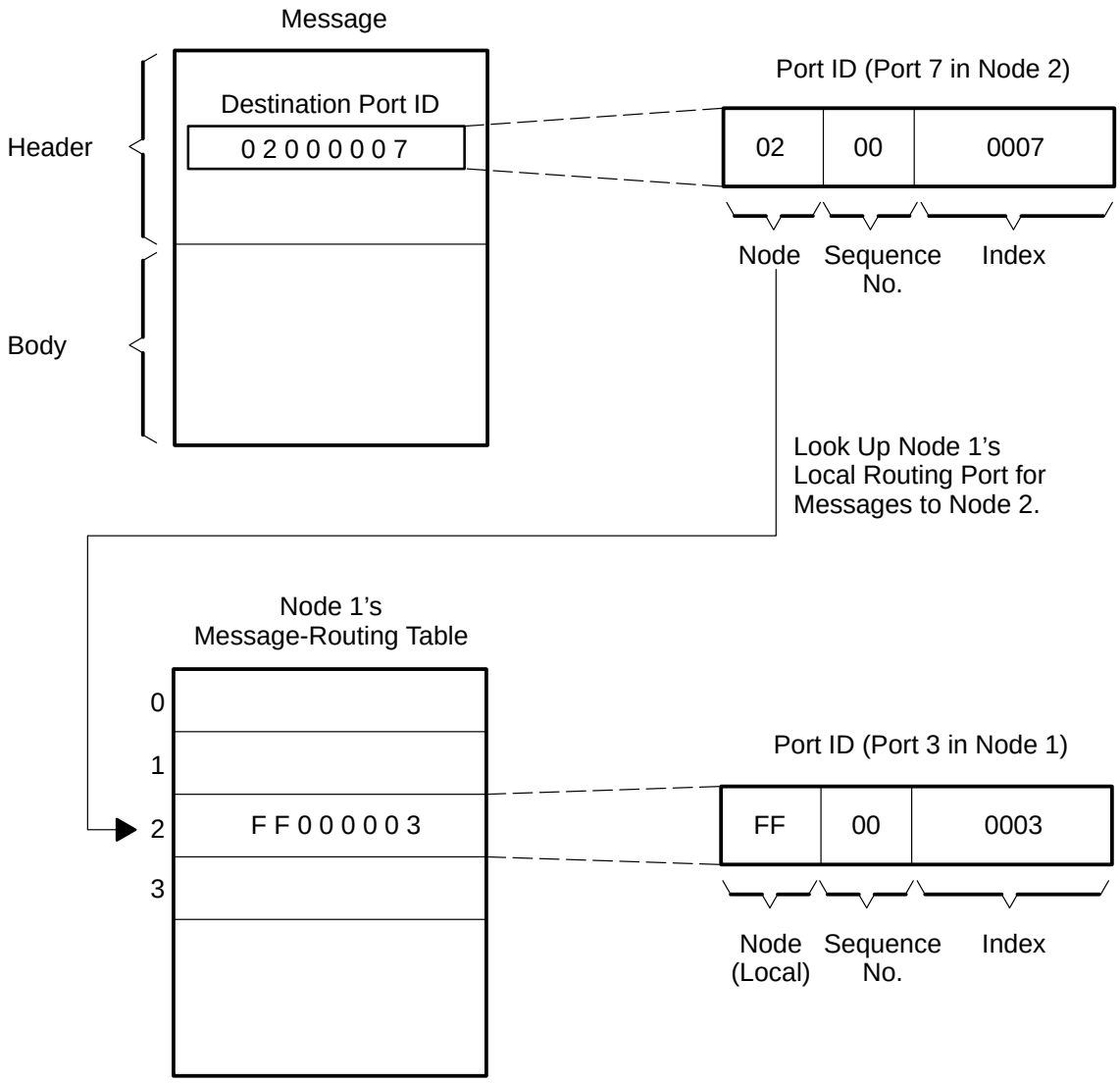
2) The kernel inspects the destination port ID in the message's header and determines that the message is destined for local port 7. It delivers the message to this port.

3) Task B, which has been waiting at port 7, receives the message.

Figure 4–3 shows in detail how the kernel executing on node 1 determines that port 3 is the local routing port for the message in the example of Figure 4–2. For concreteness, assume that ports 3 and 7 in Figure 4–2 have node-global port IDs 0x01000003 and 0x02000007, respectively. Note that the index portion of the ID is the number used to refer to the port in this example. The high byte of each ID contains the node number. The node-local form of port 3's ID, which is valid only in node 1, is 0xFF000003.

Node 1's message-routing table contains this ID in its entry for node 2; this entry indicates that all messages destined for node 2 are to be routed through port 3. Responding to Task A's call to *TaskRouteMsg*, the kernel puts port 7's ID, 0x02000007, into the message header's destination port ID field and then extracts the destination node number, 0x02, from the port ID's high byte. The kernel uses this number to index into the routing table and retrieve port 3's ID.

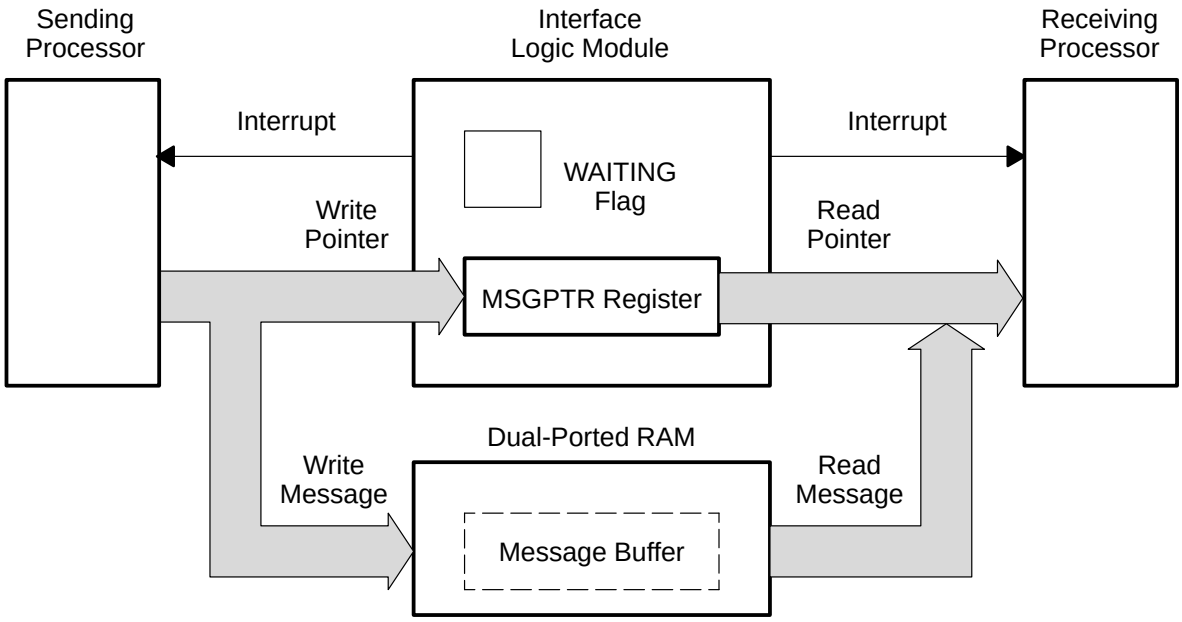
Figure 4–3. Look-Up Local Routing Port



## 4.3 Hardware Example

The example implementation of the internode message manager is based on the model hardware interface shown in Figure 4–4. The two main components in the interface are a dual-ported RAM and an interface logic module. Messages are passed through buffers in the dual-ported RAM that are accessible to the processors on either side of the interface. The protocol for transferring a message pointer from the sending processor to the receiving processor is performed through the interface logic module.

Figure 4–4. Hardware Interface Block Diagram




For simplicity, Figure 4–4 shows a single-duplex communications channel in which messages are transmitted from a sending processor on the left to a receiving processor on the right. Full-duplex communications would require two interface logic modules.

Assume that either or both of the processors in Figure 4–4 is an MVP. For efficiency, the MP utilizes block moves to transfer the contents of the message buffer between the dual-ported RAM and the MVP's on-chip RAM. These transfers can be performed by manipulating the MP's data cache, as described in Appendix A, *Cache Coherency*. The size of a data block is then a multiple of the cache subblock size, which is 64 bytes.

Accesses of the registers in the interface logic module are performed using the MP's *dst* and *dld* instructions (see Section 10.7, *Memory-Access Instructions,* in the *MVP Master Processor User's Guide*), which bypass the data cache to directly access external memory. The interface should be designed to minimize the number of such accesses required to transfer a message from one processor to another.

With these requirements in mind, the interface logic module in Figure 4–4 contains a 32-bit MSGPTR register that is used to pass message pointers and a WAITING flag that indicates when the sending processor is waiting to send a new message. To transmit a message, the sending processor does the following:

- Block-copies its message into a buffer located in the dual-ported RAM, and
- Loads a non-null message pointer into the MSGPTR register. When the sending processor loads the pointer into the register, the interface logic responds by automatically signaling an interrupt to the receiving processor.

