3DO System Programmer's Guide

This book contains descriptions of the system level components that make up Portfolio: kernel, I/O, tasks, filesystem, events, and so on. It is written for title developers who create programs that run on a 3DO system. To use this document, the you should have a working knowledge of the C programming language and object-oriented concepts.

This book contains the following chapters:

Understanding the Kernel

This chapter introduces you to the role of the kernel in the 3DO operating system. It contains the following topics:

Description of the Kernel

The kernel is a folio of function calls that is the heart of the Portfolio operating system. It controls 3DO hardware and any software running on the system. It manages resources and provides communication between running tasks and hardware devices. In particular, the kernel handles these responsibilities:

- **Multitasking.** The kernel handles the execution of all programs (tasks) running on the system and provides multitasking so that many tasks can run simultaneously.

- **Memory management.** The kernel allocates memory to all running tasks and makes sure that one task doesn't indiscriminately write to the memory allocated to another task. The kernel also consolidates free memory, manages the system's own memory, and performs other memory management duties.

- **Folio management.** Folios are the mechanism used by the kernel to bundle related functionality. Portfolio is composed of many folios, which together provide the system API. The kernel manages the creation, disposal, loading, and unloading of folios.

- **Intertask communication.** The kernel enables multiple tasks running simultaneously to communicate with one another. The two primary methods of intertask communication are by sending high-performance Boolean signals, or by sending messages.

- **Resource sharing.** The kernel provides a check-in/check-out system for critical resources that tasks share so that only one task at a time works with those resources.

- **Linked lists.** The kernel creates and manages linked lists. Each linked list is an ordered group of data structures that can grow, shrink, or change order as necessary.

- **Error codes and messages.** The kernel provides a consistent way to handle error returns throughout the system. It helps convert error codes into descriptive strings to make application development easier.

- **I/O.** The kernel provides synchronous and asynchronous communication with I/O devices, and includes device and driver definitions for those devices.

Multitasking

One of the most important jobs of the kernel is managing the tasks that run on a 3DO system. Because Portfolio supports preemptive multitasking, the kernel must provide conventions for deciding which tasks get CPU time, and for saving a task's state when execution switches from one task to another.

Privileged and Non-Privileged Tasks

The kernel makes an important distinction between tasks running on a 3DO system, and divides them into two categories: *privileged tasks* and *non-privileged tasks.* Privileged tasks have special rights that non-privileged tasks do not have. Tasks that you create are always non-privileged. Only special tasks created by a 3DO system can be privileged.

Multitasking

The kernel supports preemptive context switching for multitasking. It normally devotes one time quantum (normally 15 milliseconds) of CPU time to a task and then switches execution to another task, where it devotes another time quantum before switching to yet another task. To make the switch without destroying the current state of the executing task, the kernel saves the context of the current task in the task's TCB (task control block, a data structure that contains the parameters of each task), and reads the context of the next task from its control block before executing that task. (A task's context includes states such as its allocated address spaces and its register set.)

At any point during a time quantum, the kernel can preempt the current task, and immediately switch execution to a more important task if necessary. To determine how and when to switch from one task to another, the kernel reads task states and priorities.

Task States

A task can be in one of three states:

- **Running:** The task is currently executing in the current time quantum.
- **Ready to run:** The task is stored in the *ready queue,* awaiting execution by the kernel in a future time quantum.
- **Waiting:** The task is stored in the *waiting queue,* waiting for an external event (such as a vertical blank or an I/O request) to occur. Once the task is notified of the event's occurrence, the task

moves to the ready queue for execution.

The kernel executes tasks in the ready queue only, switching from task to task as required. It does not execute tasks in the waiting queue, so waiting tasks don't require any CPU cycles-a courtesy to the other tasks running on the system.

To determine how the ready-to-run tasks are executed, the kernel considers each task's priority.

Task Priorities

Each task has a priority that is associated with it. The priority is a value from 10 to 199: 10 is the lowest priority, 199 is the highest priority. This priority can be changed at any time to give the task a higher or lower priority than others in the ready queue-or to give the task an equal priority with other tasks.

Priority determines the order in which tasks in the ready queue are executed. The kernel executes only the highest-priority task (or tasks) in the ready queue and doesn't devote any CPU time to lower-priority tasks. If several tasks all share the same highest priority, the kernel rotates among those tasks, devoting one time quantum to each. Lower-priority tasks in the ready queue don't receive any CPU time at all until there are no higher-priority tasks in the ready queue: the higher-priority tasks either finish execution and exit the system, move to the wait queue, or change to a lower priority.

Whenever a new task with a higher priority than the one running enters the ready queue, or whenever an existing task is given a higher priority than the one running, the kernel preempts the running task. It immediately switches execution to the higher-priority task, even if the switch occurs in the middle of a time quantum. The higher-priority task then starts at the beginning of its own time quantum.

Note that round-robin scheduling takes place only when tasks of equal priority have the highest priority in the ready queue. If only one task has the highest priority, only that task runs, and all others languish until it finishes or is kicked out of the CPU limelight by another task with a higher priority. Note also that if a task executes a `Yield()` call, it forgoes the rest of its quantum, and yields the CPU immediately to other tasks of equal priority.

Waiting Tasks

To get into the wait queue, a task must execute a wait function call (such as `WaitSignal()`, `WaitIO()`, or `WaitPort()`) to define what it's waiting for. It then becomes a waiting task and receives no CPU time. When its wait conditions are satisfied, a task moves to the ready queue, where it can compete for CPU time.

The wait queue is an important feature for keeping running tasks working at top speed without, wasting CPU cycles on tasks waiting for external events. For the wait queue to work, each task must *not* use a

loop that constantly checks for an event. Repeatedly checking for an event, known as "busy-waiting," is greatly scorned in the Portfolio world-it eats up unnecessary CPU cycles and makes your task unpopular with other developers and users around the world.

Task Termination

As each task runs, it accumulates its own memory and set of resources. The kernel keeps track of the memory and resources allocated to each task and when that task quits or dies, the kernel automatically closes those resources and returns the memory to the free memory pool. This means that the task doesn't have to close resources and free memory on its own before it quits, a convenient feature. Good programming practice, however, is to close and release resources and free memory within a task whenever they are no longer in use. If a thread exits, it must free its memory so that the parent task can allocate it for its own use.

Parent Tasks, Child Tasks, and Threads

Any task can launch another task. The launched task then becomes the child of the launching task and the child task is a resource of the parent task. This means that when the parent task quits, all of its children quit too.

To sever the parent/child relationship between two tasks so that the child task doesn't quit with the parent task, the parent can use the `SetItemOwner()` function call to transfer ownership of the child task to the child task itself. When the parent task quits, the child task continues to run.

A parent task spawns child tasks to take care of real-time processing and other operations. A child task has one big disadvantage, it doesn't share memory with the parent. Because it must allocate its own memory in pages, it can waste memory if its memory requirements are small. And because parent and child don't share memory, they can't share values stored in shared data structures. To overcome these disadvantages, a parent task can spawn a *thread.*

A thread is a child task that shares the parent task's memory. The owner of the thread can transfer ownership of a thread to any thread in the parent's task family except to the thread itself. A task family is a task and all of its threads.

Managing Memory

A multitasking system must manage memory carefully so that one task won't write to another's memory. To do so, the kernel allocates exclusive memory to each task and restricts each task to writing to its own memory unless given specific permission to do otherwise.

Types of Memory

When allocating memory, the kernel must consider the two main types of memory available in a 3DO: DRAM (Dynamic Random Access Memory) and VRAM (Video Random Access Memory). DRAM is memory with a standard data bus for read/write operations. VRAM has the standard data bus, but also provides an additional bus named SPORT (short for Serial PORT), which can take streams of bits from the VRAM arrays and generate video signals.

VRAM is premium memory because it can be used for everything, including special video operations. Because the SPORT bus allows for very quick block memory transfers, VRAM is useful for manipulating graphics and other complex data. VRAM can also be used for any other standard memory operations. DRAM isn't as versatile as VRAM because it can't be used for all video operations, but works fine for standard operations. DRAM can keep data for the cel engine (described in the *Programming 3DO Graphics*), which projects graphic images into VRAM.

Memory Size

A minimum 3DO system includes 2 MB of DRAM and 1 MB of VRAM for a total of 3 MB of RAM. Optional memory configurations can, for this version of the hardware, go up to a maximum of 16 MB of RAM: 1 MB of VRAM and 15 MB of DRAM, or 2 MB of VRAM and 14 MB of DRAM.

Note: Although the current hardware implementation doesn't address more than 16 MB total, future implementations will go beyond that limit, so you must use full 32-bit addressing to ensure future compatibility.

When the kernel allocates memory to tasks, it keeps track of each page of memory it allocates. The size of a memory page is determined by the total amount of DRAM in the system divided by 64, the constant number of DRAM pages available, regardless of the amount of memory in the system. In a standard 3 MB system, each page of DRAM is 32 KB large (2 MB of DRAM divided by 64). In a system with 16 MB of DRAM, the page size is 256 KB.

VRAM page size is similar to DRAM page size. If available VRAM is 1 MB, VRAM page size is 16

KB. If available VRAM is 2 MB, VRAM page size is 32 KB.

Note: The size of memory pages and the number of memory pages is subject to change at any time, based on the current hardware design. Do not rely on these numbers within your titles.

To find out the page size of either type of memory, a task executes `GetPageSize()`.

Warning: VRAM is also divided into banks. Each bank of VRAM is 1 MB large. The SPORT bus can't transfer blocks of memory from one VRAM bank to another, so allocating VRAM within a single bank is important for anything involving SPORT transfers. When bank location is important, a task should always specify a bank, even though in a system with only 1 MB of VRAM, only one bank is available. The task can never be sure that it's not running in a 2 MB VRAM system with two banks of VRAM.

To find out in which bank of VRAM an address is located, a task executes the `GetBankBits()` call.

Allocating Memory

Whenever a task starts from a file, the file includes the task's memory requirements: how much code space and how much data space the task requires to run. Those memory requirements are determined when the task's source code is compiled-set by the size of arrays dimensioned and other factors. The kernel automatically allocates that much memory to the task before the task loads and runs. If the task requires extra memory once it's running, it must use the kernel memory-allocation function calls `AllocMem()`, `AllocMemFromMemLists()`, `AllocMemBlocks()`, or `malloc()` to request that memory.

In a memory request, the task specifies the type of RAM it requires:

- Any kind of RAM
- VRAM only
- Cel RAM (any RAM accessible to the cel engine)
- Memory that doesn't cross a page boundary
- Memory that starts on a page boundary

The task also specifies the amount, in bytes, of the memory it wants.

The kernel dedicates memory to a task a page at a time, and then allocates memory to the task from within those dedicated pages. Consider an example: a 3 MB 3DO system has a 32- KB page size; a task starts that requires 8 KB of memory. The kernel dedicates a single 32- KB page to the task by putting a fence (discussed later in this chapter) around it. The kernel allocates the first 8 KB of the page to the task. The task then requests another 20 KB of memory; the kernel allocates the next 20 KB of the page to the task. 4 KB of free RAM remain in the page for future allocation, as shown in Figure 1.

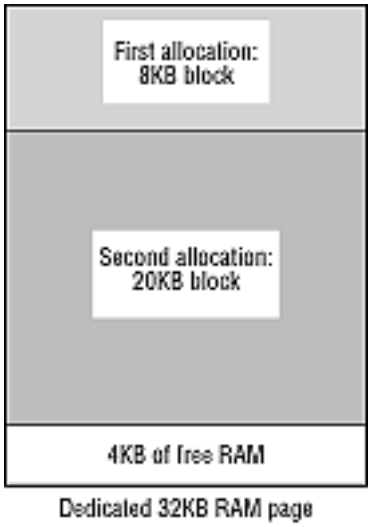

Figure 1: *Kernel page allocation.*

When a task requests a block of memory that is larger than any contiguous stretch of free RAM left in a task's RAM pages, the kernel dedicates a new page (or pages) of RAM to the task. It joins the new pages with the memory already in the task's free list. It then allocates RAM from the new page(s). The kernel will dedicate as many pages of RAM to the task as necessary to supply contiguous RAM; if it can't find that much contiguous free RAM, it notifies the task that it failed to allocate the memory.

To see how that works, consider the last example where a task started with one page of dedicated RAM, used 8 KB of the page for startup, and then requested and received 20 KB more. The task now requests 6 KB more RAM, but there are only 4 KB free in its dedicated pages. The kernel dedicates another page of RAM to the task, a page that can't be guaranteed contiguous, to the first page of RAM. In this example, consider the new page not to be contiguous. The kernel then allocates the first 6 KB of the new page to the task. If it had tried to allocate the last 4 KB free in the first page along with the first 2 KB of the next page, the block of memory would have been noncontiguous when it crossed the page boundary, so the kernel allocates all of the memory block from the second page as shown in Figure 2.

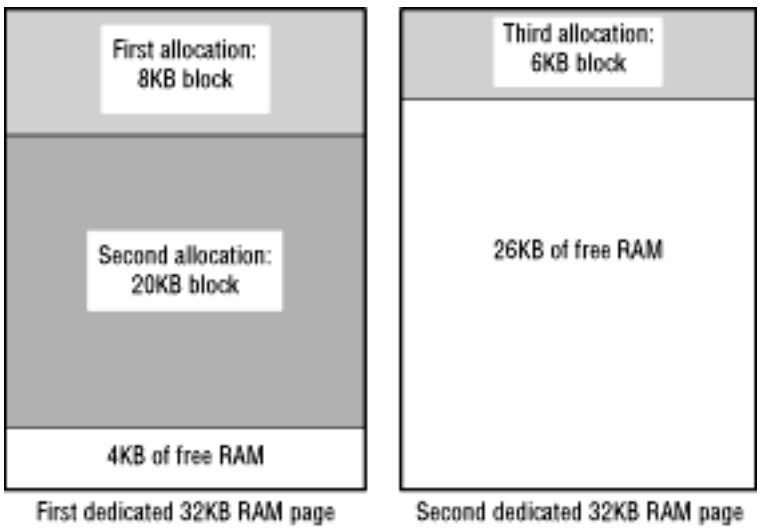

Figure 2: *Continuous allocation of memory block.*

Memory Fences

When the kernel dedicates full pages of memory to a task, it sets up a *memory fence* around the dedicated memory. The task can write to any address within its own memory, but normally it can't write outside of its dedicated memory. The only exception is when one user task writes to the memory of another user task with the second user task's permission. The first task can write there *only* with permission from the second task; if it doesn't have permission, it can't write there. A user task can never write directly to system memory (memory allocated to system tasks) because system tasks never grant write permission to user tasks.

When a task writes (or tries to write) to RAM addresses beyond its allocated memory, it can have one of two effects:

- If the task tries to write outside its fence, the kernel aborts the task.
- If the task writes inside its fence, the task isn't aborted, but it can write over its own data, causing unforeseen problems.