The receiving processor responds to the interrupt by reading the message pointer in the MSGPTR register. This read operation is detected by the interface logic, which automatically clears (sets to 0) the MSGPTR register to indicate that it is available to send a new message. Before the sending processor can send another message, it reads MSGPTR to verify that the last message has been received. If MSGPTR is nonzero, however, the sending processor does not continue to poll the register. Instead it relies on logic associated with the MSGPTR register to detect the unsuccessful read and automatically set the WAITING flag to 1. That is, the interface logic interprets the read of a nonzero value as the processor's request to be interrupted as soon as the MSGPTR register is read (and cleared) by the receiving processor.

When the receiving processor reads MSGPTR, the interface logic checks the WAITING flag; if the flag is set, the logic signals an interrupt to the sending processor to inform it that the MSGPTR register is ready to hold the next message pointer. When the sending processor responds by writing the next message pointer to the MSGPTR register, this automatically clears the WAITING flag.

Some enhancements to the above scheme may further improve performance. For example, the single MSGPTR register can be replaced by a hardware FIFO to prevent the sender from having to wait for one message to be received before the next one can be sent.

Also, if a level-sensitive interrupt pin can be dedicated to the purpose, the interrupt signal to the sending processor can be modified to always reflect the instantaneous status of the MSGPTR register. The signal is at the active level when the MSGPTR register is ready to send another message; otherwise, it is inactive. This scheme eliminates the need for the sending processor to read the MSGPTR register before writing a new message pointer to it. Before sending a message, the sending processor internally polls the status of the interrupt signal. If it finds that MSGPTR is not ready, it can internally enable the interrupt associated with the interrupt signal and wait to be interrupted.

Chapter 5

# Parallel Processor Command Interface

The MVP's master processor (MP) issues commands to the on-chip parallel processors (PPs) through a software interface. The PPs are advanced DSPs that the MP uses as coprocessors. PPs typically assist MP tasks by accelerating the execution of low-level imaging and graphics operations.

This chapter describes the protocols and structures necessary to implement a PP command interface. While the example implementation in this chapter may not meet all your needs, it should provide you a useful starting point to tailor the interface to your specific requirements.

**Topics**

## 5.1 Overview of the Parallel Processor Command Interface

The MVP is a one-chip multiprocessor system. A single MVP device contains an MP and one or more PPs. The number of PPs that are physically present depends on the chip version.

On the MP, several tasks may execute concurrently under the MP-resident kernel of the MVP executive. These tasks typically provide services to client programs that run on a host processor.

The PPs, in turn, are used as coprocessors by MP tasks. By offloading processing work to the PPs, MP tasks can accelerate the servicing of the requests they receive from host-resident client programs. The MP tasks treat the PPs as shared processing resources.

In contrast to the multitasking software that executes on the MP, the software running on a PP is single-threaded. Its job is to interpret and execute a serial stream of commands that it receives from the MP. Within the context of MP-to-PP communications, a PP is the server, and an MP task is the client. When an MP task sends a command to a PP, the PP interprets the command and, if necessary, sends a reply back to the MP task.

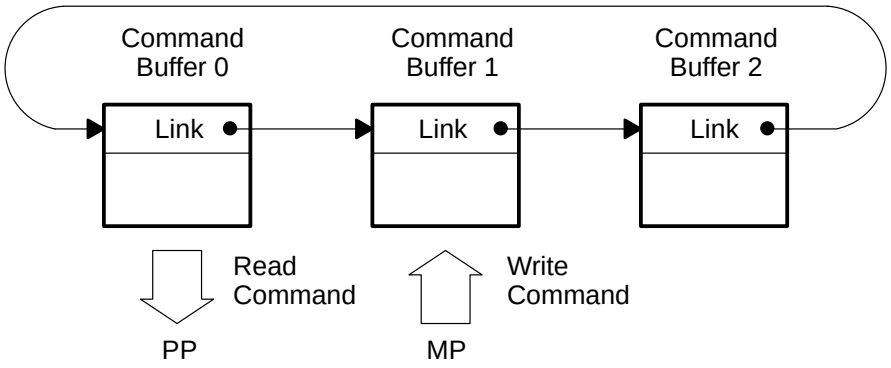
By convention, an MP task that owns a PP also owns all of the PP's local RAM. Associated with each PP are at least 6K bytes of data RAM and 2K bytes of parameter RAM. This RAM resides on the MVP chip and can be directly accessed not only by the PP itself, but also by the MP and the other PPs. Communications between a PP and the MP take place in buffers contained within the PP's local RAM.

The implementation of a PP command interface is generally application-dependent. The mechanisms necessary to support communications between the MP and PPs are embedded within the application code and are largely independent of the kernel functions in the MVP executive that executes on the MP. From the kernel's point of view, the software that manages the MP's interface to the PPs is effectively a device driver. This independence from the kernel gives you wide latitude in customizing the interface software to meet your proprietary needs.

## 5.2 Circular Command Queue

The MP sends commands to a PP through a group of fixed-size command buffers that are located in the PP's local parameter RAM. The buffers are linked to each other in circular fashion, as illustrated in Figure 5–1. The figure shows a circular queue containing three buffers, but queues that contain one, two, or four buffers, for example, can be implemented as well. The number of buffers should be chosen to accommodate the needs of the application.

Figure 5–1. Circular Queue of PP Command Buffers




Each command buffer contains a pointer to the next buffer in the linked list. The MP, after loading a new command into the current buffer, follows the link to the next buffer. The PP, after executing the command in the current buffer, follows the link to the next buffer. The PP always executes a series of commands strictly in the order in which they were issued by the MP.

A command buffer supports a simple producer-consumer exchange between the MP and PP. Within each buffer is a full/not-empty flag that the MP client task sets to 1 (full) to indicate that the buffer contains a valid command. When it has finished executing the command, the server PP clears this flag (to empty) to indicate to the MP that the buffer is available for a new command.

The following conventions ensure that commands are always executed in the same order in which they are issued:

- Once the MP has loaded a command into a command buffer, it sets the buffer's flag to full and follows the link to the next command buffer.
- Once the PP has executed the command specified in a buffer, it sets the buffer's flag to empty and follows the link to the next buffer.

- Before a client can issue a new command, it must wait until the next command buffer is marked as empty.
- Before a server PP can process a new command, it must wait until the next command buffer is marked as full.

The access rights to a command buffer are easily described. When the buffer is marked as empty, it is the exclusive property of the client. When it is marked as full, it belongs to the server PP. (The client, however, retains the right to modify the *intCode* field at any time, as described in subsection 5.2.6.)

### 5.2.1 Client Waiting

The client may have to wait on a command buffer's full/not-empty flag for one of two reasons:

- If the client is ready to load a new command into a command buffer but the buffer is not available (that is, the buffer is still marked as full)
- If the client has just issued a command that requires a reply but the reply is not yet available

In the second case above, the server PP returns the reply through the same command buffer in which the original command was delivered. It then marks the buffer as empty to inform the client that the reply is ready.

A command may or may not require a reply. When sending a PP a series of commands that do not require replies, the MP is free to continue issuing commands as long as more command buffers are available. After sending a PP a command that **does** require a reply, however, the MP cannot issue a second command to the same PP until it has received the reply to the first command.

Replies are sometimes necessary, but they do cause delays by stalling the command pipeline from the MP to the PP. The specification for a command, including whether the command requires a reply, is application-dependent. For the sake of efficiency, application programmers should attempt to limit the number of commands that require replies.

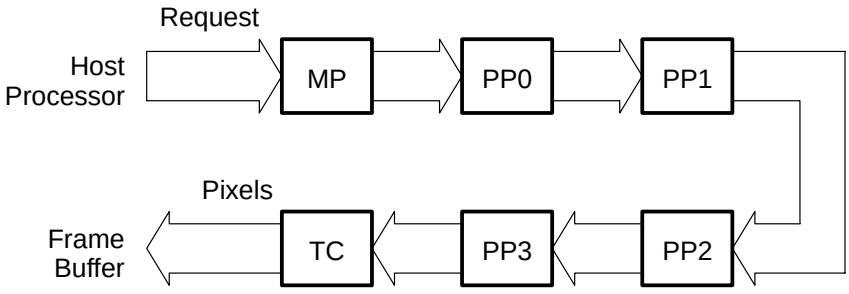
### 5.2.2 Client PPs and Pipelining

An MP task that owns two or more PPs may delegate to one PP the job of issuing commands to be executed by a second PP. Within the context of this arrangement, the first PP is the client, and the second PP is the server. Whether its client is an MP task or another PP is largely transparent to a server PP.

On an MVP chip that contains multiple PPs, commands can be directly pipelined from one PP to another without MP intervention. For example, given an MVP that contains four PPs, PPs 0 through 3 can be configured to form a multistage pipeline in which commands are passed from one processor to the next, as shown in Figure 5–2. The following are the pipeline stages for this illustration:

1) The host processor sends requests to an MP-resident task.
2) The MP task issues commands to PP0.
3) PP0 issues commands to PP1.
4) PP1 issues commands to PP2.
5) PP2 issues commands to PP3.
6) PP3 sends packet transfers to the transfer controller (TC).
7) The TC transfers the data from PP3 to the frame buffer in the MVP's external memory.