Although fences restrict tasks from writing outside of their allocated memory, they don't restrict tasks from reading memory elsewhere in the system. A user task can read memory allocated to other user tasks as well as to system tasks.

Returning Memory

The kernel keeps track of free RAM in two ways: it keeps a list of all RAM pages allocated to tasks (and so it knows which pages are free) and it lists free blocks of RAM within the dedicated pages. Whenever a task is finished with a block of allocated RAM, it can free the block for further allocation by using a function call that specifies the beginning address and size of the block. (The Kernel folio includes a number of memory-freeing calls, such as `FreeMem()` and `free()`.) If all the allocated blocks within a dedicated page of RAM are freed, the kernel knows that the page is free but keeps the page dedicated to the task for future allocation calls.

When a task wishes to release free RAM pages back to the kernel so the kernel can dedicate them to other tasks, the task issues a `ScavengeMem()` call. This call causes the kernel to reclaim free pages and to list them once again in the system free page pool.

Whenever a task quits or dies, all of its memory returns to the free RAM pool-*unless* it's a thread. When a thread dies, its memory remains the property of the parent task, because a thread shares the memory of its

parent task.

Sharing Memory

The kernel gives every page of memory dedicated to a task a status that tells which task owns the page and whether other tasks can write to that page or not. The status is set so that only the owning task can write to the page. If the owning task wants to share write privileges with another task (or with several other tasks), it can change the status with the call `ControlMem()`, which can take three actions:

- Specify another task and give it write privileges to the page.
- Specify another task and retract its write privileges to the page.
- Specify another task and give it ownership of the page.

As long as a task owns a page, it can change the page status to any state it specifies-it can turn write privileges on and off for other tasks or even for itself. However, once a task transfers ownership of a page to another task, the original task can no longer set that page's status, which is under the sole control of the new owner task. If the original task tries to write to the page, it will abort. Any I/O operation using that page as a write buffer will also abort.

Working With Folios

Portfolio's folios, each a collection of function calls dealing with a different aspect of 3DO operation, come in two types:

- **Permanent,** where the folio is constantly in RAM and ready to handle function calls.
- **Demand-loaded,** where the folio is stored on disk and must be loaded first before tasks can make function calls to the folio.

The Kernel folio is a permanent folio by necessity; it includes the function calls that open other folios. Other permanent folios are the Graphics, Audio, Math, and File folios. Examples of demand-loaded folios include the International and Compression folios.

Demand-loaded folios are automatically fetched from disk when they are needed by a task. When a task tries to open the folio, the kernel first looks in memory for the folio. If the folio isn't found in memory, the kernel then tries to load it from disk. Once the folio has been loaded from disk, it can be used by other tasks in the system without having to reload it.

When demand-loaded folios are no longer in use by any task, the kernel can remove them from memory to make the memory available to other uses in the system.

Managing Items

In a multitasking system, tasks often share resources such as RAM, I/O devices, and data structures. If the system doesn't manage those resources carefully, a task that depends on a shared resource can be brutally disappointed. For example, one task creates a data structure for a second task to read and act upon. If the first task dies and its data structure is no longer maintained or ceases to exist, the second task can find itself reading erroneous data that can crash it. In another example, a task creates a data structure and then use an invalid pointer to that structure (typically a NULL or uninitialized pointer). Because the pointer doesn't point where it should, the task can read totally erroneous data or, even worse, try to write to the data structure and crash itself, another task, or the 3DO system.

Note: Using a NULL pointer to read is illegal and causes an abort to occur on 3DO development systems.

Items

To insure the integrity of shared resources, the kernel provides *items.* An item is a system-maintained handle for a shared resource. The handle contains a globally unique item number and a pointer to the resource. When a task needs access to a shared resource, the task simply asks for the resource by item number instead of using a pointer to point directly at the resource. The kernel checks its list of items and, if it finds the requested item, performs any requested actions on the item. If the kernel finds that the item no longer exists, it informs the requesting task that access failed, and the task can go on without crashing itself or the system.

An item can refer to any one of many system components: a data structure used to create graphics, an I/O device, a folio, or even a task itself. In fact, almost all system-standard structures must be created as items. You'll find one of the most commonly used kernel calls is `CreateItem()`, which, appropriately enough, creates items. You use it to start a task, to lay out a graphics environment, to create messages to send between tasks, and to handle many other Portfolio functions.

Creating an Item

To create an item, a task typically creates a list of parameters (tag arguments) for the item. The parameters can include a name for the item, its dimensions and contents, and other important information. The task then uses `CreateItem()` to ask the kernel to create the item. The task supplies an item-type number that specifies the type of item to create, and a pointer to the list of parameters for the item. The kernel has a predefined set of item types (which you can find in the *3DO Portfolio Reference Manual*). Each item requires different parameters to define it.

The kernel receives the item parameters, and-if they're correct for the specified item type-creates the item. The item contains a globally unique ID number, the item type, a pointer to the resource handled by the item, and other parameters passed to it. The kernel returns the item number to the task that created the item. The kernel records the fact that the item belongs to that task.

Portfolio contains a large number of convenience calls that let you easily create items; for example, `CreateMsgPort()` and `CreateThread()`. It is generally simpler to use these higher-level routines instead of calling `CreateItem()` directly.

Opening Items

Many system items are predefined and stored on the system boot disk. They don't need to be created from scratch, only opened. To open a system item, a task uses the `FindAndOpenItem()` call to specify the type of the item to open, provides an array of the arguments required to open the item, and passes that information along to the kernel to open the item. The kernel finds the item on the disk, brings necessary data structures into system memory, assigns the item a number, and returns the item number to the task that asked for the item to be opened. When the task is finished with the opened item, it uses the `CloseItem()` call to close the item.

Using Items

Once an item is opened or created, tasks can use it by specifying its item number. If a task doesn't have the number for an item it wants to use, or knows the item number but doesn't know what type of item it is, the kernel provides item-handling function calls such as `FindItem(), FindNamedItem(),` and `LookupItem()`. These calls help find items by name, number, or other criteria, and then return item type, item number, or other information about the item.

Note: Some types of items, such as folios and devices, must be opened with `OpenItem()` or `FindAndOpenItem()` before a task can use the item.

The kernel provides the `SetItemPri()` call to change the priority of an item within a list of items. It also provides the `SetItemOwner()` call to change ownership of an item.

Deleting Items

Whenever a task finishes using an item that it created, it can delete the item with `DeleteItem()`, which removes the item and frees any resources committed to it, such as RAM devoted to the item's data structure. The deleted item's number is not recycled when new items are created, so the kernel can inform tasks trying to use that item number that the item no longer exists.

There is one important rule about item deletion: a task can't delete any item it doesn't own. This means that if a task creates an item and then passes ownership to another task, the first task can't delete the item-only the new owner can delete the item.

Whenever a task or thread quits or dies, the kernel automatically deletes all items that it created, and closes any items that it opened.

Semaphores

Many tasks running on a 3DO system can share data structures, storing a structure in one task's memory and allowing one or more outside tasks to read and write to that structure. Because one task using the data structure can never be sure what another task may be doing to the same structure, dangers arise. For example, one task writes to a data structure at the same time that another task writes to it. One task overwrites the data of another, and neither is aware of what happened. To avoid conflicts like this, a task must be able to lock down a shared data structure while it is working on it, and then release the structure when it's done.

To provide a lock, the kernel offers an item called a *semaphore*, which is defined as an element of a data structure. You can think of the semaphore as an "occupied" sign for a data structure. A polite task checks the semaphore of a data structure before it uses the structure. If the semaphore says that the structure is currently unused, the task can go to work on the structure. If the semaphore says that the structure is in use, the task must either wait until the semaphore says the structure is free or return to execution without using the structure.

To use a data structure with a semaphore, a task makes the function call `LockSemaphore()` to lock the structure's semaphore. If the semaphore is unlocked, the kernel locks it and the task can proceed with its business, using the semaphored data structure as it sees fit without interference from other polite tasks. When the task is done, it makes another function call, `UnlockSemaphore()`, to unlock the semaphore, releasing the semaphored data structure for use by other tasks.

When a task calls `LockSemaphore()`, it specifies what it will do if the semaphore is already locked: wait for the semaphore to be unlocked (putting itself in the wait queue); or return to execution without using the semaphored structure. If the semaphore is locked, the task acts accordingly.

The semaphore is a completely voluntary mechanism; it is what its name implies, only a flag that tells whether a data structure is in use or not. It does *not* deny write permission to tasks that want to use the data structure without checking the semaphore. If you want to share a data structure with another task, be sure that the other task is written to check for the semaphore before it goes about its business.

Intertask Communication

When one task needs to communicate with another task, it can be a simple matter of notification ("Hey! I'm finished doing what you asked me to do") or a more involved passing of detailed information ("Here's the table of values you asked me to calculate"). Portfolio provides mechanisms to handle both: *signals* for simple notification and *messages* for passing detailed information.

Signals

A signal is a kernel mechanism that allows one task to send a flag to another task. The kernel dedicates one 32-bit word in a task's TCB (task control block) for receiving signals. The kernel writes to that word each time it carries a signal to the task.

The 31 least-significant bits of the 32-bit signal word each specify a different signal; the most-significant bit is used for errors. Eight of the signal bits (0 to 7) are reserved by the kernel for system signals; the other 23 bits (8 to 30) are left for custom user-task signals.

Allocating Signals

A task can't receive signals from another user task unless it has allocated a signal bit on which to receive. To do so, it uses the `AllocSignal()` call, which returns a 32-bit value containing the next unused user signal bit (the 32-bit value is called a *signal mask*). The task can send its signal mask to other tasks; they can then send a signal back to the task, using the user signal bit set in its signal mask.

All tasks can receive system signals at any time. The lower-8 signal bits are reserved for this purpose. For example, the system sends a task a signal (`SIGF_DEADTASK`) whenever one of its child threads or task dies.

Sending a Signal

To send a signal to another task, a task prepares a 32-bit signal mask. It sets the appropriate bit (or bits, if it's sending more than one signal) in the mask to 1 and sets the rest of the bits to 0. For example, if the task wants to send a signal using bit 14, it creates the signal mask "00000000 00000000 01000000 00000000" (in binary). The task then uses the `SendSignal()` call to specify the item number of the task it wants to signal and passes along the signal mask. The kernel logically ORs the signal mask into the receiving task's TCB. Bits set in the `t_SigBits` field of the TCB indicate signals that the task has received, but not yet acknowledged.

Receiving a Signal

A task waits for one or more signals by using the `WaitSignal()` call. The kernel checks to see if any of the bits in the task's signal mask match the bit mask passed `WaitSignal()`, indicating that a signal has been received on that bit. If so, `WaitSignal()` clears the bits that match and immediately returns, letting the task act on any signals it has received. If there are no received signals in the signal mask, the task is put in the wait queue until it receives a signal it wants.

Freeing Signal Bits

To free up signal bits that a task has allocated, the task uses the `FreeSignal()` call to pass along a *free signal mask.* The free signal mask should have a 1 in each bit where the signal bit is to be freed (that is, set to 0 in the signal mask) and a 0 where the signal bit remains as it is.

Messages

A message is an item that combines three elements: a variable number of bytes of message data, 4 bytes available for an optional reply to the message, and a specified place (a reply port, explained later) where a reply to the message can be sent.

A message won't work without two *message ports:* one created by the task receiving the message and another created by the task sending the message. The message port is an item that sets a user signal bit for incoming message notification. It includes a message queue that receives and stores incoming messages.

Creating a Message

To create a message, a task can use a number of calls including `CreateMsg()`, `CreateSmallMsg()`, and `CreateBufferedMsg()`. These functions accept a string of text as the message's name, a priority for the message, and the item number of the reply port for replies to the message. It returns the item number of the newly created message for working with the message later.

Creating a Message Port

To create a message port, a task uses the `CreateMsgPort()` `call,` which it provides with a string of text as a message port name. The kernel creates a message queue in system RAM for the message port, automatically assigns a user-task signal bit for the message port, and gives the message port an item number. The task is now ready to receive messages at the port.

Sending a Message

If a task wants to send a message to another task, it must first know the item number of a message port of the receiving task. (If it knows the name of the message port, it can use the `FindMsgPort()` call to find the item number.) The sending task then uses either `SendMsg()` or `SendSmallMsg()` to fill out a message, providing a destination message port item number and some message data to pass. The kernel inserts the message, according to priority, into the destination port message queue, then signals the receiving task that a message has arrived at its message port.

Receiving a Message

To receive a message, a task has two options:

- It can use the `GetMsg()` call to check the message port and retrieve the top message in the list if there are any messages. If there are no messages, the task resumes execution.

- It can use the `WaitPort()` call to wait for a message. This puts the task into wait state until it receives a message at its message port. The task then retrieves the message and resumes execution.

Replying to a Message

A task that sends a message usually needs a reply from the task that receives the message, so the sending task must specify a message port of its own as the *reply port.* (If the sending task doesn't have its own message port, it must create one before creating a message.) When the receiving task receives the message, it uses either `ReplyMsg()` or `ReplySmallMsg()` to return the same message to the reply port with a 4-byte reply written into the message (stored in the 4-byte `msg_Result` field of the `Message` structure). The sending task receives the reply and reads the 4-byte reply.

Interpreting a Message

When one task sends a message to another task, the meaning of the message data is completely arbitrary and is determined by the two tasks sharing the message. In many cases, the message data is composed of a pointer to a data structure created in the sending task's memory along with the data structure's size. The receiving task then uses the pointer and size to read the data at that address.

Portfolio Error Codes

Portfolio has a uniform definition for the format of error codes. Whenever system components must return an error, they always use the standard error code format explained here.

All Portfolio error codes are negative 32-bit numbers that are subdivided into multiple subcomponents. Using these components, you can identify which subsystem generated the error, and get a module-specific error number. Table 1 lists the various components of an error code:

Table 1: *Error code components.*

Bit(s)	Purpose
31	This bit is always set. It makes all error codes negative numbers.
25-30	Object type that generated the error. This field identifies what generated the error: a folio, a device, a task, or another object type, generated the error. Possible values for this field include ER_FOLI, ER_DEVC, ER_TASK, or ER_LINKLIB.
13-24	Object ID. This is a code that uniquely identifies the component that generated the error. The value for this field is created with MakeErrID() and is basically two 6-bit characters identifying the module that caused the error. For example, kernel errors have an object ID of Kr.
11-12	A severity code. Possible values for this field include: ER_INFO, ER_WARN, ER_SEVERE, or ER_FATAL.
9-10	An environment code. This code defines who created the module that generated the error. Possible error values for this field include: ER_E_SSTM for system modules, ER_E_APPL for application modules, and ER_E_USER for user-code modules.

8	Error class. Possible error values for this field are: either ER_C_STND for a standard error code or ER_C_NSTND for a nonstandard error code. Standard error codes are errors that are well known in the system and have a default string to describe them. Nonstandard errors are module-specific and the module must provide a string to the system to describe the error.
0-7	The actual error code. The meaning of this code depends on the module that generated the error, and whether it is a standard or nonstandard error.

The `PrintfSysErr()` kernel call accepts any error code, and prints an error string describing the error to the debugging terminal. When developing code, it is extremely useful to know why a system call is failing.

Note: Besides handling multitasking, memory management, folios, items, intertask communication, and linked lists, the kernel also handles I/O and hardware control-both topics for another section. You'll find them described in The Portfolio I/O Model."

The Portfolio I/O Model

This chapter provides an overview of the 3DO system's hardware devices and the system software that controls them. This chapter contains the following topics:

Introduction

The 3DO environment is rich in device possibilities: any 3DO unit may be set up with few or many devices. Manufacturers can make 3DO units with a bare-bones configuration, or they can add extra devices such as a second CD-ROM drive or a digital television tuner. Once the 3DO unit is out of the box, users can add more devices by plugging in any of a large variety of controls, from simple controller pads to keyboards, or by adding peripherals such as modems or RAM mass storage.

Because a task can run on a multiplayer with any number of devices attached to it, a task can't assume that it knows what devices are available to it. Consider for example if a task knew the low-level details of a particular CD-ROM drive. If the drive hardware were to change between 3DO manufacturers, the task would suddenly fail to operate correctly.

To help a task traverse a world of devices that may change from minute to minute, Portfolio provides device drivers and I/O calls that can sense attached devices and know how to communicate with them, all without specific hardware knowledge on the part of client tasks.

Hardware Devices and Connections

The 3DO standard defines a set of buses and ports for connecting hardware devices. These bus and port definitions make the best use of existing hardware technologies. Keep in mind a very important warning as you read about the hardware described in this chapter: *the bus and port definitions may change.* As hardware prices fall, better performance will be possible using advanced hardware that costs less than the currently specified hardware; the 3DO standard *will* change to take advantage of better hardware as it becomes available. In other words, the hardware descriptions in this chapter are a snapshot of the current 3DO standards. Don't write software that assumes 3DO hardware is always going to be manufactured as it is now. Fortunately, you don't have to write hardware-specific code, because Portfolio takes care of the specifics for you, as you'll read later in this chapter.

Internal Buses

The 3DO system currently offers a set of three internal buses that provide connections for devices mounted within the 3DO box. They include the following:

- The **main data bus** that connects the CPU to RAM. It can also connect to a full motion video (FMV) cartridge.
- The **SlipStream** bus that currently has a video-out signal and, in the future, may support video-in as well.
- The **Slow bus** that is a simple, slow, and low-cost 8-bit bus for devices mounted on the motherboard. Those devices can include ROM, NVRAM, and a digital tuner.

The Expansion Port

The expansion port, which is a parallel port, provides external connections for plug-in modules that a user may want to add to a 3DO unit. Those modules include devices such as an auxiliary CD-ROM drive, a modem, RAM mass storage, and, in the future, read/write mass storage such as a hard disk drive.

To support high-power expansion devices, the expansion port uses a custom protocol that provides a fast data rate. The port can support up to 15 devices, all of which are auto-configuring, so users don't have to set DIP switches or set confusing software parameters to use a device; they just plug the device in and let the software do the rest.

The Control Port

The control port, which is a serial port, is used mainly to connect user-interface devices that control the 3DO unit. These devices include:

- The **controller pad** a standard control that includes a joypad and buttons, and may include a headphone jack. This device is used to point, select, and control on-screen action. It can also provide stereo sound accompaniment through the headphone jack.
- The **photo-optic gun** is used to shoot at targets on the video screen.
- **Stereoscopic glasses** are used to view stereoscopic displays.
- An **interlink unit** connects one 3DO unit to one or more other units.
- A **mouse or trackball** is used to point to and select objects on the screen.
- A **keyboard** is used to type in text.
- A **robot control device** is used to remotely control a mechanical robot.
- Any number of **custom devices** created and sold by 3DO-licensed manufacturers.

The control port supports a daisy chain of devices. Because a new device can always be plugged in to the last device in the chain, the control port doesn't have a fixed maximum number of devices, but instead is limited by the data bandwidth of the port. In other words, you can keep plugging in more control devices until the control port is overwhelmed with data and starts to choke up. If this happens, the control port fails to recognize the newly plugged in control devices that go beyond the port's capacities.

The control port is designed to be simple, robust, and convenient to users, who may plug and unplug controllers while software is running. All 3DO control devices are autoconfiguring. Portfolio polls the control port to see what's plugged in, and provides the appropriate drivers to work with each device. The control port is not fast when compared to the expansion port and internal buses; it reads and writes data once every video field (which occurs 60 times per second on NTSC systems, and 50 times per second on PAL systems), and currently handles up to 2048 bits of information per field. In addition to data, the control port carries an analog stereo audio output, which can be heard over earphones plugged into a controller stereo jack.

Tasks interface to the control port using the event broker. To learn about interfacing to external control port hardware, see The Event Broker.

The Event Broker

This chapter shows how a task uses the event broker to work with interactive devices. This chapter contains the following topics:

About the Event Broker

Working with an interactive device such as a controller pad poses a challenge to traditional I/O architectures; a user can change the device's state at any time, so a task working with a device must monitor it constantly. Successful device monitoring requires a task to run a loop that continually polls the interactive device, or requires the interactive device driver to accept an I/O request but does not respond until it detects a change in the device. In the first case, the polling task uses a lot of processor time. In the second case, a task has to have several IOReqs pending to avoid losing any device changes while the driver returns an IOReq. Portfolio provides the *event broker*, a solution to interactive device I/O that requires neither polling nor multiple IOReqs.

The event broker is a task that constantly monitors activity on attached interactive devices. Currently, interactive devices are attached only to the control port on the 3DO system. The event broker monitors the control port to see what the devices are doing.

Activities on interactive devices are called *events.* Events include joypad presses and releases, key strokes, mouse rolling, plugging a new control device into the control port, unplugging a device from the control port, and many others. Events can occur on any hardware device attached to the control port. A hardware device is called a *pod.* The event broker always identifies the pod in which an event occurs.

The event broker starts when the system starts up, and stands between user tasks and the interactive pods on the control port. To work with interactive devices, a user task connects to the event broker and registers itself as a *listener.* A listener is a task that connects to the event broker to receive event notification and to pass data to pods. To become a listener, a task creates a message port for communication with the event broker. Then the task communicates its requests to the event broker with messages; the event broker responds to these messages with its own messages. I/O operations through the event broker are carried out through messages instead of IOReqs.

Specifying and Monitoring Events

The first message a listener task sends to the event broker contains two *event masks.* A listener task uses the first event mask to specify which of the 256 different event types it is interested in hearing about. This mask is the *trigger mask;* it lists all event types that can trigger an event message to the listener. The second event mask is the *capture mask*; it lists all event types for which the event broker must check status whenever it sends an event message.

In each video field the event broker checks pods for events, 60 times per second on NTSC systems, and 50 times per second on PAL systems. If any of the events it senses match an event in a listener task trigger mask, the event broker composes an event message to send to the listener. That event message

contains a pointer to an accompanying data block that contains information about all event types that are specified in either the trigger mask or the capture mask.

The event broker takes a "snapshot" of the pods' states whenever a triggering event occurs. For example, a listener can specify "controller pad button down" as a trigger event and then specify "gunlike-device button pressed" and "keyboard key pressed" as capture events. Whenever a controller pad button is pressed down in a pod, the event broker is triggered to send an event message to the listener. The event message's data block contains the status of "controller pad button down" (true because it was the triggering event), the status of "gunlike-device button pressed" (true or false, depending on whether or not the button was pressed on any attached gunlike devices during the field), and the status of "keyboard key pressed" (true or false, depending on whether or not any keys were pressed on any attached keyboards). The listener task now knows that the controller pad button was pressed, and knows whether or not any keyboard keys or buttons on gunlike devices were pressed during the same video field. The listener can then take appropriate action.

Because the event broker notifies listeners of events via messages, a listener task can enter the wait state for an event broker message, just as a task waits for any other message. This allows synchronous I/O with interactive pods. A listener can also continue execution after sending an event broker message, which allows asynchronous I/O. In either case, whenever a listener task receives a message from the event broker, it must reply to the event broker and say the message is processed and can be reused. If the listener task does not reply, the event message queue can fill up, and it can lose events in which it was interested.

Working With Input Focus

Input focus is an important concept to both 3DO users and 3DO tasks. When several tasks occupy the display at the same time, input focus determines which task should respond to the user's actions. For example, a pinball game occupies one part of the display, while a CD-ROM encyclopedia occupies another part. When the user presses the "left" direction of the controller pad's joypad, does the left pinball flipper flip, or does the reader for the encyclopedia move back a page? It depends on which program has the input focus. If the user chose the reader earlier (via menu selection, for example) so that the reader is now the active task, the reader holds the input focus and responds to the controller pad.

The event broker sends events to the appropriate focus holder-if connected tasks are set up to pay attention to focus. Whenever a task connects to the event broker, it can request to be one of three listener types:

- A *focus-dependent listener,* which only receives its specified events from the event broker when the event broker knows that the task has the input focus.

- A *focus-independent listener,* which receives all of its specified events from the event broker at all

times.

- A *focus-interested listener*, which receives all of its specified events when it has the focus, and only non-user interface type events when it does not have the focus.

Tasks such as the encyclopedia reader and the pinball game mentioned above are usually focus-dependent listeners because they only respond when the user selects the task. Tasks that offer a program-selection menu to the user are usually focus-independent because they must respond at any time to user requests, even if they do not hold the focus.

Reconfiguring or Disconnecting an Event Broker Connection

As a listener task works with the event broker, it can send a message requesting a new configuration at any time. The new configuration can reset the task's focus dependence, specify new capture and trigger events, set a new message port, or change other connection parameters. A listener task can also request to be disconnected from the event broker so that it will no longer be notified of interactive events.

Other Event Broker Activities

Although the event broker's main activity is monitoring and reporting on interactive device events, it can also:

- Provide lists of connected devices and listener tasks
- Send and receive I/O data from interactive devices
- Send generic commands to interactive devices
- Notify a listener task of events occurring in other listener tasks

Sending Messages Between Tasks and the Event Broker

All communication between the event broker and connected listener tasks is through messages (a process discussed in earlier kernel chapters). These messages, which pass from listener to event broker or from event broker to listener, are called *event broker messages.* Before programming with the event broker, you should understand how these messages work.

Message Flavors

Each event broker message comes in a *flavor* that identifies the purpose of the message. The flavor of the message determines the type (or types) of data structures contained in the data block, and specifies how the message recipient should handle the message. The data structure EventBrokerHeader is always the first field of the first data structure within a message data block. Its definition is shown below:

```
typedef struct EventBrokerHeader
{
   enum EventBrokerFlavor ebh_Flavor;
}  EventBrokerHeader;
```

EventBrokerHeader indicates the flavor of the message. Table 1 shows the available flavors.

Table 1: *Event broker message flavors.*

Event Broker Flavor	Operation Requested
EB_NoOp	No operation requested.
EB_Configure	(Task to EB) Connect this task to the event broker and register its configuration.
EB_ConfigureReply	Event broker reply to EB_Configure.
EB_EventRecord	(EB to task) The events listed in the data block occurred in this field and are events in which the

	task is interested.
EB_EventReply	A listener's reply to EB_EventRecord.
EB_SendEvent	(Task to EB) An event has happened within a task. This command is not currently implemented.
EB_SendEventReply	Event broker reply to EB_SendEvent.
EB_Command	This operation is not currently determined.
EB_CommandReply	Event broker reply to EB_Command.
EB_RegisterEvent	(Task to EB) Register this custom event name and assign an event number to that name. This command is not currently implemented.
EB_RegisterEventReply	Event broker reply to EB_RegisterEvent.
EB_GetListeners	(Task to EB) Tell the requesting task which tasks are connected to the EB.
EB_GetListenersReply	Event broker reply to EB_GetListeners.
EB_SetFocus	(Task to EB) Assign the input focus to a specified task.
EB_SetFocusReply	Event broker reply to EB_SetFocus.
EB_GetFocus	(Task to EB) Tell the requesting task which task has the current input focus.

EB_GetFocusReply	Event broker reply to EB_GetFocus.
EB_ReadPodData	(Task to EB) Send requesting task data from the specified pod. This command is not currently implemented.
EB_ReadPodDataReply	Event broker reply to EB_ReadPodData.
EB_WritePodData	(Task to EB) Write the enclosed data to the specified pod. This command is not currently implemented.
EB_WritePodDataReply	Event broker reply to EB_WritePodData.
EB_LockPod	(Task to EB) Lock the specified pod so only the requesting task can issue commands or write data to it. This command is not currently implemented.
EB_LockPodReply	Event broker reply to EB_LockPod.
EB_UnlockPod	(Task to EB) Unlock the specified pod so that other tasks can issue commands or write data to it. This command is not currently implemented.
EB_UnlockPodReply	Event broker reply to EB_UnlockPod.
EB_IssuePodCmd	(Task to EB) Issue this generic command to the specified pod.
EB_IssuePodCmdReply	Event broker reply to EB_IssuePodCmd.
EB_DescribePods	(Task to EB) Describe pods

```
                        |attached to the control port.
-------------------------------------------------------
EB_DescribePodsReply    |Event broker reply to
                        |EB_DescribePods.
-------------------------------------------------------
EB_MakeTable            |(Task to EB) Create a pod table
                        |and return a pointer to it.
-------------------------------------------------------
EB_MakeTableReply       |Event broker reply to
                        |EB_MakeTable
-------------------------------------------------------
```

With the exception of `EB_NoOp`, event broker messages come in pairs; every event broker operation request is answered with a reply from either the event broker or the receiving task. The reply confirms an operation's execution (or failure to execute). A reply message can carry information requested in the operation request.

All operation requests are sent from a listener task to the event broker with one exception: `EB_EventRecord`, which the event broker sends to a listener to tell it what specified events happened during a field.

Message Types

Portfolio supports three types of messages:

- Standard
- Small
- Buffered

To communicate with the event broker, a listener task can use either standard or buffered messages; small messages do not work.

Whenever a listener task asks the event broker for information, it uses a buffered message. For example, if the `EB_DescribePods` command is sent, the event broker puts the pod information in the buffer associated with the message and returns the message to the listener task.

If the listener task is giving information to the event broker, for example using an `EB_WritePodData` command, the task can use either a standard (pass by reference) or buffered (pass by value) message. The event broker reads the data from the data block, but does not modify it.

When a task replies to event broker messages, such as returning an `EB_EventRecord` message, it

should use the `ReplyMsg()` function. Always provide NULL for the data pointer, and 0 for the data size arguments to `ReplyMsg()`.

Flavor-Specific Message Requirements

Each message flavor requires a specific data structure (or set of data structures) to accompany it. The data structures are defined in the include file *event.h* (see Reading Event Data). The data structures are also discussed in the sections of this chapter that describe different event broker operations.