Figure 5–2. Configuring the MVP's On-Chip Processors to Form a Pipeline



### 5.2.3 PP Mailbox

A server PP can send a brief message to a client through a mailbox. The purpose of the message may be, for example, to inform the client that a command buffer it has been waiting for has become available. The client can be either an MP task or another PP. The length of a PP message is limited to 32 bits. PP messages are distinct from and independent of the messages that are exchanged between MP-resident tasks, which can be of arbitrary length.

The purpose of a PP message is to qualify the meaning of a message interrupt that the PP sends to one of the other processors on the MVP chip. A message interrupt is an MVP hardware mechanism that allows one on-chip processor to interrupt another by means of a dedicated interrupt. A PP can notify the MP or another PP of an event by sending a message interrupt. The processor that receives the interrupt determines the source of the interrupt by inspecting its interrupt-pending register. The interrupted processor then reads the message contained in the mailbox of the PP that sent the interrupt. The meaning assigned to a particular message code is determined within the application.

The mailbox is a word in the interrupting PP's parameter RAM that holds a 32-bit message. The address of the mailbox is known to the interrupting PP and to all processors that can receive message interrupts from the PP. Each server PP owns a single mailbox. The PP performs all its communications with other processors through this one mailbox. Application-dependent meanings are assigned to the various message values. The value of 0, however, is reserved to indicate an empty mailbox.

Once the receiving processor has read the message from the sending PP's mailbox, it writes 0 to the mailbox to mark the mailbox as empty. If the sending PP needs to send another message interrupt, it must wait until the mailbox becomes empty before proceeding. While it is waiting, the PP continually polls the mailbox.

### 5.2.4 Spinning

Continual polling is referred to as **spinning**. During the time that a PP spins on its mailbox, it performs no other useful processing. Because it is single-threaded, however, it has no other task to perform anyway.

One advantage of spinning is that when the mailbox is finally cleared, the PP can respond more rapidly than would be possible if the transition were detected within an interrupt service routine.

The PP's spinning on the mailbox does not steal memory cycles from processors that access other areas of on-chip RAM over the MVP's internal crossbar switch network. Memory contention can occur only when a second PP attempts to access the first PP's local parameter RAM, which contains the mailbox. Note that accesses by the MP are never delayed because the on-chip crossbar arbitration logic always assigns the MP's accesses a higher priority than any PP's accesses.

In addition to polling mailboxes, other instances may arise in which a PP spins on a memory word until another processor modifies the word. The current implementation, however, contains only one other instance: a server PP can spin on the full/not-empty flags in one of its command buffers. In general, the rule is that a PP spins only on a word in its own local RAM. The purpose of this rule is to avoid creating excessive memory contention.

### 5.2.5 The Full/Not-Empty Flag

A server PP continually polls a command buffer's flag while it is waiting for the flag to change from empty to full. The reason for allowing the PP to spin on the flag is similar to that given above for allowing it to spin on its own mailbox. Recall that while a server PP spins on one of its own command buffers, it accesses only its own local parameter RAM.

A client (an MP task or another PP), however, never spins on a command buffer flag. One reason to avoid polling is that it may cause excessive memory conflicts. The command buffer is located in the local RAM that belongs to the server PP. Repeated accesses of a server's local RAM by the client may interfere with accesses of the same RAM by the server PP itself. A second reason is that the client processor might be the MP. Because the MP supports multitasking, it is likely to have other work to perform while it is waiting for a command buffer to become available. Spinning would prevent it from performing the work.

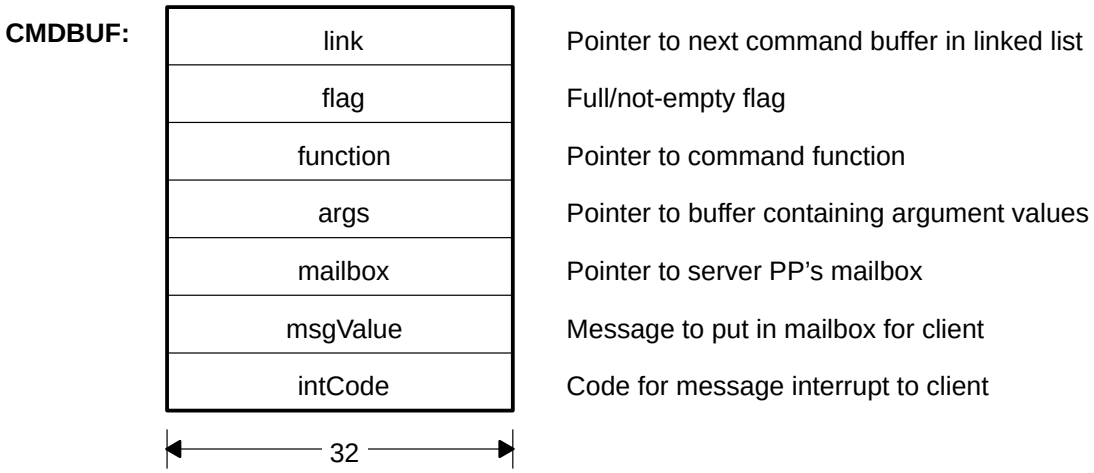
For these reasons, a server PP uses a message interrupt to notify a client when a command buffer becomes free. The client asks to be interrupted, however, only after it has already determined for itself that a command buffer that it needs is not yet available.

Before accessing a command buffer, the client first reads the full/not-empty flag in the buffer to check whether the buffer is available. If the buffer is marked as full, the buffer is unavailable and the client must wait until the server PP marks the buffer as empty. Rather than continue to poll the flag, however, the client requests a message interrupt from the server PP. The PP must respond by sending the interrupt as soon as the buffer becomes free. The client places this request in a special field (*intCode*) of the command buffer on which it is waiting. When the server PP frees the command buffer, it notifies the client by loading a message into its (the server PP's) mailbox and sending a message interrupt to the client.

## 5.2.6 Command Buffer

Each PP command buffer has the structure shown in Figure 5–3.

Figure 5–3. Structure of a PP Command Buffer




- The 32-bit *link* field contains a pointer to the next command buffer in the circularly linked list.

- The 32-bit *flag* field holds the full/not-empty flag. The client sets this flag to 1 (full) to indicate that the buffer contains a new command that is ready to be processed by the server PP. The server PP sets this flag to 0 (empty) to indicate that the command buffer is available for a new command from the client.

- The 32-bit *function* field contains a pointer to the PP-resident function that actually executes the command. (In an alternative implementation, this field could contain a command number, and the PP-resident command interpreter would be required to look up the function pointer in a command function table.)

- The 32-bit *args* field contains a pointer to a buffer containing the argument list. The size and structure of the argument buffer is entirely application-dependent. The argument buffer can be located anywhere in the server PP's local parameter RAM or data RAM. In the case of a command that requires a reply, the return values are placed in the argument buffer pointed to by the *args* field of the command buffer in which the original command was sent.

The last three fields in the command buffer structure contain the information needed by the server PP to interrupt a client that is waiting for a command buffer to become available.