Operation requests and replies are sent with the same message. The event broker or receiving task fills in the appropriate reply data, and returns the message.

When requesting data from the event broker, a listener task must use a buffered message to hold the response. Because the reply data from an operation can be extensive, the message should have a large enough data block for the event broker to fill in data. If the data block is not large enough, the event broker puts no data in the buffer and returns an error.

Note that a reply message cannot copy useful data to the data blocks if the reply does not require large amounts of data. For example, a listener task can use an `EB_GetFocus` message to ask the event broker to report which task has the input focus . The event broker only needs to store the item number of the task with focus in the error field of the `EB_GetFocus` message and return the message without copying any data to the data block. In such a case, a task can use a standard message to request information instead of a buffered message because the buffer is not used.

Monitoring Events Through the Event Broker

Waiting for Event Messages on the Reply Port

After a task is connected as a listener, it can choose either of the following methods for event notification:

- The task can enter the wait state until it receives an event broker message on the reply port (synchronous I/O).
- The task can continue to execute, and pay attention to the event broker only when the task receives notification of a message received on the reply port (asynchronous I/O).

The Event Broker Trigger Mechanism

While listeners wait for event notification, the event broker checks the pods once per field for events. The event broker checks occurring events against the trigger mask of each connected listener. If an event matches a set bit in a listener's trigger mask and the listener has the input focus (if required), the event broker sends an event notification to the listener.

When the event broker sends an event notification, it uses its own event messages, which it creates as necessary. After the event broker creates an event message, it copies the information into the message's data block. The information contains a report of the status of each event type that matches a set bit in either the listener of the trigger or capture masks. You can think of the event message data block as a snapshot of the status of all listener-specified events during the field. Only events specified in the trigger mask can trigger the snapshot, but the status of all events specified in either mask is reported in the message.

For example, consider a listener that specifies the "Gun Button Pressed" event type in its trigger mask, and the "Control Button Update" and "Keyboard Update" event types in its capture mask. When any gun attached to the control port is pressed during a field, the event broker prepares an event message for the listener. The event message's data field reports a gun button press and the status of the Control Button Update and Keyboard Update events (indicating whether or not either of those events occurred during the field). If the "gun button pressed" event had not occurred, the event broker would not have sent an event message to the listener, even if keyboard or controller pad button updates occurred, because those event types were in the capture mask but not in the trigger mask.

Retrieving and Replying to Event Messages

After a listener has been notified of an incoming event message from the event broker, it must retrieve the message to read the reported events. To do so, it uses the `GetMsg()` call. The listener can then read the message's data block (described in Reading an Event Message Data Block below), and either act on the data or store it for later use.

After a listener has processed the event data in an event message, the listener must reply to the event broker. Its reply tells the event broker that the event message is once again free for use. If the listener does not reply to an event message, the event broker assumes that the message was not processed. Each unprocessed event message is one element in the listener's event queue. When the number of unprocessed event messages exceeds the maximum event queue depth, the event broker stops reporting events to the listener. Consequently, the listener loses events until it can process and free up messages in its queue. If the listener asks to be notified of an event queue overflow, the event broker reports it as soon as an event message is free. The update event types (`EB_ControlButtonUpdate`, `EB_MouseUpdate`, and so on) are all reported as occurring, because an event queue overflow is considered an update.

To reply to an event message and return it to the event broker, a listener uses the `ReplyMsg()` call. Always pass NULL as the data pointer, and 0 as the data size parameter to `ReplyMsg()`.

Reading an Event Message Data Block

When a listener receives an event message from the event broker, a pointer to the event data block is in the `msg_DataPtr` field of the message structure, and the size in bytes of the data block in the `msg_DataSize` field. To interpret the event data stored in the event message, the listener task must be able to read the data structures used in the data block.

Event Data Block Structure

When the event broker reports events in an event data block, it can report one or more events. Therefore, it must have a flexible data arrangement within the data block to report any number of events. An event data block starts with an EventBrokerHeader structure that specifies the message flavor as `EB_EventRecord` and follows with one or more `EventFrame` structures, each containing the report of a single event. (Each event report is called an *event frame*.) The final structure in the data block is a degenerate EventFrame data structure, which indicates the end of the event frames.

The EventFrame Data Structure

The EventFrame data structure is defined in *event.h* as follows:

```
typedef struct EventFrame
{
    uint32            ef_ByteCount;                      /* total size of
EventFrame */
    uint32            ef_SystemID;                      /* 3DO machine ID, or
0=local */
    uint32            ef_SystemTimeStamp;                     /* event-count
timestamp */
    int32            ef_Submitter;                      /* Item of event sender, or
0 */
    uint8            ef_EventNumber;                     /* event code, [0,255] */
    uint8            ef_PodNumber;                     /* CP pod number, or 0 */
    uint8            ef_PodPosition;                     /* CP position on
daisychain, or 0 */
    uint8            ef_GenericPosition;                     /* Nth generic device
of type, or 0 */
    uint8            ef_Trigger;                    /* 1 for trigger, 0 for
capture */
    uint8            rfu1[3];
    uint32            rfu2;
    uint32            ef_EventData[1];                     /* first word of event
data */
} EventFrame;
```

- `ef_ByteCount` gives the total size of the `EventFrame` in bytes. Because the event data at the end of the frame varies in size depending on the event, this value changes from frame to frame.
- `ef_SystemID` reports the ID number of the 3DO unit where this event occurred. This value is useful if two or more

3DO units are linked together (as they might be for networked games). If the event occurred on the local 3DO unit, this value is set to 0.

- `ef_SystemTimeStamp` is the exact system time the event broker recorded this event. This time value corresponds to the current vblank count value as provided by the timer device's `TIMER_UNIT_VBLANK` unit.

- `ef_Submitter` gives the item number of the event sender. This value is important when multiple listener tasks tied into the event broker send events to each other. This item number is the item number of the sending task. If the event comes from a pod and not a task, this field is set to 0.

- `ef_EventNumber` is an event code from 0 to 255 that identifies the generic type of event. These event types are the same as the event types identified within a configuration event mask. The include file *event.h* defines a constant for each type as shown in Table 2.

- `ef_PodNumber` gives the unique pod number of the pod where an event originated. This number is constant and is assigned when a pod is first plugged into the 3DO unit. It does not change if the pod changes its position in the control port daisy chain. This value is set to 0 if the event did not originate in a pod (for example, an event generated by another task or the CD-ROM drive).

- `ef_PodPosition` gives the position of the event-reporting pod in the control port daisy chain. Numbering starts with 1 for the first pod, and continues consecutively the further the chain extends from the 3DO unit. If the event did not originate in a pod, this value is set to 0.

- `ef_GenericPosition`, gives the daisy-chain position of the event-reporting pod among identical generic device types in the daisy chain. A generic device type is a functional description of the pod or part of the pod, and currently includes the following defined types:

    - `POD_IsControlPad`
    - `POD_IsMouse`
    - `POD_IsGun`
    - `POD_IsGlassesCtlr`
    - `POD_IsAudioCtlr`
    - `POD_IsKeyboard`
    - `POD_IsLightGun`
    - `POD_IsStick`
    - `POD_IsIRController`

- `For example, an event occurs in the second controller pad in the daisy chain, in which case this value is 2, or in the fourth photo-optic gun in the daisy chain, in which case this value is 4. This value is set to 0 if the event did not occur in a generic device.`
- A single pod attached to the 3DO unit can have more than one generic device type. For example, a single pod can be a combination control pad and audio controller. In that case, when an event occurs on its control pad buttons, the generic position listed in the event frame is that of a control pad among other control pads connected to the 3DO unit.

- `ef_Trigger` identifies an event as one that triggered the event notification or as a captured event that did not trigger the notification. If the value is 1, the event was a triggering event; if the value is 0, the event is a captured event.

- `ef_EventData` is the first word of a data structure that contains data about an event. The rest of the data structure follows this field in memory. The definition of the data structure depends entirely on the generic device where the event

occurred. The data structures are described in Reading Event Data.

The Final (Degenerate) Event Frame

The final event frame within an event data block must be a degenerate event frame, which contains the value 0 for the `ef_ByteCount` field. The rest of the EventFrame data structure cannot be present.

Reading Event Data

The event data array at the end of each full event frame is a data structure that determines the type of device reporting the event. For example, an event occurring on a controller pad uses a structure designed to report control pad data. An event occurring on a mouse uses a structure designed to report mouse data.

Control Pad Data Structure

The data structure that reports control pad data is `ControlPadEventData`, defined in *event.h* as follows:

```
typedef struct ControlPadEventData
{
  uint32 cped_ButtonBits;  /* left justified, zero fill */
} ControlPadEventData;
```

- `cped_ButtonBits` contains a value whose bits tell the status of each controller pad button during the field. The constants defining these flag bits are as follows:
    - `ControlDown` - the down arm of the joypad cross.
    - `ControlUp` - the up arm of the joypad cross.
    - `ControlRight` - the right arm of the joypad cross.
    - `ControlLeft` -the left arm of the joypad cross.
    - `ControlA` -the A button.
    - `ControlB` - the B button.
    - `ControlC` - the C button.
    - `ControlStart` - the P (play/pause) button.
    - `ControlX` -the X (stop) button.
    - `ControlLeftShift` - the left shift button.
    - `ControlRightShift` - the right shift button.

- The meaning of these bits varies for each type of control pad event:
    - For the `EVENTNUM_ControlButtonPressed` event, 1 means the button was pressed since the last field, 0 means the button was not pressed.
    - For the `EVENTNUM_ControlButtonReleased` event, 1 means the button was released since the last field, 0 means the button was not released.
    - For the `EVENTNUM_ControlButtonUpdat` event, 1 means the button is down; 0 means the button is up.
    - For the `EVENTNUM_ControlButtonArrived` event, 1 means the button is down; 0 means the button is up.

Mouse and Trackball Data

The data structure used to report mouse and trackball data is `MouseEventData`, defined in *event.h* as follows:

```
typedef struct MouseEventData
{
  uint32    med_ButtonBits;   /* left justified, zero fill */
  int32     med_HorizPosition;
  int32     med_VertPosition;
} MouseEventData;
```

- `med_ButtonBits` contains a value whose bits tell which mouse button (or buttons) generated the event. The constants defining these flag bits are as follows:
    - `MouseLeft` - the left mouse button.
    - `MouseMiddle` - the middle mouse button.
    - `MouseRight` - the right mouse button.
    - `MouseShift` - the mouse's shift button.
- 3DO mice currently have three buttons: left, middle, and right. The fourth constant provides for an extra shift button if one ever appears on 3DO mice.
- The meaning of the above bits varies for each type of mouse event:
    - For the `EVENTNUM_MouseButtonPressed` event, 1 means the button was pressed since the last field; 0 means the button was not pressed.
    - For the `EVENTNUM_MouseButtonReleased` event; 1 means the button was released since the last field, 0 means the button was not released.
    - For the `EVENTNUM_MouseUpdate` evenly; 1 means the button is down, a 0 means the button is up.
    - For the `EVENTNUM_MouseDataArrived` event; 1 means the button is down, a 0 means the button is up.
    - For the `EVENTNUM_MouseMoved` event; 1 means the button is down, 0 means the button is up.
- `med_HorizPosition` and `med_VertPosition` report the mouse's current position in absolute space. That position is reckoned from an origin (0,0) that is set when the mouse is first plugged in, and uses units that will typically (depending on the mouse) measure 100, 200, or 400 increments per inch. It is up to the task to interpret the absolute position of the mouse to a pointer position on the display.

Light Gun Data Structure

The data structure that reports light gun data is LightGunData, defined in *event.h* as follows:

```
typedef struct LightGunEventData
{
  uint32  lged_ButtonBits;      /* left justified, zero fill */
  uint32  lged_Counter;         /*cunter at top-center of hit */
  uint32  lged_LinePulseCount;   /* # of scan lines which were hit */
} LightGunEventData;
```

- `lged_ButtonBits` contains a value whose bits tell the state of the various triggers, buttons, and other pushable or flippable controls on a light gun. Most light guns have one trigger and one auxiliary button. The only flag bit currently defined for this field is:
    - `LightGunTrigger` - the main light gun trigger.

- `lged_Counter` contains a value which specifies the number of times that the light gun's internal clock (nominally 20 MHz) counted up, between the time it was reset (during vertical blanking) and the time that the light gun's optical sensor "saw" a flash of light from the display as the display's electron beam passed through its field of view. The beam travels left to right along each line, and downwards from line to line.

- `lged_LinePulseCount` contains the value 0 if the light gun sensor did not detect a sufficiently strong pulse of light while scanning the video field. In this case, the `lged_Counter` value may or may not be valid. If the light gun sensor detected a sufficiently strong pulse of light, `lged_LinePulseCount` contains a nonzero value. Some light guns can actually count the number of successive horizontal scan lines during a pulse and return that value in this field. For light guns of this sort, the `lged_LinePulseCount` is considered a "quality of signal" indicator. Other light guns simply return a constant value (typically 15) to indicate that their sensor detected at least one pulse that was strong enough to trip the sensor.

Joystick Data

The data structure used to report joystick data is `StickEventData`, defined in *event.h* as follows:

```
typedef struct StickEventData
{
  uint32            stk_ButtonBits;                 /* left justified, zero fill */
  int32            stk_HorizPosition;
  int32            stk_VertPosition;
  int32            stk_DepthPosition;
} StickEventData;
```

- `stk_ButtonBits` contains a value that identifies which joystick button (or buttons) generated the event. The constants defining these flag bits are as follows:
    - `StickCapability` - the AND of `Stick4Way` and `StickTurbulence.`
    - `Stick4Way` - determines how many buttons the stick has.
    - `StickTurbulence` - indicates whether or not the stick understands output commands.
    - `StickButtons` - the stick buttons.
    - `StickFire` - the joystick was fired.
    - `StickA` - the A button.
    - `StickB` - the B button.

- `StickC` - the C button.
- `StickUp` - the up button of the stick.
- `StickDown` - the down button of the stick.
- `StickRight` - the right button of the stick.
- `StickLeft` - the left button of the stick.
- `StickPlay` - the play button of the stick.
- `StickStop` - the stop button of the stick.
- `StickLeftShift` - the left-shift button of the stick.
- `StickRightShift` - the right-shift button of the stick.

- `stk_HorizPosition`, `stk_VertPosition`, and `stk_DepthPosition` contain binary numbers and return a value between 0 through 1023. A value of 0 means the joystick is pushed left or all of the way down. A value of 1023 means the joystick is pushed right or all the way up. Keep in mind that some joysticks cannot go all the way to 0 or 1023.