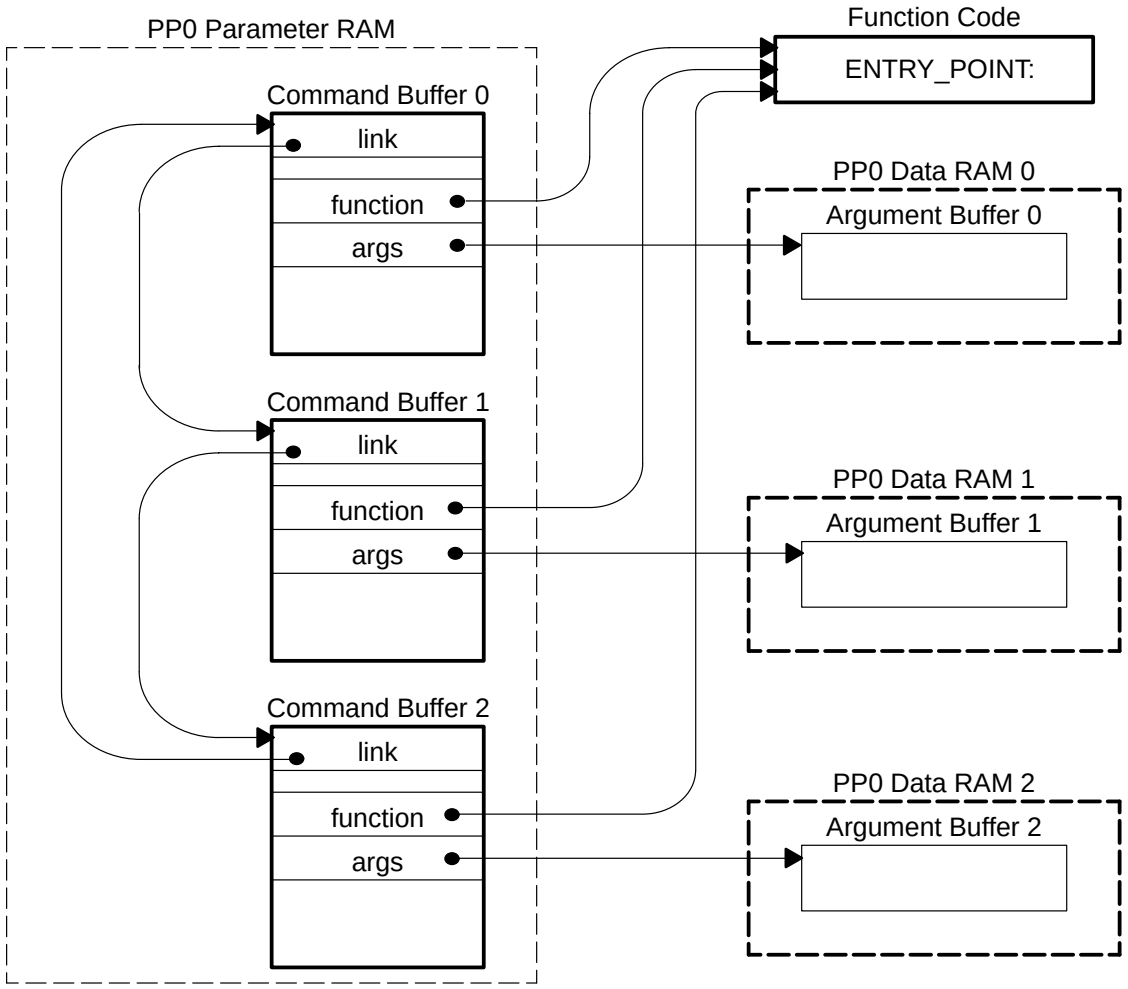
- The 32-bit *mailbox* field contains a pointer to the mailbox that belongs to the server PP that interprets the command. Because each PP has only a single mailbox, the *mailbox* fields of all the PP's command buffers contain the same pointer value.
- The 32-bit *msgValue* field contains the message that the server PP loads into the mailbox. This value qualifies the meaning of the message interrupt that the server PP sends to the client.
- The 32-bit *intCode* field contains a client's request to be interrupted when the command buffer becomes available. The command code stored in this field contains all the information necessary for the server PP to send a message interrupt to the client. A code of 0 is reserved to indicate that no message interrupt has been requested by the client. The meaning of a nonzero code is defined by the MVP machine architecture. To send the interrupt, the server PP loads this code into one of its internal registers and executes a *cmnd* instruction. During the execution of this instruction, the machine interprets the code to determine both the destination processor and the type of interrupt to be sent. For example, the command word 0x0000 2100 transmits a message interrupt to the MP.

### 5.2.7 Repeated Commands

Many graphics, audio, and image processing operations are iterative in nature. The same calculations, but with different data, are repeated many times. In an application of this sort, a server PP may receive from a client a series of essentially identical commands. Only the argument values change from one command to the next. A client can exploit these characteristics to avoid the overhead of updating certain fields in each PP command buffer before issuing a new command.

In the example of Figure 5–4, PP0 is configured to operate as a server to an MP-resident client task. Assume that the client sends a series of 1200 identical commands to the PP. PP0's parameter RAM contains three command buffers that are linked to form a circular queue. While executing these commands, the *function* and *args* fields in each command buffer remain unchanged over a long series of identical commands. Before issuing a new command, the client updates only the data in the argument buffer pointed to by *args*. The same function is used to execute each command, and each of the three argument buffers remains permanently associated with its respective command buffer. The client initializes the *function* and *args* fields once at the beginning of a series of identical commands. It avoids the unnecessary overhead of repeatedly loading these fields with precisely the same values before issuing each individual command in the sequence.

Figure 5–4. A Circular Queue of Three Command Buffers




Before issuing the 1200 identical commands, the MP task loads the *function* fields of the three command buffers with the same function pointer (this is the address of the ENTRY_POINT label shown in Figure 5–4). The *args* fields of command buffers 0, 1, and 2 are loaded with pointers to argument buffers 0, 1, and 2, respectively. In this example, one of PP0's three data RAMs is dedicated to each of the three argument buffers.

Before issuing the first command, the MP task loads the initial set of arguments into argument buffer 0, which is located at the beginning of PP0 data RAM 0. Before issuing the second command, the MP task loads the next set of arguments into argument buffer 1, and so on. In the course of issuing all 1200 commands in the sequence, the MP must issue each of the three command buffers 400 times.

## 5.3 Command Interpreter

The software that executes on a server PP has three components:

- The **function library** contains the routines that actually perform the operations specified in the commands.
- The **state information** is data that the PP maintains in its local RAM between commands; this data can be accessed as global variables by the various routines in the library.
- The **command interpreter** manages communications with the client and dispatches commands to specific routines for execution.

The command interpreter program that runs on a server PP is a simple program loop that continually waits for the next command to be issued to execute it. As described previously, the command buffers form a circularly linked list. The command interpreter follows the link from one command buffer to the next, executing the commands strictly in the sequence in which they were issued by the client.

The command interpreter executes a command by calling the function that is pointed to by the *function* field in the command buffer. The single argument passed to the function is the pointer value taken from the command buffer's *args* field, which points to a second buffer that contains the actual argument values. If the command requires a reply, any values to be returned to the client are loaded by the function into this argument buffer before the function exits and returns control to the command interpreter.

### 5.3.1 Initial Configuration

Before an MP task can begin issuing commands to a server PP, it must configure the PP by initializing the data structures that reside in the PP's local parameter RAM. These data structures include the command buffers and the mailbox, all of which are initially empty. To begin executing the command interpreter, the MP unhalts the PP and hands it a pointer to the first command buffer. At this point, the PP is ready to begin executing commands from the MP.

The data structures that are manipulated by the command interpreter have been designed to accommodate application-dependent modifications in the structure of the PP command interface with minimal changes to the command interpreter. For example, the specific number of command buffers in the circular queue can be altered by modifying only the initialization software that runs on the MP.

The argument buffer pointed to by the command buffer's *args* field can be located anywhere in the PP's local data or parameter RAM. The PP can always find this buffer through the pointer contained in the *args* field of the command buffer.

The command interpreter need not be aware of the identity of a client, which could be an MP task or another PP. All the information necessary to direct a message interrupt to the client is contained in the *intCode* field of the command buffer. Note that if the PP is in fact receiving commands from another PP instead of from the MP, the *intCode* field is simply modified so that the interrupt is transmitted to this other PP rather than to the MP. Whether the client program is executing on the MP or on another PP is transparent to the server PP.

### 5.3.2 PP Reallocation Overhead

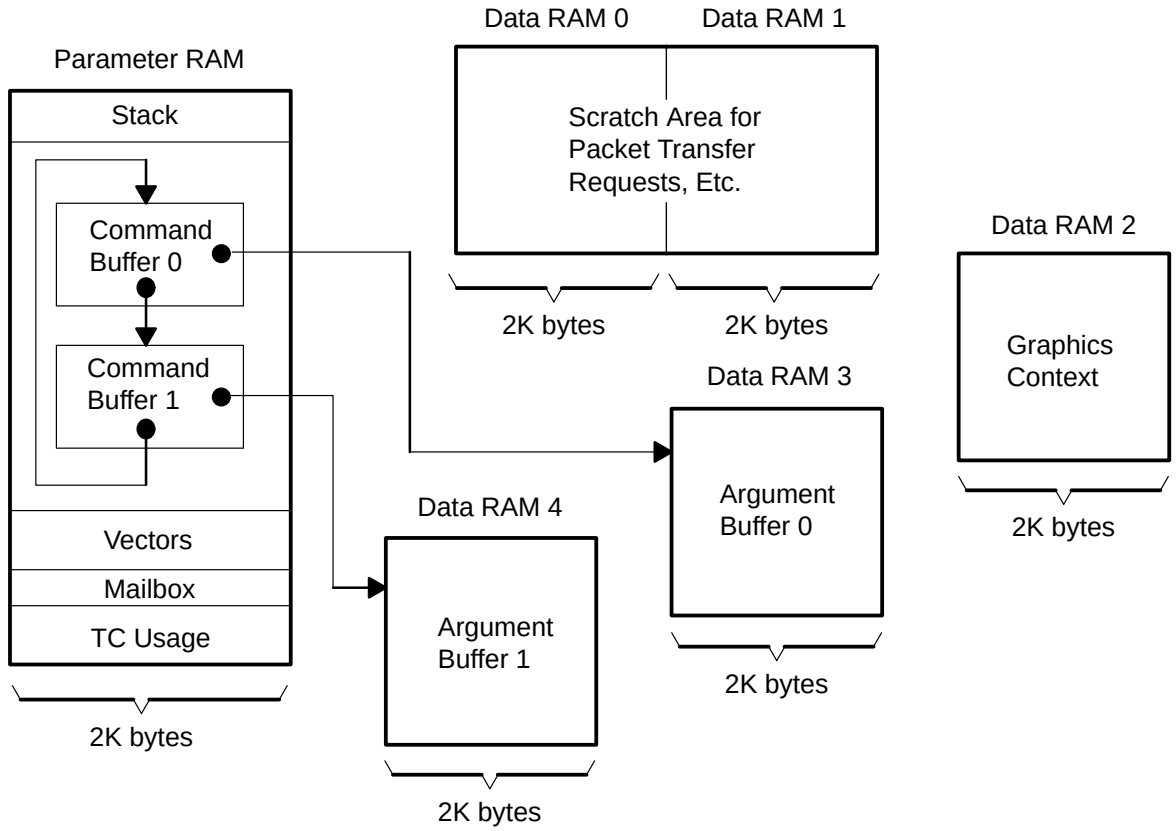
The PPs are treated as shared processing resources by the various tasks that execute on the MP. The sharing of the PPs among these tasks is managed by a resource manager task, which also runs on the MP. When ownership of a PP is transferred from some MP-resident task A to another MP-resident task B, task A must save the state of the PP so that it can resume its use of the PP at a later time. Any state information left in the PP's local (on-chip) RAM is likely to be overwritten when task B assumes control of the PP. The overhead of saving and restoring the state of a PP increases with the amount of data necessary to represent the state.