Device State Data Structure

When media is inserted or removed from a device, `EVENTNUM_DeviceOnline` or `EVENTNUM_DeviceOffline` respectively, the DeviceStateEventData structure contains the Item number of the device, and the unit number within the device that has gone on-line or off-line. The `DeviceStateEventData` structure is defined as follows:

```
typedef struct DeviceStateEventData
{
  Item    dsed_DeviceItem;
  uint32  dsed_DeviceUnit;
} DeviceStateEventData;
```

Filesystem State Data Structure

When a filesystem is mounted, dismounted, or placed off-line, `EVENTNUM_FilesystemMounted`, `EVENTNUM_FilesystemDismounted`, or `EVENTNUM_FilesystemOffline`, the FileSystemEventData structure contains the Item number of the Filesystem node, and the name of the filesystem which is changing state. The FileSystemEventData structure is defined as follows:

```
typedef struct FilesystemEventData
{
  Item  fsed_FilesystemItem;
  char  fsed_Name[FILESYSTEM_MAX_NAME_LEN];
} FilesystemEventData;
```

The Process of Event Monitoring

Although event broker messages come in enough flavors to allow a task to carry out a variety of I/O operations through the event broker, the event broker's main function is to monitor events in user-interface devices and report the events to listening tasks. From a listener task's point of view, the process of event monitoring has the following steps:

1. Connecting to the event broker
    - Create a message port
    - Create a configuration message
    - Create a configuration block
    - Send the configuration message to the event broker
    - Receive the event broker's connection reply message
2. Monitoring events through the event broker
    - Wait for event messages on the allocated message port
    - Read an event message data block
    - Process event data block information (if desired)
    - Reply to the event broker for event message receipt
3. Changing configuration (if desired) or disconnecting from the event broker
    - Send a new configuration data structure to the event broker

Each of these steps use event broker data structures and message flavors.

Connecting a Task to the Event Broker

Before a task can work with the event broker, it must connect to the event broker by sending a message that requests connection as a listener. To do so, the task sends a message of the flavor `EB_Configure`, using Portfolio's standard message-sending procedure. The message also specifies input focus, and indicates the event types in which the task is interested.

Creating a Message Port

Before creating a configuration message, a listener task must create its own message port to use as a reply port where the event broker will send messages. To do so, the listener uses the `CreateMsgPort()`call, which returns the item number of the new message port.

Creating a Configuration Message

After a listener task creates a message port, it must create a message using `CreateMsg()` or `CreateBufferedMsg()`. Both calls accept a string that names the message, supplies an optional priority number for the message, and provides the item number of the listener message port. `CreateBufferedMsg()` also takes the size of the buffer to allocate with the message. Both calls return the item number of the newly created message.

Creating a Configuration Block

To turn the newly created message into a configuration request, the listener task must create a configuration data block, which consists of the ConfigurationRequest structure, shown in Example 1.

Example 1: *ConfigurationRequest structure.*

```
typedef struct ConfigurationRequest
{
  EventBrokerHeader                          cr_Header;                  /* {
EB_Configure } */
  enum ListenerCategory                         cr_Category;
/* focus, monitor, or
                                                /* hybrid */
  uint32                                  cr_TriggerMask[8]; /* events to trigger on */
  uint32                                  cr_CaptureMask[8]; /* events to capture */
  int32                                  cr_QueueMax;                     /* max # events
in

                                                /* transit */
  uint32                                  rfu[8];                          /* must be 0
for now */
} ConfigurationRequest;
```

The listener task must fill in the following fields to set the parameters of its configuration request:

- `cr_Header` is an EventBrokerHeader data structure in which the event flavor `EB_Configure` must be set.
- `cr_Category` defines the listener type of input focus. It has four possible values:

  - `LC_FocusListener` sets the task to be a focus-dependent listener. It must have the focus to receive any events.
  - `LC_FocusUI` sets the task to be a focus-interested listener. It must have the focus to receive user interface events, but gets all other event types regardless of focus.
  - `LC_Observer` sets the task to be a focus-independent listener. It gets all event types regardless of focus.
  - `LC_NoSeeUm` sets the task to ignore all events completely, the equivalent of not connecting to the event broker.

- `cr_TriggerMask` and `cr_CaptureMask` contain the event masks that specify events in which the task is interested. The trigger mask specifies each event type that will trigger the event broker to notify the task; the capture mask specifies each event type that the event broker reports on whenever it notifies the task of an event triggering. These masks are discussed in The Event Broker Trigger Mechanism.
- `cr_QueueMax` specifies the maximum number of event reports that the event broker posts at any one moment in the task's event queue. This value can be set to the constant `EVENT_QUEUE_MAX_PERMITTED`, which sets the currently defined maximum number of events in the queue (20 in this release), or the constant `EVENT_QUEUE_DEFAULT`, which sets the currently defined default number of events (three in this release). The queue must have a minimum depth of at least two events.
- The `rfu` field is reserved for future use, and currently must be set to 0.

Event Types

The 256 different event types available through the event broker are divided into two main bodies:

- **System events**. 128 system-defined generic events.
- **Custom events.** 128 task-defined events primarily used for communicating events in one event broker listener to other listeners. Custom events are not currently implemented.

Both system events and custom events are divided into two types that are defined by the way the event broker reports them to focus-interested tasks:

- **UI events**, which are not reported to a focus-interested task if a task does not hold the focus. The event broker provides 64 system and 64 custom UI events.
- **Non-UI events**, which are reported to a focus-interested task even if the task does not hold the focus. The event broker provides 64 system and 64 custom non-UI events.

Table 1 shows how the 256 different bit values in a bit mask are divided into system and custom events, and then UI and non-UI events. Custom event types are not supported in this release of Portfolio.

Table 1: *Flag Bit constants defined in event.h.*

System UI Event Type Constants	Event Definition

EVENTBIT0_ControlButtonPressed	A controller pad button was pressed.
EVENTBIT0_ControlButtonReleased	A controller pad button was released.
EVENTBIT0_ControlButtonUpdate	A controller pad button was pressed or released, or control button information may be lost in an event queue overflow.
EVENTBIT0_ControlButtonArrived	Data from a controller pad button has arrived (it arrives every field).
EVENTBIT0_MouseButtonPressed	A mouse button was pressed.
EVENTBIT0_MouseButtonReleased	A mouse button was released.
EVENTBIT0_MouseUpdate	A mouse button was pressed or released, the mouse has moved, or mouse info may be lost in an event queue overflow.
EVENTBIT0_MouseMoved	A mouse has moved.
EVENTBIT0_MouseDataArrived	Data from a mouse has arrived (it arrives every field).
EVENTBIT0_GunButtonPressed	A gun button was pressed. This is not currently implemented.
EVENTBIT0_GunButtonReleased	A gun button was released. This is not currently implemented.
EVENTBIT0_GunUpdate	A gun button was pressed or released, or gun button information may be lost in an event queue overflow. This is not currently implemented.
EVENTBIT0_GunDataArrived	Data from a gun has arrived (it arrives every field). This is not currently implemented.
EVENTBIT0_KeyboardKeyPressed	A keyboard key was pressed. This is not currently implemented.
EVENTBIT0_KeyboardKeyReleased	A keyboard key was released. This is not currently implemented.

EVENTBIT0_KeyboardUpdate	A keyboard key was pressed or released, or keyboard information may be lost in an event queue overflow. This is not currently implemented.
EVENTBIT0_KeyboardDataArrived	Data from a keyboard has arrived (it arrives every field). This is not currently implemented.
EVENTBIT0_CharacterEntered	A character was entered. This is not currently implemented.
EVENTBIT0_GivingFocus	This task was given focus.
EVENTBIT0_LosingFocus	This task lost focus.
EVENTBIT0_LightGunButtonPressed	A light gun button was pressed.
EVENTBIT0_LightGunButtonReleased	A light gun button was released.
EVENTBIT0_LightGunUpdate	A light gun button was pressed or released, or light gun button information may be lost in an event queue overflow.
EVENTBIT0_LightGunFireTracking	Data from a light gun is being tracked.
EVENTBIT0_LightGunDataArrived	Data from a light gun has arrived (it arrives every field).
EVENTBIT0_StickButtonPressed	A joystick button was pressed.
EVENTBIT0_StickButtonReleased	A joystick button was released.
EVENTBIT0_StickUpdate	A joystick button was pressed or released, or joystick button information may be lost in an event queue overflow.
EVENTBIT0_StickMoved	A joystick has moved.
EVENTBIT0_StickDataArrived	Data from a joystick has arrived (it arrives every field).
EVENTBIT0_IRKeyPressed	An IR key was pressed and a button code is returned. This is device specific. This is not currently implemented.

EVENTBIT0_IRKeyReleased	An IR key was released and a button code is returned. This is device specific. This is not currently implemented.

Table 2 shows the flag bit constants defined in *event.h* for the system events, along with the meaning for each of the event types. These event type constants define flags for system-defined events, each of which can occur during a single control port polling.

Table 2: *Event type constants that define flags for system-defined events.*

System Non-UI Event Type Constants	Event Definition
EVENTBIT2_DeviceOnline	Media was inserted in a device.
EVENTBIT2_DeviceOffline	A device was removed.
EVENTBIT2_FilesystemMounted	A file system was mounted.
EVENTBIT2_FilesystemOffline	A file system went off-line.
EVENTBIT2_FilesystemDismounted	A file system was dismounted.
EVENTBIT2_ControlPortChange	A new device was plugged into or disconnected from the control port.
EVENTBIT2_PleaseSaveAndExit	This task was requested to save current status and exit.
EVENTBIT2_PleaseExitImmediately	This task was requested to exit immediately.
EVENTBIT2_EventQueueOverflow	This task's event queue has overflowed and the task has lost events.

Setting the Trigger and Capture Masks

Each of the masks in the ConfigurationRequest data structure is an 8-element array of 32-bit words that provides 256 bits, 1 bit for each event type. To set a mask, the task logically ORs the desired constants defined for each word of the array, and stores the results in the appropriate word. The constant name identifies the appropriate word for storage. For example, all the desired flag constants starting with `EVENTBIT0` must be logically ORed and then stored in word 0 of the array. All the desired flag constants starting with `EVENTBIT2` must be logically ORed and then stored in the second word of the array.

Sending the Configuration Message

Once the ConfigurationRequest data structure has been created and filled in, the listener task must send the message to the event broker. To do this, the task first must find the event broker message port using the `FindMsgPort()` call, as shown in Example 2.

Example 2: *FindMsgPort().*

```
{
Item ebPortItem;

    ebPortItem = FindMsgPort(EventPortName);
    if (ebPortItem < 0)
    {
        /* big trouble, the event broker can't be found! */
    }
}
```

After the listener task has the event broker message port item number, the task sends its configuration message using `SendMsg()`:

```
 SendMsg(ebPortItem, msg, &configuration, sizeof(configuration));
```

The `SendMsg()` call accepts four arguments: the item number of the event broker port, the item number of the message the task has created, a pointer to the ConfigurationRequest data structure the task has initialized, and the size of that data structure.

When the event broker receives the configuration message, it adds the task as a listener. The event broker reads the reply port contained in the message and uses that port whenever it needs to communicate with the listening task. The event broker also reads the ConfigurationRequest data structure and sets up the listener's configuration accordingly.

Receiving the Configuration Reply Message

When the event broker finishes processing the configuration message, it changes the message to a configuration reply message. That message contains a pointer to a single data structure: an EventBrokerHeader with the value `EB_ConfigureReply`. The event broker returns the message to the task requesting configuration. The receiving task checks the configuration message's result field (`msg_Result`, which is cast to an `Err` type) to see if the operation was successful or not. If the value is a negative number, it is an error code, and the configuration was not successful (the task was not connected). If the value is 0 or a positive number, the configuration was successful and the task is connected.

High-Performance Event Broker Use

The pod table interface gives applications a fast and efficient way to monitor the current state of certain control port devices on a field-by-field basis. The pod table does not require applications to process a large number of event messages.

The Pod Table

Devices such as analog joysticks and light guns provide applications with position information, as well as with button-state information. Button-state information does not change frequently (a few times per second in most cases), but positional information changes often during almost every video field (up to 60 times a second). These rapid position changes are inherent in the high resolutions of these input devices, and their tendency to "jitter" slightly as a result of small mechanical movements or electrical noise in the circuitry. Applications must track the position of an input device. For example, by moving a set of crosshairs on the display, an application may need to parse and process messages from the event broker many times each second This process can lead to an undesirable slowdown of the application.

The *pod table* is an alternative method for accessing the current position of devices such as the analog joystick and light gun, with substantially lower overhead for both the application and the event broker. This table (actually a set of arrays and data structures) contains the current position information, and current button-state, for up to eight generic control port devices in each family. By examining the contents of the pod table, applications can track device position whenever it is convenient for them to do so.

Gaining Access to the Pod Table

The event broker constructs the pod table by the event broker when an application sends it a message asking for access to the table. This message has the standard event broker format and uses a message code of `EB_MakeTable`.

The event broker builds the table and replies to the application's message by sending an `EB_MakeTableReply` message whose data structure includes a pointer to the table. The table is updated whenever new data arrives from the control port. An application that gains access to the table in this fashion can "poll" the contents of the table at its convenience. The table contains a semaphore that the application must lock to ensure the event broker does not update the table while the application examines it.

Relinquishing Access to the Pod Table

The pod table is retained in memory and kept up to date, as long as the application that requested it is executing. The system maintains the table as long as the event broker receives a table-request message from at least one message port. As soon as the event broker detects that the last such message port was deleted, it ceases updating the pod table and releases the memory occupied by the table.

Structure of the Pod Table

The pod table consists of two portions: a fixed-size and fixed-format section and a set of variable-size arrays, which are created when needed. The variable size arrays may not always be present. Figure 1 shows the fixed-size section and fixed-format.

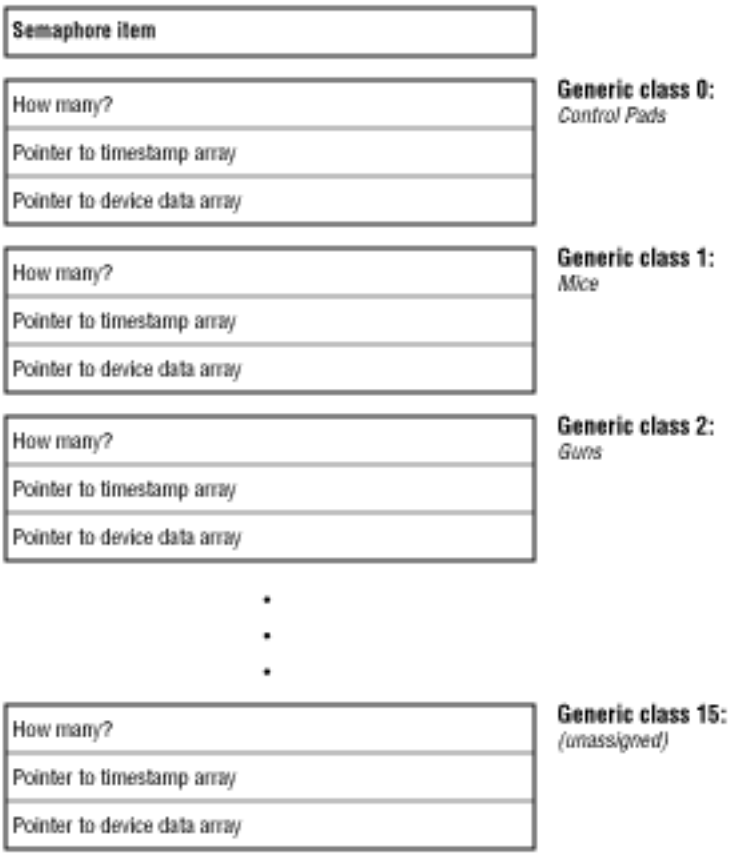

Figure 1: *Pod table structure fixed-size and fixed-format section.*

The table begins with a semaphore item. It is followed by a total of 16 almost-identical substructures: one substructure per generic class, with the names of the substructures indicating the generic class to which they refer. Within each substructure are two pointers: one to an array of `uint32` timestamps, and another to an array of generic-class-specific event data structures. Each substructure also contains a `uint32` "how many" field that gives the number of elements in the event and timestamp arrays. The "how many" field can contain 0, in which case the two pointers will contain NULL.