Between commands, a server PP typically maintains within its local RAM certain state information. This state information may define the context within which commands are executed to perform graphics or imaging operations. For example, in a graphics application, a server PP may need to keep track of parameters such as the current line width, line style, color, area-fill pattern, and raster-op. If, at the beginning of a new command, this information is not available in the PP's local RAM, it must be copied from the off-chip memory. Depending on how frequently this occurs and the amount of data to be transferred, the startup overhead for each new command could be considerable. To minimize this overhead, as much of this data as possible is preserved in the local RAM between commands. The more state information a server PP maintains in its local RAM, however, the more data it has to save off-chip in the event that the PP changes hands from one MP task to another.

## 5.4 An Example Configuration

Figure 5–5 shows an example configuration of the PP's local RAM. This example is for an application that uses only one of the MVP's on-chip PPs. Associated with the single PP in this configuration are a 2K-byte parameter RAM module and five 2K-byte data RAM modules. Note that software written to run on this type of configuration can execute on PP0 of a four-PP MVP device; for example, PP1 can be turned off so that two of PP1's data RAMs can be used as argument buffers by PP0. These two RAMs are identified as data RAMs 3 and 4 in Figure 5–5.

Figure 5–5. Example Configuration of the PP's Local RAM




In this example, the PP operates as a low-level graphics server. One or more MP tasks issue commands to the PP to draw high-level graphics primitives such as lines, polygons, and text. The MP delegates to the PP all responsibility for actually drawing the shapes into the frame buffer.

A portion of the PP's parameter RAM is dedicated to hardware functions such as interrupt vectors and data buffers for transfers performed by the MVP's transfer controller. In this example, the PP's system stack is placed at the high end of the parameter RAM and grows toward lower addresses. The parameter RAM also contains two command buffers and a mailbox.

Each command buffer contains a pointer to an argument buffer. In this application, each argument buffer is permanently associated with a command buffer. Separate data RAMs are dedicated to argument buffers 0 and 1 in Figure 5–5. The advantage of using separate RAMs is that the MP and the PP can share the argument buffers in ping-pong (double buffered) fashion to avoid memory-access contention. While the PP is reading the data for one command from one of the argument buffers, the MP can simultaneously load data for the next command into the other argument buffer.

For similar reasons, two separate data RAMs are available for use as scratch memory by the PP, which uses them to construct the packet transfer requests that it issues to the transfer controller (TC). These are the transfers that actually draw the shapes to the frame buffer. While the TC is servicing a packet transfer request contained in one of the two RAMs, the PP can be constructing the next request in the other RAM. The benefit of this arrangement is that the PP and TC can access their respective RAMs in parallel without mutual interference.

One additional data RAM is available to contain the graphics context, look-up tables, and other reference data. The graphics context consists of graphics state information that the command interpreter must maintain from one command to the next. It includes data such as the foreground and background colors, line width and style, and so on. Certain tables may need to be constructed during the execution of a command. These can be placed either in data RAM 2 next to the graphics context, or on the stack in the parameter RAM. The freedom to place look-up tables in either of these RAMs improves the likelihood of exploiting the full parallelism inherent in the PP's 64-bit instructions without encountering unavoidable access conflicts between the PP's global and local memory ports.

To increase parallelism between successive commands, the PP may begin executing a new command even before a packet transfer initiated by the previous command has been completed by the TC. To support this type of parallel operation of the PP and TC, the PP-resident routine that executes a command must adhere to the following convention: Each time the PP begins executing a new command, it must avoid modifying the buffer area that contains the parameters for a packet transfer initiated by the previous command until that packet transfer has completed. Depending on the application, this buffer area can consist of one or both of data RAMs 0 and 1, shown at the top of Figure 5–5.

For some applications, performance may be improved slightly by moving the PP's mailbox and command buffers from its parameter RAM to one of its data RAMs. Recall that in the event of a packet transfer from one area of external (off-chip) memory to another, the TC temporarily buffers data in a 128-byte region of the parameter RAM. If the TC is accessing this region of the parameter RAM at the same time that the PP is spinning on a mailbox or command buffer in the same parameter RAM, the resulting access conflicts can delay the TC. Moving the mailbox and command buffers to data RAM 2 in Figure 5–5, for example, avoids such potential delays.

## 5.5 MP's Command-Interface Library

A small library of functions provide MP-resident programs with the capability to communicate with PPs through the command interface. These functions perform all the operations necessary to manage the PP command interface through the command buffer, argument buffer, and mailbox structures that are described in Sections 5.1 through 5.4. The functions are summarized in Table 5–1.

Table 5–1. The MP's Command-Interface Library

| Function Name | Description | See Page |
|---|---|---|
| PpCmdBufBusy | Check Whether the PP's Command Buffer Is Busy | EX:5-20 |
| PpCmdBufGetArgs | Get the Argument Pointer From a Command Buffer | EX:5-21 |
| PpCmdBufInit | Initialize the Command Interface to a PP | EX:5-22 |
| PpCmdBufIssue | Issue a Command to a PP | EX:5-23 |
| PpCmdBufNext | Get the PP's Next Command Buffer | EX:5-24 |
| PpCmdBufNotifyIssue | Issue a Command With Notification | EX:5-25 |
| PpCmdBufSetArgs | Set the Argument Pointer in a Command Buffer | EX:5-26 |
| PpCmdBufSetFunc | Set the Function Pointer in a Command Buffer | EX:5-27 |
| PpCmdReadMbox | Read a PP's Mailbox | EX:5-28 |
| PpCmdWriteMbox | Write to a PP's Mailbox | EX:5-29 |

The remainder of this chapter presents detailed descriptions of each of the functions in the MP's command-interface library. Arguments, return values, and syntax are specified for each function. The functions are listed in alphabetical order.

**Syntax**

long PpCmdBufBusy(void **cmdBuf*);

**Description**

The *PpCmdBufBusy* function checks to see whether the specified PP command buffer is busy. Once the MP has issued a command buffer to a server PP, the command buffer is busy until the PP has finished processing the command. The PP indicates that it has finished by clearing the command buffer's flag.

Argument *cmdBuf* is a pointer to the command buffer.

The function returns a value of 0 if the command buffer is no longer busy. Otherwise, the function returns a nonzero value and requests notification from the PP (by means of a message interrupt) when the command buffer is no longer busy.

Before calling this function, the MP program should install an interrupt service routine to respond to the message interrupt from the PP.

When the function indicates that the command buffer is busy, the PP will always send a message interrupt to the MP as soon as the command buffer is no longer busy. The PP may occasionally send a message interrupt even when the command buffer is not busy. While this occurs too rarely to affect overall performance, the MP program must be able to handle the extra interrupt correctly. When the MP program is expecting a PP message interrupt and an interrupt arrives from the PP, the MP should call *PpCmdBufBusy* to verify that the command buffer is no longer busy. If the function call indicates that the command buffer is still busy, the MP should await the next message interrupt from the PP and then repeat the call to *PpCmdBufBusy*.

**Syntax**

void *PpCmdBufGetArgs(void **cmdBuf*);

**Description**

The *PpCmdBufGetArgs* function returns the argument pointer value from the specified command buffer. This is a pointer to a buffer containing the argument data that the server PP requires to process the command.

Argument *cmdBuf* is a pointer to the command buffer into which the function pointer is to be loaded.

**Syntax**

void *PpCmdBufInit(long *ppNum*, void **function*,
long *numCmdBufs*);

**Description**

The *PpCmdBufInit* function initializes the command interface to a specified PP. The function verifies that the PP is halted. Next it initializes the data structures in the PP's parameter RAM that the MP uses to communicate with the PP. These include the mailbox and a specified number of circularly linked command buffers. Finally, the function unhalts the PP so that it will be ready to begin accepting commands.

Argument *ppNum* is the PP number. This is a value in the range 0 through 7. The function ignores all but the three LSBs of the argument's value.

Argument *function* is a pointer to the PP command interpreter program. This is the address at which the PP begins executing when it is unhalted.

Argument *numCmdBufs* is the number of command buffers that are to be set up in the PP's parameter RAM. A typical value is 2.

The function returns a pointer to the first command buffer. This is the buffer in which the MP should place its initial command to the PP.

If the target PP is not halted at the time of the call, the function takes no action and returns a null pointer.

**Syntax**

void PpCmdBufIssue(void **cmdBuf*);

**Description**

The *PpCmdBufIssue* function issues a command to a PP. The function sets the command buffer's flag to 1 to indicate that the command in the buffer is ready to be executed.

Argument *cmdBuf* is a pointer to the command buffer that contains the command that is to be issued.

**Syntax**

void *PpCmdBufNext(void **cmdBuf*);

**Description**

The *PpCmdBufNext* function advances the command buffer pointer to the PP's next command buffer and returns a pointer to the command buffer. The command buffers for a particular PP form a circularly linked list; each command buffer contains a pointer to the next command buffer in the list.

Argument *cmdBuf* is a pointer to the current command buffer.

**Syntax**

void PpCmdBufNotifyIssue(void **cmdBuf*);

**Description**

The *PpCmdBufNotifyIssue* function issues a command buffer to a server PP and requests notification from the PP when it completes the command. The PP notifies the calling task by sending a message interrupt to the MP.

Argument *cmdBuf* is a pointer to the command buffer. The function sets the command buffer's flag to indicate that the buffer contains a valid command that is ready to be executed by the server PP.

The *PpCmdBufNotifyIssue* function differs from the *PpCmdBufIssue* function, which issues a command but does not request notification from the PP. The *PpCmdBufIssue* function is typically used in conjunction with the *PpCmdBufBusy* function, which requests notification only in the event that the PP is still processing the command at the time of the call.

Before calling this routine, the MP program should install an ISR to respond to the message interrupt from the PP.

**Syntax**

Syntax: void PpCmdBufSetArgs(void **cmdBuf*,
void **argBuf*);

**Description**

The *PpCmdBufSetArgs* function loads the argument pointer into the specified command buffer. This is a pointer to a buffer containing the argument data that the server PP requires to process the command.

Argument *cmdBuf* is the pointer to the command buffer into which the argument pointer is to be loaded.

Argument *argBuf* is the pointer to the argument buffer.

**Syntax**

void PpCmdBufSetFunc(void **cmdBuf*, void ***function*);

**Description**

The *PpCmdBufSetFunc* function loads a function pointer into the specified command buffer. The function pointed to by this value is the function the PP executes to process the command.

Argument *cmdBuf* is a pointer to the command buffer into which the function pointer is to be loaded.

Argument *function* is the function pointer that is to be loaded into the command buffer.

**Syntax**

long PpCmdReadMbox(long *ppNum*);

**Description**

The *PpCmdReadMbox* function returns the value contained in the specified PP's mailbox.

Argument *ppNum* is the PP number. This is a value in the range 0 to 7. The function ignores all but the three LSBs of the argument value.

This function can be called from an interrupt service routine to read a PP's mailbox.

**Syntax**

void PpCmdWriteMbox(long *ppNum*, long *value*);

**Description**

The *PpCmdWriteMbox* function writes a value to the designated PP's mailbox.

Argument *ppNum* is the PP number. This is a value in the range 0 to 7. The function ignores all but the three LSBs of the argument value.

Argument *value* is the value to be written to the mailbox.

This function can be called from an interrupt service routine to write a value to a PP's mailbox.

Appendix A

# Cache Coherency

The MVP's master processor (MP) contains a 4K-byte set-associative data cache in which the MP buffers data copied from the MVP's external memory. This cache is designed to be transparent to software running in a single-processor environment.

The cache is not entirely transparent in a tightly coupled multiprocessor environment, however. In particular, consider a system in which the MVP communicates with a host processor through a shared block of RAM that is directly accessible to both. Because the host processor communicates with the MVP through message buffers in the MVP's external memory, the low-level communications software must comprehend the operation of the MVP MP's data cache to avoid causing cache coherency errors.

The following sections explain how to use the MVP's hardware mechanisms to ensure cache coherency:

**Topics**

# A.1 Operation of the MP’s Data Cache

At some instant, the master processor’s (MP’s) data cache may contain a copy of a shared message buffer in memory. If the host processor writes to this buffer, the copy contained in the cache is no longer valid. An error occurs if the MP reads the stale data in its cache.

Similarly, if the MP attempts to write to a message buffer in its external memory, the write is intercepted by the cache and only the cache is immediately updated. The modified data in the cache is marked as dirty, and it may not be written back to memory for some time. In the meantime, the copy contained in the memory is not valid. An error occurs if the host reads the stale data in memory.

The MP’s data cache is partitioned into sixteen 256-byte blocks, each of which can be mapped to a cache-aligned 256-byte block of memory. Each block, in turn, is partitioned into four 64-byte subblocks. A subblock represents the minimum amount of data that can be transferred between the cache and the memory at one time. See Section 6.4, *Cache Architecture,* in the *MVP Master Processor User’s Guide*.

Associated with each cache subblock is a **present flag** and a **dirty flag** (see Figure 3–6, *MP Cache Tag Registers,* on page MP:3-22 of the *MVP Master Processor User’s Guide*). The present flag indicates whether the cache subblock contains a current copy of an area of memory. The dirty flag is set to indicate that the MP has modified the data in the cache subblock and the subblock has not yet been written back to memory. When an MP write to memory is intercepted by the cache, the subblock containing the modified data is marked as dirty. A dirty subblock is eventually written back to memory, but the writeback is delayed until the subblock either is replaced or is explicitly flushed.

## A.2 Flushing a Cache Subblock

The MP communications software uses the MP's *dcachef* instruction to explicitly flush a cache subblock (see subsection 10.7.3, *Cache Flushes,* in the *MVP Master Processor User's Guide)*. If the subblock contains dirty data, this instruction forces the subblock to be written back to memory. The instruction then clears both the present flag and the dirty flag associated with the subblock. At this point, the subblock in memory is up-to-date and the copy of the subblock in the cache has been deleted.

Once the cache subblock has been flushed, the host processor may read from or write to the corresponding subblock in memory. During the time that the host accesses this subblock, the MP must not attempt to write to or read from the subblock.

Once the host has loaded the memory subblock with new message data for the MP, it informs the MP that it may now access the subblock. The MP's attempt to read any word in the subblock causes the entire subblock to be copied to the data cache. During the time that the MP accesses the subblock, the host must not attempt to write to or read from the subblock.

## A.3 Message-Buffer Placement

The *dcachef* instruction serves as the basis for a communication protocol that ensures cache coherency by enforcing mutually exclusive access to a message buffer shared between the host and the MP.

The buffer for any message passed between the MVP and an external processor spans an integral number of subblocks in memory. Even though the message may not fully occupy the subblock at the beginning or end of the message, the entire subblock must still be reserved for the message. An attempt to use the leftover portion of the subblock to buffer another message, for example, would compromise the MP's cache-coherency mechanisms.

Once you have allocated a message buffer according to these rules, you can initialize the buffer by calling the kernel's *TaskInitMsg* function. Note that a message allocated by the *TaskAllocMsg* function does not follow these rules and should be used only for messages sent between tasks on the same processor.

Appendix B

# Task Error Codes

A task error is an error that occurs within the context of a particular task during a call to a kernel function. When one of the kernel functions returns a value that indicates that it has failed, the calling program can interrogate the kernel about the error by calling *TaskGetLastError*, which returns a task error code that specifies the precise cause of the failure. The *TaskGetLastError* function always returns the code of the last task error to occur within the task. If no task error has occurred since the task was created, the function returns a value of 0.

Table B–1 lists the task error codes defined in the C include file *task.h*, which is included in the multitasking executive release package. To enhance the portability of your application programs, use the symbols in the table to refer to task errors rather than hardcoded numerical values.