The event data structures used in these arrays are the same ones used to send information in a normal event broker event frame. For example, the substructure for generic class 0 (control pads) contains a pointer to an array of `ControlPadEventData` structures, and the substructure for generic class 8 (analog joysticks) contains a pointer to an array of StickEventData structures.

Empty Generic Class Substructures

If there are no devices of a particular generic class connected to the 3DO unit, the substructure for this generic class can be left empty. In this case, the "how many" field contains 0, and the two array pointers will contain NULL. For example, if the 3DO unit were started up without a control pad attached, the first portion of the pod table might be as shown in Figure 2.

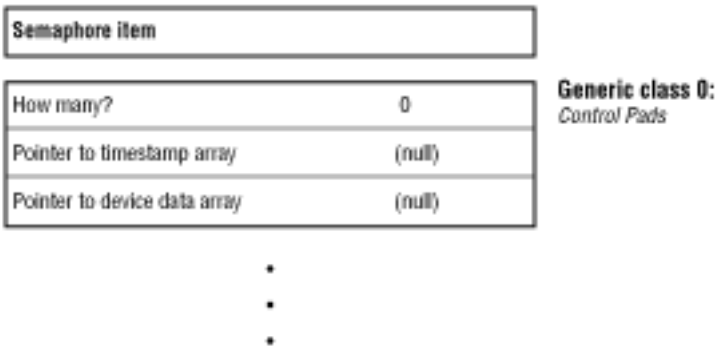

Figure 2: *Pod substructure.*

Non-Empty Generic Class Substructures

If one or more devices of a particular generic class are connected to the 3DO system, the pod table reflects this. The "how many" field is nonzero, and the array pointers are non-NULL and point to valid arrays. Figure 3 shows an example of the generic class substructure.

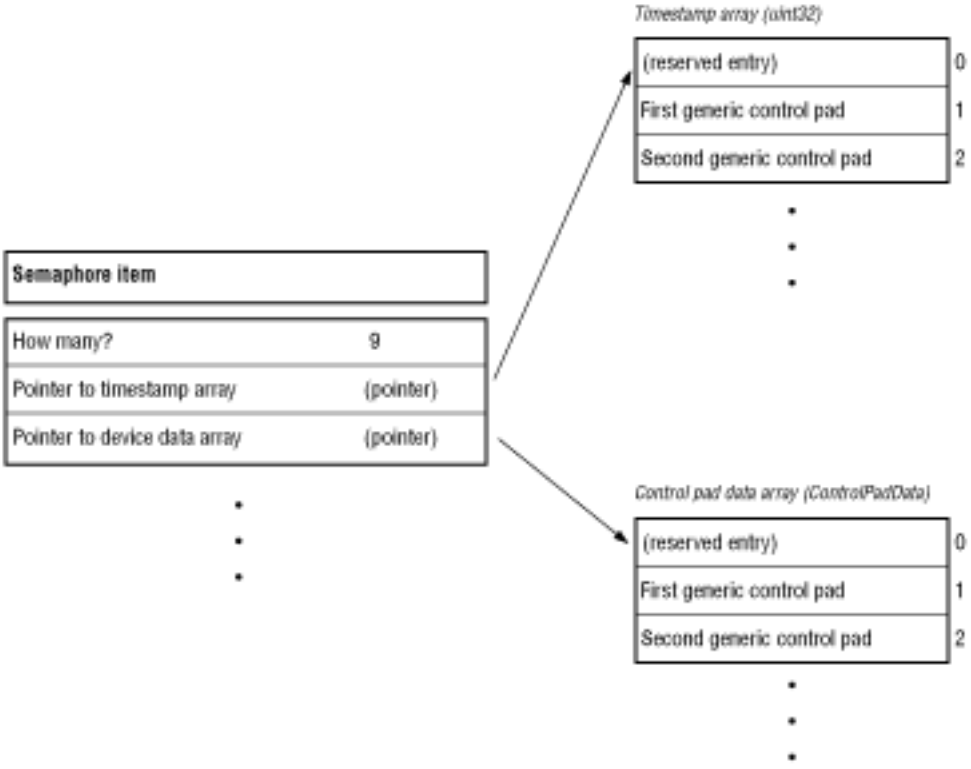

Figure 3: *Generic class substructure.*

The "how many" field does not necessarily contain the number of devices of a generic class that are currently attached to the system; rather, this field identifies the sizes of the data arrays. Normally, the event broker will reserve sufficient space in the arrays for up to eight devices in each class, and the "how many" field will reflect this fact. If your application needs to get a complete and accurate listing of the devices currently attached to the system, it should use the `EB_DescribePods` message to ask the event broker for a current list.

The entries in the timestamp and device-data arrays should be accessed with a starting index of 1. That is, the data for the first generic joystick will be found in `StickEventData[1]`. This ensures that the numbering scheme for these arrays matches the generic-device numbers in the event broker's event-frame and pod-description messages. Entry 0 in each array is reserved for future use, and should not be accessed by applications.

The values in the timestamp arrays are standard Portfolio VBL counter timestamps. They have a resolution of 1/60th of a second on NTSC systems and 1/50th of a second on PAL systems. They are derived from the same source as the event timestamp values in an event broker event message, and also correspond to the VBL-count times applications read using the timer device.

If a controller is unplugged from the control port, the event broker ceases updating its entries in the pod table. An application can detect stale information by checking to see if the validity timestamp is changing.

A substructure goes from empty to nonempty if a device of an as yet unseen generic class is connected to the multiplayer, and a driver for this device is available. The array pointers in a substructure can be changed if the event broker needs to enlarge the arrays to support a larger number of devices than expected. In any case, a change in the table's structure will occur only when the event broker has locked the semaphore on the pod table. Applications are urged to lock the semaphore before accessing the contents of the table, and unlock it immediately afterwards, to avoid having data in the table change while the application is reading it.

Use and Abuse of the Pod Table

It is easy to misuse the pod table, which can hurt the performance and reliability of your application, rather than helping it.

The pod table mechanism is intended to give applications a way to view the most recent information for each control port device on the system. As such, it lets you track a light gun's target position or the offset of an analog joystick with very little effort and overhead. You may be tempted to use it for other, less appropriate purposes: for example, to detect button-down and button-up interactions at the control pads or light gun or analog joysticks. *Do not do this.* It can cause compatibility problems, now or in the future.

You can also get into trouble because an application has no guaranteed way of knowing just when the data in the pod table is likely to be changed. If somebody taps and releases a button on the control pad very quickly, it is possible for the pod table to be updated twice (reflecting both the button-down and button-up changes) before your application gets around to polling it. As a result, your application will never notice this particular button-press, and it will not react to the user's action.

Pod Table Polling

You may think that you can avoid this problem by polling the pod table frequently, perhaps once per field (60 times a second on an NTSC machine). That might work today, but it might not work tomorrow. Future versions of the 3DO hardware and software may scan the control port more frequently than today's version to reduce the delay between the time a button is pressed and the time a program reacts to it. In fact, this is very likely. If your application depends on polling the pod table more rapidly than the event broker can update it, it will probably not run reliably on some future 3DO hardware.

Polling the pod table frequently adds quite a bit of overhead and complexity to an application. You will either have to interrupt your main-line code several times per field to poll the table (making the code bigger and bulkier) or it will be necessary to have a code thread wake up every few milliseconds to check the table (thus adding more overhead than the pod table mechanism eliminated, and slowing down your application's frame rate). You might be tempted to start writing "busy/wait" polling loops to wait for events; these would make your program multitasking-hostile and network-unfriendly.

The event broker's event message system was designed specifically to eliminate such problems. The event broker is a couple of steps closer to the hardware than your application is, and takes advantage of this fact. It does all of the button-down/button-up detection for you. Every time new data arrives from the control port it can queue up several event messages in a row for your application without losing any data, and it will wake up your program or thread (sending a message) when an event arrives.

The best way to use the pod table is to use it *only* to keep track of position information (analog joystick-position, mouse offset, light gun target pulse count, and so on.). For all real user-interaction events (button-down, trigger pulled, and so on), use the event broker event message mechanism.

Using the Pod Table Semaphore

When you lock the semaphore on the pod table before accessing the table, be sure to unlock it as quickly as possible. If left locked, you will lock out the event broker and can cause your application to miss events. Ideally, you should lock the semaphore, copy the information you need from the pod table to a variable in your program's own storage, and then unlock the semaphore immediately.

It is tempting to not lock the semaphore and read the tables directly. This is dangerous, as you might get inconsistent data within the table. It might also lead to crashes if new controllers are plugged into the system and the event broker must alter table pointers.

Finally, since the array pointers within the pod table can change from event to event, it is important not to cache these values anywhere. Whenever you lock the pod table semaphore, you must sample the array pointers from the table.

Synchronization Between the Pod Table and Event Messages

The event frames in an event broker message have timestamps associated with each event, as do the arrays in the pod table. Both timestamps are derived from the same source (the timer device VBL counter) and so are consistent with one another.

However, if you are checking timestamps in the event messages, and are also looking at timestamps in the pod table, you should be cautious. It is possible for you to receive an event message (containing a timestamped event), look in the pod table, and find that the timestamp for the corresponding device does not match the one in the event message. This can (and often will) happen because the pod table was updated between the time that the event message was sent to your application, and the time that your application processed it. The pod table timestamp will usually be more recent than the event frame timestamp.

This might cause some confusion if, for example, an application tracking the light gun cursor based on pod table information but the application was "firing bullets" based on the event messages. The application might mistakenly shoot a bullet at the position the gun was aimed before or after the moment the trigger was pulled, rather than where it was aimed at the precise moment of firing.

All of the information in any particular event frame has the same timestamp. For example, if your application receives a "light gun button down" event frame, then the button-bits and the light gun counter in that event frame were sampled at exactly the same moment. Your application should fire at the spot indicated in the event frame, rather than at the (possibly different) location currently indicated in the pod table.

Reconfiguring or Disconnecting a Task

As a task works with the event broker, it may want to reconfigure itself to monitor different event types, change its focus dependence, reset the size of its event queue, or use a new reply port for event broker messages. Or it may want to completely disconnect itself from the event broker so that it no longer monitors events. In each case, the task sends a new configuration message to the event broker.

Reconfiguring the Event Broker Connection

The task initializes a new `ConfigurationRequest` structure with the new settings it wants. It then sends an `EB_Configure` message to the event broker to have the new settings take effect. The event broker uses the reply port of the `EB_Configure` message to determine which tasks are connected to the event broker. When changing the configuration, the application should use the same reply port for the message as it did when it first connected to the event broker.

If a task wants to change the reply port it uses to receive event broker messages, it must first disconnect from the event broker using a configuration message with its old reply port. Then the task must create a new configuration message with a new reply port, and send the new configuration message to the event broker. The task must also close the old configuration message and old reply port.

Disconnecting From the Event Broker

If the task wishes to disconnect itself completely from the event broker, it sends a configuration request, using its current reply port, and specifies a 0 for both the trigger and capture masks. The event broker then disconnects the task. A task can also call `DeleteMsgPort()` on its current reply port. On the next tick, the event broker will disconnect the task.

Other Event Broker Activities

Although the event broker's primary activity is monitoring pod table events for listeners, it performs other activities such as checking on connected listeners and pods, finding out where the focus lies, and controlling individual pods, such as stereoscopic glasses and audio controllers.

Getting Lists of Listeners and Connected Pods

There are many times when a task needs a list of pods connected to the control port or listeners connected to the event broker. There are two flavors of messages that retrieve this kind of information: `EB_DescribePods` asks for a list of connected pods; `EB_GetListeners` asks for a list of connected listeners.

Asking for a List of Connected Pods

When a task uses an `EB_DescribePods` message to ask the event broker for a list of connected pods, the message data block contains only the EventBrokerHeader data structure specifying the flavor `EB_DescribePods`. The message sent to the event broker should be a buffered message, created with `CreateBufferedMsg()`, which contains sufficient room to hold the complete list of connected pods.

Reading a Returned List of Connected Pods

When the event broker replies to an `EB_DescribePods` message, it turns the message into an `EB_DescribePodsReply` message. The message's data buffer contains a list of pods currently attached to the control port. Those pods are described in the data structure PodDescriptionList, which is defined as follows:

```
typedef struct               PodDescriptionList
{
  EventBrokerHeader           pdl_Header;
  int32                       pdl_PodCount;
  PodDescription              pdl_Pod[1];
} PodDescriptionList;
```

- `pdl_Header` is an EventBrokerHeader data structure set to the message flavor `EB_DescribePodsReply`.
- `pdl_PodCount` contains the number of pods described in the next field.
- `pdl_Pod[]` is an array of pod descriptions, with one element for each pod currently attached to the control port. Each pod description is contained in the data structure PodDescription, defined as follows:

```
typedef struct PodDescription
{
  uint8           pod_Number;
  uint8           pod_Position;
  uint8           rfu[2];
  uint32          pod_Type;
  uint32          pod_BitsIn;
```

```
  uint32          pod_BitsOut;
  uint32          pod_Flags;
  uint8           pod_GenericNumber[16];
  Item            pod_LockHolder;
} PodDescription;
```

- `pod_Number` gives the unique and unchanging pod number of the pod.
- `pod_Position` gives the position of the pod in the control port daisy chain. It is an integer from 1 to the total number of pods on the control port.
- `pod_Type` lists the native pod type of the pod, that is, the specific type of device attached to the daisy chain (not the generic type).
- `pod_BitsIn` and `pod_BitsOut` give the number of bits shifted into the pod during the control port data stream for each video field, and the number of bits shifted out of the pod during each data stream.
- `pod_Flags`, contains flags that show the generic device type (or types) that the pod contains. These flags, shown in Table 4, occupy the left-most bits of the `pod_Flags` word.