Table B–1. Task Error Codes

| Symbol for Task Error Code | Meaning |
|---|---|
| TASK_ERROR_BADALLOC | The specified message allocation information field is not valid. |
| TASK_ERROR_BADMSG | The specified message pointer is not valid, or the message it points to has been corrupted. |
| TASK_ERROR_BADNODENUM | The specified processor node number is not valid. |
| TASK_ERROR_BADPORTID | The specified port ID is not valid. |
| TASK_ERROR_BADPRIVIX | The specified private-data index not valid. |
| TASK_ERROR_BADRESID | The specified resource ID is not valid. |
| TASK_ERROR_BADSEMAID | The specified semaphore ID is not valid. |
| TASK_ERROR_BADTASKID | The specified task ID is not valid. |
| TASK_ERROR_CLOSEPORT | The task was waiting on a port at the time that the port was closed. |
| TASK_ERROR_CLOSESEMA | The task was waiting on a semaphore at the time that the semaphore was freed. |
| TASK_ERROR_FLAGBOUND | The task attempted to bind a port or semaphore to an event flag that is already bound. |
| TASK_ERROR_LISTFULL | The specified function cannot be added to either the init-list or exit-list because the list is full. |
| TASK_ERROR_MSGSIZE | The requested message size is outside the legal range 0 to 16 384. |
| TASK_ERROR_OUTOFMEM | The function cannot allocate the amount of memory needed for the specified kernel resource. |
| TASK_ERROR_OUTOFPRIV | The function cannot allocate the requested private-data word. |
| TASK_ERROR_NOBINDPORT | The task attempted to unbind a port that is not bound. |
| TASK_ERROR_NOBINDSEMA | The task attempted to unbind a semaphore that is not bound. |
| TASK_ERROR_PORTBOUND | The task attempted to bind one of its event flags to a port that is already bound. |
| TASK_ERROR_RESTBLFULL | An ID cannot be assigned to the specified resource because the kernel's resource table is full. |
| TASK_ERROR_SEMABOUND | The task attempted to bind one of its event flags to a semaphore that is already bound. |
| TASK_ERROR_SUSPDEF | The task attempted to suspend the default task. |
| TASK_ERROR_WAITDEF | The default task attempted to wait. |
| TASK_ERROR_WAITFREE | The task attempted to wait on an event flag that is free (not bound to a port or semaphore). |

Appendix C

# Glossary

## A

**allocation node:** The processor node into which an internode message is allocated.

**argument buffer:** The memory block into which the argument values that accompany a command to a server PP are placed.

**assembler:** A software utility that creates a machine-language program from a source file. There are two assemblers associated with the MVP: a mnemonic-based RISC-type assembler for the MP and an algebraic assembler for the PP.

## B

**binding:** Associating or linking together two complementary software objects.

**blocked:** The state of a task that is not ready to execute. A task can be blocked either by being suspended or by voluntarily choosing to wait for an event such as the arrival of a message or signal.

**buffer pool:** See *message buffer pool*

# C

**cache:** A fast memory into which frequently used data or instructions from slower memory are copied for fast access. Fast access is facilitated by the cache's high speed and its on-chip proximity to the CPU.

**cache clean:** An MP instruction that updates external memory by writing modified (dirty) data-cache subblocks back to memory, thus resetting that subblock's dirty bit to 0.

**cache coherency:** The state or condition in which the contents of one or more cache memories consistently and accurately represent the corresponding contents of the external memory.

**cache flush:** An MP instruction that updates external memory by writing modified (dirty) data-cache subblocks back to memory, thus resetting that subblock's present and dirty bits to 0.

**clean:** See *cache clean*

**client:** A program or task that requests services from a server program or task.

**coherency:** See *cache coherency*

**command interpreter:** A software routine that accepts commands from a client program or task and dispatches each command to the appropriate subroutine for execution.

**counting semaphore:** See *semaphore*

**crossbar:** A generally configurable, high-speed bus switching network for a multiprocessor system, permitting any of several processors to connect to any of several memory modules.

# D

**data cache:** The MP's two SRAM banks that hold cached data needed by the MP. Data RAMs for the PPs are not cached.

**data RAM:** On-chip RAM that is available for the general-purpose storage of data by the MP or PPs on the MVP.

**debugger:** A window-oriented software interface that helps you to debug MVP programs running on an MVP emulator or simulator.

**default task:** The task that runs when no other task is ready to run.

**destination port:** The message port to which a message is sent. See also *port ID*

**dirty flag:** A storage bit associated with each subblock of MP data-cache memory that indicates whether the subblock contains modified data that needs to be written back to main memory. See also *cache flush*, *cache clean*

**doubleword:** A 64-bit value.

## E

**event:** A stimulus that can cause a task to begin executing.

**event flag:** A bit in memory that is bound to either a port or a semaphore and that indicates whether a specific event has taken place.

**event register:** A field in a task descriptor that contains the associated task's 32 event flags.

**exception:** A condition that is handled outside the normal program flow of a task. An exception in a task is the software equivalent of a hardware interrupt in a processor.

**exception flag:** A bit in a task's exception register that indicates the status of a particular type of exception.

**exception handler:** A function associated with a task that handles exceptions raised in that task. The manner in which an exception handler responds to an exception in a task is analogous to the way an interrupt service routine (ISR) responds to a hardware interrupt in a processor.

**exception register:** A field in a task descriptor that contains the associated task's 32 exception flags.

**executive:** The portion of a multitasking software system that is responsible for executing application tasks, providing communications among tasks, and managing shared resources.

**exit-list:** A list of functions that are executed by the kernel each time a task exits.

**external address:** See *off-chip address*

## F

**flush:** See *cache flush*

**foreign node:** A processor node other than the one in which the executable code for a particular task resides. This term is relative. A task that executes on a particular node n considers this node its local node, but tasks that execute on other nodes consider node n to be a foreign node.

## H

**halfword:** A 16-bit value.

**heap:** A large pool of memory that can be allocated at runtime by application programs and by the executive.

## I

**ID:** *Identifier*. A 32-bit field that contains a 16-bit resource-table index, an 8-bit sequence number, and an 8-bit resource-type code and that identifies a kernel resource such as a port, semaphore, or task.

**init-list:** A list of functions that are executed by the kernel each time a new task is created.

**internal address:** See *on-chip address*

**internode message manager:** A special task that manages an interface with another processor node to facilitate message transfers across the interface. Routing of messages across internode boundaries is one of the system services provided by the MVP executive.

**ISR:** *Interrupt service routine*. A module of code that is executed in response to a hardware or software interrupt.

## K

**kernel:** A set of low-level software primitives within the MVP multitasking executive that implement multitasking, the passing of messages and signals, and the monitoring of multiple events.

**kernel resource:** A data item or object created by and used within the kernel. The kernel defines three types of resources: port, semaphore, and task. See also *port*, *semaphore*, *task*

## L

**linker:** A software tool that combines object files to form an object module that can be allocated into system memory and executed by the device.

**local node:** The local processor node. This term is relative. A task that executes on a particular node n considers node n its local node, but tasks that execute on other nodes consider node *n* to be a foreign node.

**local RAM:** The on-chip RAM that is associated with a particular PP in an MVP.

**LSB:** *Least significant bit.* The bit having the smallest effect on the value of a binary numeral, usually the rightmost bit. The MVP numbers the bits in a word from 0 to 31, where bit 0 is the LSB.

## M

**mailbox:** A 32-bit word in a PP's parameter RAM in which it places messages to clients (chiefly, the MP).

**master processor:** See *MP*

**message buffer pool:** A group of message buffers, typically of uniform size, that belong to a particular task. See also *reclamation port*

**message event:** The event caused by the arrival of a message at a port. If a task is waiting for the event, that task is scheduled to begin executing.

**message header:** A fixed-size structure at the beginning of a message buffer that contains the information that describes the message, its destination port, and so on, to the kernel functions that handle the message.

**message interrupt:** An MVP hardware mechanism through which one on-chip processor can signal an interrupt to another on-chip processor, if that interrupt is enabled.

**message port:** See *port*

**message-routing table:** A table that specifies the routing ports for messages sent to foreign processor nodes.

**MP:** *Master processor*. A general-purpose RISC processor that coordinates the activity of the other processors on the MVP. The MP includes an IEEE-754 floating-point hardware unit.

**MSB:** *Most significant bit*. The bit having the greatest effect on the value of a binary numeral. It is the leftmost bit. The MVP numbers the bits in a word from 0 to 31, where bit 31 is the MSB.

**multimedia video processor:** See *MVP*

**MVP:** *Multimedia video processor*. A single-chip multiprocessor device that accelerates applications such as video compression and decompression, image processing, and graphics. The multimedia video processor contains a master processor and from one to eight parallel processors, depending on the device version. For example, the TMS320C80 device contains four PPs.

**MVP multitasking executive:** See *executive*

## N

**node:** Any entity capable of sending and receiving messages, such as a processor port.

**node-global port ID:** A port ID that contains an explicit node number in the range 0 to 127. A node-global destination port ID is required to route a message across processor node boundaries in a multiprocessor system.

**node-local port ID:** A port ID that does not contain an explicit processor node number but that is implicitly local. The resource-type code in the eight MSBs of a node-local port ID is set to a value of –1.

**node number:** A small nonnegative integer that uniquely identifies each processor node in a multiprocessor system. A node-global port ID contains a node number in the range 0 to 127.

## O

**off-chip address:** An address external to the MVP chip. Addresses from 0x0200 0000 to 0xFFFF FFFF are off-chip addresses. See also *on-chip address*

**on-chip address:** An address internal to the MVP chip. Addresses from 0x0000 0000 to 0x1FFF FFFF are on-chip addresses. See also *off-chip address*

# P

**packet:** A collection of patches of data.

**packet transfer:** See *PT*

**parallel processor:** See *PP*

**parameter RAM:** A general-purpose 2K-byte RAM that is associated with a specific processor, part of which is dedicated to packet transfer information and the processor interrupt vectors.

**patch:** A group of lines of equal length whose starting addresses are an equal distance apart.

**periodic event:** An event that repeats at regular intervals, as opposed to an event that either occurs only once or repeats at irregular intervals.

**pool:** See *message buffer pool*

**port:** A type of shared data object that contains a queue of messages.

**port ID:** A 32-bit value the kernel uses to identify a port it has opened. See also *ID*

**port table:** A fixed-size table of pointers that the kernel uses to keep track of the ports it has opened.

**PP:** *Parallel processor*. The MVP's advanced digital signal processor that is used for video compression/decompression (P $\times$ 64 or MPEG), still-image compression/decompression (JPEG), 2-D and 3-D graphic functions such as line draw, trapezoid fill, antialiasing, and a variety of high-speed integer operations on image data. An MVP single-chip multiprocessor device may contain from one to eight PPs, depending on the device version.

**PP command interface:** The software interface through which the MP (or other client processor) issues commands to be executed by a server PP.

**preempt:** To interrupt the processing of a task to allow a more urgent task to begin executing.

**present flag:** A bit in the cache tag register associated with a cache subblock that indicates whether the information in the subblock is present in the cache.

**private context:** A portion of a task's context that is private to a particular software library or group of related functions and is hidden to the main task program and to the other libraries used by that program.

**processor node:** In a multiprocessor system, a processor that executes the MVP executive and that is able to communicate with the other processors through the executive's internode message-passing mechanisms.

**PT:** *Packet transfer*. A transfer of data blocks between two areas of memory. The MVP supports packet transfers of one, two, or three dimensions.

## R

**raster-op:** A raster operation, which is an arithmetic or logical combination of the source and destination data that takes place during the transfer of a pixel array from one location to another.

**ready:** A task state indicating that the task either is currently executing or is able to execute as soon as it acquires the processor.

**ready queue:** A linked list that contains the descriptors for all tasks that are in the READY state.

**real-time system:** A system in which each processing job is completed by a specified deadline. A real-time system is characterized not so much by its best-case speeds as by its worst-case delays, which must be specified to guarantee job completion by a given time.

**reclamation port:** The message port to which a message buffer is returned when the message is discarded. The buffer containing the message is then available to send another message. See also *port ID*

**reentrant code:** Program code that can be executed concurrently by more than one task. Reentrant code contains no task-specific data.

**remote procedure call:** See *RPC*

**reply message:** A message sent by a server in reply to a request message from a client.

**reply port:** The message port to which a reply to a request message is to be sent. See also *port ID*

**request message:** A message sent by a client to request service from a server.

**resource:** See *kernel resource*

**resource ID:** A 32-bit value the kernel uses to identify a port, semaphore, or task it has opened or created. See also *ID*

**resource table:** A fixed-size table of pointers that the kernel uses to keep track of the internal resources it has opened or created. See also *kernel resource*

**resume:** To make a suspended task ready to execute again.

**round-robin scheduling:** A method for scheduling the execution of a set of tasks of equal priority. The tasks share the processor on a rotating basis. Each of the tasks takes its turn executing for a time interval that is roughly equal for all tasks.

**routing port:** In a multiprocessor system, a local port through which the kernel of the MVP executive routes messages that are destined for a particular foreign processor node. See also *message-routing table*

**RPC:** *Remote procedure call*. A procedure (or function) call that is executed on a processor other than the one on which the call originated. See also *stub routine*

## S

**scheduler:** See *task scheduler*

**scheduling:** The strategy used in an operating system to share a resource such as a processor or memory.

**semaphore:** A type of shared data object that contains an integer count and is used to provide mutual exclusion.

**semaphore ID:** A 32-bit value that the kernel uses to identify a semaphore that it has opened. See also *ID*

**semaphore table:** A fixed-size table of pointers that the kernel uses to keep track of the semaphores that it has opened.

**server:** A program or task that provides services to a client program or task.

**signal:** To increment the count at a semaphore. See also *semaphore*

**signal event:** The event caused by the arrival of a signal at a semaphore. If a task is waiting for the event, that task is scheduled to begin executing.

**single-threaded:** A method of programming in which only a single thread or program element executes on a processor at any one time. This is in contrast to a multitasking system in which multiple tasks run concurrently on a single processor.

**spinning:** A technique for continuous monitoring of a status flag in a register or memory location in which the processor repeatedly polls the flag until the awaited status change occurs.

**state:** See *task state*

**stub routine:** A routine that is called from an applications program by means of a standard function call but that does not itself execute the specified function. See also *RPC*

**suspend:** To halt further execution of a task indefinitely or until another task instructs the kernel to resume the suspended task.

## T

**task:** A program element that executes concurrently with other program elements in a multitasking system.

**task argument:** The pointer value that is passed to a task as an argument at the time that the kernel begins executing the task.

**task descriptor:** A data structure that specifies a task's state, priority, function pointer, argument, and event flags. A task descriptor is allocated for each task at the time that the task is created.

**task error:** An error that occurs within a task during a call to a kernel function. In contrast to a system error that can affect the entire system, a task error has no direct effect on tasks other than the task in which the error occurred.

**task function:** The program code for a task, which is written in the form of a standard C function. The task scheduler in the executive's kernel begins executing a task by making a standard C function call to the task function.

**task ID:** A 32-bit value that the kernel uses to identify a task that it has created. See also *ID*

**task priority:** A number assigned to a task to indicate its relative urgency. The kernel of the MVP executive allows you to assign a priority number in the range 0 to 31 to a task. Larger numbers represent higher (more urgent) priorities.

**task scheduler:** The portion of the kernel that is responsible for determining the order in which the tasks in the ready queue are executed.

**task state:** The state of an MP-resident task running under the kernel of the MVP executive. A task can be in one of four states: READY, WAITING, SUSPENDED, and WAITSUSPEND.

**task table:** A fixed-size table of pointers that the kernel uses to keep track of the tasks it has created.

**TC:** *Transfer controller*. The MVP's on-chip DMA controller for servicing the cache and for transferring one-, two-, and three-dimensional data blocks between each processor on the MVP and its external memory.

**thread of execution:** A schedulable unit of execution in a multitasking system. The term refers specifically to the progressive execution of a program element; it excludes other attributes, such as the system resources allocated to a task or process.

**tightly coupled multiprocessor:** A multiprocessor system in which the processors communicate with each other through shared memory, as contrasted with a loosely coupled multiprocessor system in which processors communicate over a network.

**time slicing:** A method of multitasking in which each task is allotted a maximum amount of execution time, referred to as a time slice or quantum, before the next task is allowed to run.

**transfer controller:** See *TC*

## V

**VC:** *Video controller*. The portion of the MVP responsible for the video interface.

**video controller:** See *VC*

## W

**wait and signal:** See *semaphore*

**wait queue:** A queue of tasks waiting at a port for messages or waiting at a semaphore for signals.

**waiting:** A state in which a task has voluntarily blocked itself from further execution until an awaited event occurs. The kernel allows a task to wait either for the arrival of a message at a port or for the arrival of a signal at a semaphore.

**WAITING state:** See *task state*

**word:** A sequence of 32 adjacent bits that constitutes a register or memory value. The PP supports 32-bit words. The MP also supports doublewords of 64 bits for loads and stores.

# Index

## A



## B



## C

## D



## E



## F

## H



## I



## K

## L



## M



## N

## P



## Q



## R

## S



## T

## U

## V



## W



## Y

## IMPORTANT NOTICE

Texas Instruments (TI) reserves the right to make changes to its products or to discontinue any semiconductor product or service without notice, and advises its customers to obtain the latest version of relevant information to verify, before placing orders, that the information being relied on is current.

TI warrants performance of its semiconductor products and related software to the specifications applicable at the time of sale in accordance with TI's standard warranty. Testing and other quality control techniques are utilized to the extent TI deems necessary to support this warranty. Specific testing of all parameters of each device is not necessarily performed, except those mandated by government requirements.

Certain applications using semiconductor products may involve potential risks of death, personal injury, or severe property or environmental damage ("Critical Applications").

TI SEMICONDUCTOR PRODUCTS ARE NOT DESIGNED, INTENDED, AUTHORIZED, OR WARRANTED TO BE SUITABLE FOR USE IN LIFE-SUPPORT APPLICATIONS, DEVICES OR SYSTEMS OR OTHER CRITICAL APPLICATIONS.

Inclusion of TI products in such applications is understood to be fully at the risk of the customer. Use of TI products in such applications requires the written approval of an appropriate TI officer. Questions concerning potential risk applications should be directed to TI through a local SC sales office.

In order to minimize risks associated with the customer's applications, adequate design and operating safeguards should be provided by the customer to minimize inherent or procedural hazards.

TI assumes no liability for applications assistance, customer product design, software performance, or infringement of patents or services described herein. Nor does TI warrant or represent that any license, either express or implied, is granted under any patent right, copyright, mask work right, or other intellectual property right of TI covering or relating to any combination, machine, or process in which such semiconductor products or services might be or are used.