  Table 1: *Flag constants used for the pod_Flags field.*

Flag Constant	Hex Value	Count From Left-Most Bit
POD_IsControlPad	0x80000000	0
POD_IsMouse	0x40000000	1
POD_IsGun	0x20000000	2
POD_IsGlassesCtlr	0x10000000	3
POD_IsAudioCtlr	0x08000000	4
POD_IsKeyboard	0x04000000	5
POD_IsLightGun	0x02000000	6
POD_IsStick	0x01000000	7
POD_IsIRController	0x00800000	8

- `pod_GenericNumber` is an array of 16 unsigned 8-bit values. These values correspond to the bits of the `pod_Flags` field: `pod_GenericNumber[0]` corresponds to flag bit 0, `POD_IsControlPad`; `pod_GenericNumber[1]` corresponds to the first flag bit, `POD_IsMouse`; and so on. The value in each element of the array stores the rank of this pod in the order of all other identical generic devices connected to the serial port. For example, if `pod_GenericNumber[2]` contains a 4, then the pod contains a generic gun that is the fourth generic gun connected to the serial port.

- `pod_LockHolder` gives the item number of the task that has this pod locked for its exclusive use. If this field is set to 0, then the pod is unlocked. Pod locking is not currently implemented, so this field should not be used.

Asking for a List of Connected Listeners

When a task asks for a list of connected listeners, it uses an `EB_GetListeners` message, whose data block contains only the EventBrokerHeader data structure specifying the flavor `EB_GetListeners`. Messages used for this purpose must be created with `CreateBufferedMsg()` in the same way as messages used with the `EB_DescribePods` command. `CreateBufferedMsg()` is described in Creating a Configuration Message.

Reading a Returned List of Connected Listeners

When the event broker replies to an `EB_GetListeners` message, it turns the message into an `EB_GetListenersReply` message. The message buffer contains a list of listeners currently connected to the event broker. The list is contained in the data structure ListenerList, defined as follows:

```
typedef struct ListenerList
{
  EventBrokerHeader                          ll_Header; /* { EB_GetListenersReply } */
  int32                        ll_Count;
  struct {
    Item                        li_PortItem;
    enum ListenerCategory                          li_Category;
  } ll_Listener[1];
} ListenerList;
```

- `ll_Header` is an `EventBrokerHeader` structure set to the message flavor `EB_GetListenersReply`.
- `ll_Count` contains the number of listeners described in the next field.
- `ll_Listener` is an array of listener descriptions with one element for each listener currently connected to the event broker. Each listener element contains these fields:
    - `li_PortItem` contains the item number of the listener's reply port
    - `li_Category` gives the listener's focus-interest category

Working With Input Focus

It is often important for a listener to know which task currently has the input focus or to be able to change the input focus from one task to another without user intervention. The event broker can handle both of these requests.

Finding the Current Focus Holder

To find the current focus holder, a listener can inquire with an `EB_GetFocus` message. Its data block is an `EventBrokerHeader` data structure set to the message flavor `EB_GetFocus`.

When the event broker receives an EB_GetFocus message, it gets the item number of the task currently holding the focus, writes that value in the error field of the message, and returns the message to the task. The task receiving the message reads the current focus holder's item number from the error field. The error field will contain a negative number (an error code) if the operation failed.

Setting the Current Focus Holder

Warning: Although the event broker currently allows any user task to set the current focus holder, in the future this request may restricted to only privileged tasks.

To move the focus from one task to another without input from the user, a listener can use an EB_SetFocus message. The data block for this message uses the SetFocus data structure, defined as follows:

```
typedef struct SetFocus
{
  EventBrokerHeader sf_Header; /* { EB_SetFocus } */
  Item            sf_DesiredFocusListener;
} SetFocus;
```

- sf_Header is an EventBrokerHeader data structure set to the message flavor EB_SetFocus.
- sf_DesiredFocusListener is the item number of the task to which the event broker should give the focus.

When the event broker receives the message, it changes the input focus to the requested task and writes that task's item number into the error field of the message. It returns the message to the sending task. That task can now read the error field to get the item number of the focus-holding task (if successful) or an error code (if unsuccessful).

Commanding a Pod

Sometimes a task may want to control a pod through the event broker. For example, a task may wish to alternate lens opaqueness in a pair of stereoscopic glasses, or mute the sound coming through an audio controller. To issue commands to a pod, the task uses a message of the flavor EB_IssuePodCmd. It accompanies the message with a data block that uses the PodData data structure, defined as follows:

```
typedef struct PodData
{
  EventBrokerHeader                    pd_Header;
  int32             pd_PodNumber;
  int32             pd_WaitFlag;
  int32             pd_DataByteCount;
  uint8             pd_Data[4];
} PodData;
```

- pd_Header is an EventBrokerHeader data structure set to the message flavor EB_IssuePodCmd.
- pd_PodNumber gives the pod number of the pod to which the command is sent.
- pd_WaitFlag can be set to either 1 or 0. If set to 1, it asks the event broker to send the command to the pod and

reply immediately. If set to 0, it asks the event broker to send the command to the pod, wait for the command to finish execution, and then reply to command message.

- `pd_DataByteCount` gives the size in bytes of the `pd_Data` field that follows.

- `pd_Data` is an array of bytes that contains the command sent to the pod. The first element of the array (element 0) contains the generic device class of the pod to which the command is sent; the second element (element 1) contains the command subcode, which is specific to the generic class in the first element. If the command defined by these two bytes requires extra data, that data goes into the elements of the array listed below.

Generic Device Class

The include file *event.h* currently defines the following constants to specify generic device classes:

- `GENERIC_ControlPad` - a controller pad.
- `GENERIC_Mouse` - a mouse.
- `GENERIC_Gun` - a photo-optic gun.
- `GENERIC_GlassesCtlr` - a stereoscopic glasses controller.
- `GENERIC_AudioCtlr` - an audio controller.
- `GENERIC_Keyboard` - a keyboard.
- `GENERIC_LightGun` - a light gun.
- `GENERIC_Stick` - a joystick.
- `GENERIC_IRController` - a infrared controller pad (wireless).

The array element `pd_Data[0]` should hold the constant appropriate to the commanded pod.

Audio Controller Subcodes

The audio controller accepts commands defined by audio controller subcodes. At present, Portfolio offers one audio subcode, specified by the following constant:

- `GENERIC_AUDIO_SetChannels` controls the output of the two stereo audio channels.

This subcode goes in the array element `pd_Data[1]`. It requires one byte of data following it in `pd_Data[2]`. This data specifies how the stereo channels are controlled. The four options, defined by the following constants, are:

- `AUDIO_Channels_Mute` - turns off the audio output in both channels.

- `AUDIO_Channels_RightToBoth` - feeds the right audio signal to both left and right channels.

- `AUDIO_Channels_LeftToBoth` - feeds the left audio signal to both the left and right channels.

- `AUDIO_Channels_Stereo` - feeds the left audio signal to the left channel and right audio signal to the right channel.

Glasses Controller Subcodes

The stereoscopic glasses controller accepts commands defined by glasses controller subcodes. At present, Portfolio offers one

glasses subcode, specified by the following constant:

- `GENERIC_GLASSES_SetView` - controls the opacity of each lens in a pair of stereoscopic glasses.

This subcode goes into the array element `pd_Data[1]`. It requires two bytes of data following it in `pd_Data[2]` and `pd_Data[3]`. The first of the two data bytes controls the left lens of the glasses, the second byte controls the right lens. The four possible values for each lens, defined by the following constants, are:

- `GLASSES_AlwaysOn` - keeps the lens clear during both odd and even video fields.
- `GLASSES_OnOddField` - turns the lens clear every odd video field, and turns it opaque every even field.
- `GLASSES_OnEvenField` - turns the lens clear every even video field, and turns it opaque every odd field.
- `GLASSES_Opaque` - turns the lens opaque during both odd and even video fields.

Lightgun Controller Subcode

Currently, Portfolio offers one light gun controller subcode, which is specified by the following constant:

- `GENERIC_LIGHTGUN_SetLEDs` controls the setting of LEDs on the light gun. The setting of up to eight LEDs is supported, although the stock light gun has only two.

The possible options, defined by the following constants, are:

- `LIGHTGUN_LED1` - controls the setting of the first LED on the light gun.
- `LIGHTGUN_LED2` - controls the setting of the second LED on the light gun.

The Event Broker's Reply to a Command

After the event broker processes a pod command message, it returns the message to the sending task to report on the command execution. The returned message is of the flavor `EB_IssuePodCmdReply`, and contains a reply portion that is a negative number if there was an error executing the command, or is 0 or a positive number if the command execution was successful.

Event Broker Convenience Calls

Portfolio provides a set of convenience calls that allow a task to easily monitor events on controller pads or mice.

Note: The convenience calls are "single threaded." Only one thread within any application can use them at a time.

Connecting to the Event Broker

This call connects the task to the event broker, sets the task's focus interest, and determines how many controller pads and mice the task wants to monitor:

```
Err InitEventUtility(int32 numControlPads, int32 numMice,int32
focusListener);
```

Where:

- `numControlPads` sets the maximum number of controller pads the task wants to monitor.
- `numMice` sets the maximum number of mice the task wants to monitor. These values should be an integer of 0 or higher.
- `focusListener` is a Boolean value that sets the focus of the task when it is connected as a listener. If the parameter is true (nonzero), the task is connected as a focus-dependent listener. If the parameter is false (0), the task is connected as a focus-independent listener.

When the call executes, it creates a reply port and a message, sends a configuration message to the event broker that asks the event broker to report appropriate mouse and controller pad events, and deals with the event broker's configuration reply. The call returns 0 if successful, and a negative number (an error code) if unsuccessful.

Monitoring a Control Pad or a Mouse

This call specifies a controller pad or mouse to monitor, specifies whether the event broker should respond immediately or wait until something happens on the controller pad, and provides a data structure for data from the controller pad:

```
Err GetControlPad(int32 padNumber, int32 wait,
ControlPadEventData *data);
```

- `padNumber` sets the number of the generic controller pad on the control port (i.e., the first, second, third, and so on, pad in the control port daisy chain, counting out from the 3DO unit) that the task wants to monitor. The first pad is 1, the second is 2, and so on.

- `wait` is a Boolean value that specifies the event broker's response to this call. If it is true (nonzero), the event broker waits along with the task until an event occurs on the specified pad, and returns data only when there is a change in the pad. If it is false (0), the event broker immediately returns with the status of the pad.

- `data` is a pointer to a ControlPadEventData structure that shows where memory was allocated for the returned control pad data to be stored. (The task must create an instance of this data structure before it executes `GetControlPad()`).

When the call executes, it either returns immediately with event information, or waits to return with event information until there is a change in the controller pad. It returns a 1 if an event has occurred on the pad, a 0 if no event has occurred on the pad, or a negative number (an error code) if there was a problem retrieving an event. If an event has occurred, the task must check the ControlPadEventData data structure for details about the event.

This call performs the same function as `GetControlPad()`, but gets events from a specified mouse instead of a specified controller pad:

```
Err GetMouse(int32 mouseNumber, int32 wait, MouseEventData
*data);
```

Its parameters are the same as those for `GetControlPad(),` but `mouseNumber` specifies a mouse on the control port instead of a controller pad, and the pointer data is now a pointer to a data structure for storing mouse event data instead of controller pad event data.

Disconnecting From the Event Broker

This call disconnects a task that was connected to the event broker with the `InitEventUtility()` call:

```
Err KillEventUtility(void);
```

When executed, it disconnects the task from the event broker, closes the reply port, and frees all resources used for the connection. It returns 0 if successful, and a negative number (error code) if

unsuccessful.

Function Calls

These convenience calls allow a task to use the event broker to monitor controller pads and mice.

Connecting To and Disconnecting From the Event Broker

- `InitEventUtility()` Connects task to the event broker.
- `KillEventUtility()` Disconnects a task from the event broker.

Monitoring Events

- `GetControlPad()` Gets control pad events.
- `GetMouse()` Gets mouse events.

InitEventUtility

Connects task to the event broker.

Synopsis

```
Err InitEventUtility( int32 numControlPads, int32 numMice,int32
focusListener )
```

Description

This convenience call allows a task to easily monitor events on controller pads or mice. This function connects the task to the event broker, sets the tasksfocusinterest,and'determines how many controller pads and mice the task wants to monitor.

The function creates a reply port and a message, sends a configuration message to the event broker (which asks the event broker to report appropriate mouse and controller pad events), and deals with the event brokersconfigurationreply.'

Arguments

numControlPads
: The number of controller pads to monitor.

numMice
: The number of mice to monitor.

focusListener
: The focus of the task when it is connected as a listener. If the value is nonzero, the task is connected as a focus-dependent listener. If the value is zero, the task is connected as a focus-independent listener.

Return Value

This function returns a value greater than or equal to 0 if all went well or a negative error code if an error occurred.

Implementation

Convenience call implemented in input.lib V20.

Associated Files

event.h, input.lib

See Also

GetControlPad(), GetMouse(), KillEventUtility()

GetControlPad

Gets control pad events.

Synopsis

```
Err GetControlPad( int32 padNumber, int32 wait,ControlPadEventData
*data )
```

Description

This is a convenience call that allows a task to easily monitor events on controller pads. This function specifies a controller pad to monitor, specifies whether the routine should return immediately or wait until something happens on the controller pad, and provides a data structure for data from the controller pad.

When the function executes, it either returns immediately with event information or waits until there is a change in the controller pad before returning. If an event has occurred, the task must check the ControlPadEventData data structure for details about the event.

Arguments

padNumber

Sets the number of the generic controller pad on the control port (i.e., the first, second, third, and so on pad in the control port daisy chain, counting out from the 3DO unit) that the task wants to monitor. The first pad is 1, the second is 2, and so on.

wait

A boolean value that specifies the event brokersresponse.IfitisTRUE(a nonzero value), the event broker waits along with the task until an event occurs on the specified pad and only returns with data when there is a change in the pad. If it is FALSE
(zero), the event broker immediately returns with the status of the pad.

data

A pointer to a ControlPadEventData data structure for the returned control pad data to be stored.

Return Value

This function returns 1 if an event has occurred on the pad, 0 if no event has occurred on the pad, or a

negative number (an error code) if a problem occurred while retrieving an event.

Implementation

Convenience call implemented in input.lib V20.

Associated Files

event.h, input.lib

See Also

InitEventUtility(), GetMouse(), KillEventUtility()

GetMouse

Gets mouse events.

Synopsis

```
Err GetMouse( int32 mouseNumber, int32 wait,MouseEventData *data )
```

Description

This convenience call allows a task to easily monitor mouse events.

This function is similar to `GetControlPad()` but gets events from a specified mouse instead of a specified controller pad. It specifies a mouse to monitor, specifies whether the call should return immediately or wait until something happens on the mouse, and provides a data structure for data from the mouse.

When the function executes, it either returns immediately with event information or waits there is a change in the mouse before returning. If an event has occurred, the task must check the MouseEventData data structure for details about the event.

Arguments

mouseNumber

Sets the number of the generic mouse on the control port (i.e., the first, second, third, and so on mouse in the control port daisy chain, counting out from the 3DO unit) that the task wants to monitor. The first mouse is 1, the second is 2, and so on.

wait

A boolean value that specifies the event brokersresponse.IfitisTRUE(a non zero value), the event broker waits along with the task until an event occurs on the specified pad and only returns with data when there is a change in the pad. If it is FALSE
(zero), the event broker immediately returns with the status of the pad.

data

A pointer to a MouseEventData data structure that contains the mouse data that is returned.

Return Value

This function returns 1 if an event has occurred on the mouse, 0 if no event has occurred on the mouse, or an error code (a negative value) if an error occurred when attempting to retrieve an event.

Implementation

Convenience call implemented in input.lib V20.

Associated Files

event.h, input.lib

See Also

`InitEventUtility()`, `GetControlPad()`, `KillEventUtility()`

KillEventUtility

Disconnects a task from the event broker.

Synopsis

```
Err KillEventUtility( void );
```

Description

This is a convenience function that allows a task to easily monitor events on controller pads or mice. This function disconnects a task that was connected to the event broker with the `InitEventUtility()` function.

When executed, the function disconnects the task from the event broker, closes the reply port, and frees all resources used for the connection.

Return Value

This function returns a value greater than or equal to 0 if all went well or a negative error code if an error occurs.

Implementation

Convenience call implemented in input.lib V20.

Associated Files

event.h, input.lib

See Also

`GetControlPad()`, `GetMouse()`, `InitEventUtility`s()

I/O Architecture

Portfolio provides services to access hardware devices in a hardware-independant manner. This allows applications to remain compatible across a wide range of hardware products. The Portfolio I/O model is a central component that enables a wide range of exotic devices to coexist and evolve for many years.

The Portfolio I/O model is inherently asynchronous. Lengthy requests made by a task to a hardware device are handled in the background. The client task receives a notification when the request has been satisfied. This architecture allows a single task to have multiple outstanding requests pending on different hardware devices, which tends to maximize the use of the system bandwidth by keeping all subsystems working.

Imagine a task that needs to read a block of data from a CD-ROM. It sends a request to the CD-ROM device driver. The driver does the setup necessary to initiate a hardware transfer. The CD-ROM mechanism is then activated and the needed block of data is transferred to memory. When the transfer is complete, an interrupt occurs that wakes up the driver. The driver then notifies the client task that the data it needs has arrived. While the hardware is servicing the request from the task, the task is free to do other activities, such as playing music.

The Portfolio I/O model relies on three basic abstractions: the *device*, the *driver*, and the *I/O request.*

Devices

A software device may or may not correspond to a hardware device. Technically, a software device is anything that responds to I/O requests and lives in supervisor space. A software device might correspond directly to a hardware device such as a timer, a CD-ROM player, or a controller pad; it might correspond to a port or bus that controls many different devices; or it might correspond to a more abstract entity such as an open file on a CD-ROM disk. Portfolio treats a software device as a single I/O mechanism, no matter what the device's correspondence to actual hardware.

Note: Whenever you read the term "device" by itself, it refers to a software device. A hardware device is always referred to as "hardware device."

Portfolio maintains a list of all devices resident or active on the 3DO unit. This list changes as devices are added to or taken from the system. Each device in the list has an item number; a task that requests I/O with a device must refer to the device by its item number. To find a device's item number, a task uses the call `OpenNamedDevice(),` where the task names the device it wants and the kernel returns the device's item number.

Drivers

Every device maintained by the kernel must have a corresponding driver before any task can use the device. The driver accepts and acts on every task request for I/O. The driver must be able to recognize at least three I/O commands-`CMD_READ, CMD_WRITE`, and `CMD_STATUS`-and may recognize more device-specific I/O commands as well. Not all devices are writable, so not all devices can perform the `CMD_WRITE` command. For example, a CD-ROM drive will respond to the command without writing data. All devices can execute `CMD_STATUS` commands and tell an enquiring task their current status.

A single driver can control several identical or very similar components. Each of these components is called a *unit.* Each unit may respond to slightly different commands or may respond in slightly different ways. Even so, the units are similar enough that a single driver can control them. When a task requests I/O through such a driver, it specifies not only the device item number, but the unit number. For example, the kernel provides a timer device that supports two timer units: the microsecond timer and the vertical blank timer. Both timer units respond to the same commands, but one unit returns elapsed microseconds, while the other returns elapsed vertical blanks. To read the timer, a task specifies the timer item number, and then asks for either the microsecond or vertical blank unit.

Drivers are all privileged code. The 3DO Company provides them for 3DO-approved hardware devices such as controller pads and for more abstract system software devices such as open files.

I/O Requests (IOReqs)

An I/O request (IOReq for short) is a data structure that the kernel passes back and forth between a task and a device. Think of the IOReq as a messenger that goes from task to device, passing along an I/O command and other I/O information. When an I/O operation is finished, the device returns the IOReq to the task, passing along any reports it has to make on the I/O operation.

Before a task can communicate with a device, the task must create an IOReq using the `CreateIOReq()` call. An IOReq is a system item that must be used to communicate with a device.

Within an IOReq lies information necessary for executing an I/O command. Part of that information is the IOInfo data structure, which contains an I/O command, a unit specification, pointers to read and write buffers, and other I/O parameters provided by the task requesting I/O.

To initiate an I/O operation, a task must give the IOReq to a device by calling `SendIO()` or `DoIO()`. When the device completes the operation, it notifies the calling task that the operation is complete. The calling task is then free to use the IOReq to initiate further I/O operations.

A task gets notified that an I/O operation is complete by either receiving a signal or receiving a message.

The notification mechanism is determined when the IOReq is created.

Performing I/O

From a task's point of view, the process of I/O between a task and a device can be broken down into four stages:

- **Preparing for I/O**. Task opens a device, creates one or more IOReq data structures, and one or more IOInfo data structures.
- **Sending an IOReq**. Task fills in IOInfo values, writes output data into a write buffer (if appropriate), submits the task's IOInfo values to the kernel for processing.
- **Receiving an IOReq**. Task receives notification that an I/O operation is complete, reads data about the operation within a returned IOReq, and (if appropriate) reads data from a read buffer.
- **Finishing I/O**. Task finishes exchanging IOReqs with a device, deletes the IOReqs it created earlier, and closes the device.

If a task wants to perform a series of I/O operations on a single device, it only needs to prepare for I/O once. The task can then send and receive IOReqs to and from the device until it is finished.

Preparing for I/O

Before a task can send and receive IOReqs, it must prepare for I/O by opening a device and creating IOReq and IOInfo data structures.

To open a device, a task uses the following function call:

```
Item OpenNamedDevice( const char *name, void *args )
```

The `name` argument is a pointer to a null-terminated string of characters that contains the name of the device. The `a` argument is a pointer reserved for future use. Currently, this value must always be NULL.

When `OpenNamedDevice()` executes, it opens the named device and returns the item number of that device. Future I/O calls to the device use the device's item number.

Creating an IOReq

Once a task opens a device, the task must create one or more IOReq items to communicate with the device. Before creating an IOReq, it's important to consider how the task will be notified when the IOReq is returned. There are two options:

- Notification by signal
- Notification by message

Notification by signal is the default method. When the device finishes with an I/O operation and wants to return the IOReq to the task, it sends the system signal `SIGF_IODONE` to the task. The task then knows there's a completed IOReq, and it can retrieve the IOReq when it wants to.

Notification by message is specified by providing a message port when an IOReq is created. When the device finishes with an I/O operation and wants to return the IOReq, it sends a message to the specified port. The task can read the message to get a pointer to the IOReq and then read or reuse the IOReq when it wants to.

Either notification method has advantages and disadvantages. Notification by signal uses fewer resources, but doesn't identify the returning IOReq. It merely says that an IOReq has returned. Notification by signal is useful for I/O operations that use a single IOReq passed back and forth between the task and a device.

Notification by message uses more resources, but because each message identifies a particular IOReq, the task knows exactly which IOReq is returned when it receives notification. Notification by message is useful for I/O operations that use more than one IOReq.

Once a task opens a device, the task creates an IOReq data structure using this call:

```
Item CreateIOReq( const char *name, uint8 pri, Item dev, Item mp )
```

The `name` argument is a pointer to a null-terminated character string containing the name of this IOReq. If the IOReq is unnamed, this argument should be NULL. The `pri` argument is a priority value from a low priority of 0 to a high priority of 255. The `dev` argument is the device number of the opened device to which this IOReq is to be sent.

The `mp` argument is the item number of a message port where the task is notified of the IOReq's completion. If the task wants to be notified of IOReq completion by message, it must create a message port and supply its item number here. If the task wants only to be notified of IOReq completion by signal, then this argument should be set 0.

When this call executes, the kernel creates an IOReq that is sent to the specified device and returns the item number of the IOReq. All future I/O calls using this IOReq specify it with its item number.

Initializing an IOInfo Structure

Although a task can create an IOReq, it can't directly set values in it because an IOReq is an item. The task instead defines an IOInfo data structure in its own memory, and then, when it requests an I/O operation, the task fills in the appropriate values and passes the IOInfo on to the kernel with the IOReq.

The definitions of an IOInfo data structure and the IOBuf data structure used within it (both defined in *io.h*) are as follows:

```
typedef struct IOBuf
{
    void        *iob_Buffer;                /* ptr to users buffer */
    int32         iob_Len;                  /* len of this buffer, or transfer
size*/
} IOBuf;

typedef struct IOInfo
{
    uint8        ioi_Command;               /* Command to be executed */
    uint8        ioi_Flags;                 /* misc flags */
    uint8        ioi_Unit;                  /* unit of this device */
    uint8        ioi_Flags2;                /* more flags, should be set to zero */
    uint32        ioi_CmdOptions;           /* device dependent options */
    uint32        ioi_User;                 /* for user use */
```

```
    int32       ioi_Offset;                 /* offset into device for transfer to
*/
                                            /*begin */
    IOBuf       ioi_Send;                   /* copy out information */
    IOBuf       ioi_Recv;                   /* copy in info, (address validated) */
} IOInfo;
```

The possible values for the IOInfo fields depend on the device to which the IOReq is sent. The fields include the following:

- **ioi_Command** carries the I/O command to be executed by the device driver. Every driver can recognize at least three commands defined in *io.h*: `CMD_WRITE`, `CMD_READ`, and `CMD_STATUS`.
- **ioi_Flags** contains flags for I/O operation variations. Portfolio currently defines just one flag: `IO_QUICK`, which asks that I/O take place immediately (synchronously) without notification. All other bits must be set to 0.
- **ioi_Unit** specifies the unit of the device to communicate with. Available units change from device to device. The timer, for example, offers the `TIMER_UNIT_USEC` unit and the `TIMER_UNIT_VBLANK` unit.
- **ioi_Flags2** currently is not used and is always set to 0.
- **ioi_CmdOptions** sets command options that vary from device to device.
- **ioi_User** contains a pointer to anything the user task wishes to point to. This is a handy field for keeping a reminder of the data on which an I/O operation is working. When the I/O operation completes, you can then look in the `io_Info.ioi_User` field of the IOReq structure to retrieve this pointer.
- **ioi_Offset** contains an offset into the device-a location-where the I/O operation begins. For output operations, this offset defines the starting point in the device where the task's data is written. For input operations, this offset defines the starting point in the device where data is read. This value, when supported by a device, uses the units of block size for whatever device it's addressing.
- **ioi_Send** contains the pointer to and size of a write buffer in which the task stores output data to be written to the device.
- **ioi_Recv** contains the pointer to and size of a read buffer in which the driver stores input data read from the device.

The IOInfo structure should always contain a command value in `ioi_Command`, whether it's one of the three standard commands-`CMD_READ`, `CMD_WRITE`, or `CMD_STATUS`-or a device-specific command. If the command involves writing data, `ioi_Send` should be filled in with a write buffer definition; if the command involves reading data, `ioi_Recv` should be filled in with a read buffer definition. And if the task wants the fastest possible I/O operation with a device that can respond immediately (a timer, for example), `ioi_Flags` should be set to `IO_QUICK`. Any other fields should be filled in as appropriate for the operation.

Passing IOInfo Values to the Device

To pass the IOInfo values to the system and send the IOReq, the task can use this call (found in *io.h*):

```
Err SendIO( Item ioReqItem, const IOInfo *ioiP )
```

The ioReqItem argument is the item number of the IOReq to be sent to the device. The ioiP, argument is a pointer to the IOInfo structure that describes the I/O operation to perform.

When the kernel carries out `SendIO(),` it copies the IOInfo values into the IOReq, then checks the IOReq to be sure that all values are appropriate. If they are, the kernel passes on the I/O request to the device responsible. The device then carries out the request.

`SendIO()` returns 1 if immediate I/O was used and the IOReq is immediately available to the task. `SendIO()` returns a 0 if I/O was done asynchronously, which means that the request is being serviced by the device in the background. `SendIO()` returns a negative error code if there was an error in sending the IOReq. This usually occurs if there were inappropriate values included in the IOInfo structure.

Asynchronous vs. Synchronous I/O

When a task sends an I/O request to a device using `SendIO()`, the device may or may not satisfy the request immediately. If `SendIO()` returns 1, it means that the operation is completed and no other actions are expected. If `SendIO()` returns a 0, it means that the I/O operation has been deferred and is being worked on in the background.

When an operation is deferred, your task is free to continue executing while the I/O is being satisfied. For example, if the CD-ROM device is doing a long seek operation in order to get to a block of data you have asked it to read, you can continue executing the main loop of your task while you wait for the block to be transferred.

When `SendIO()` returns 0, which means the I/O request is being serviced asynchronously, you must wait for a notification that the I/O operation has completed before you can assume anything about the state of the operation. As shown previously, when you create an IOReq using `CreateIOReq()`, you can specify one of two types of notification: signal or message. When an asynchronous I/O operation completes, the device sends your task a signal or a message to inform you of the completion.

Once you receive the notification that an I/O operation is complete, you must call `WaitIO()` to complete the I/O process.

```
Err WaitIO( Item ioreq )
```

`WaitIO()` cleans up any loose ends associated with the I/O process. `WaitIO()` can also be used when you wish to wait for an I/O operation to complete before proceeding any further. The function puts your task or thread on the wait queue until the specified IOReq has been serviced by the device.

The return value of `WaitIO()` corresponds to the result code of the whole I/O operation. If the operation fails for some reason, `WaitIO()` returns a negative error code describing the error.

If you have multiple I/O requests that are outstanding, and you receive a signal telling you an I/O operation is complete, you might need to determine which I/O request is complete. Use the `CheckIO()` function:

```
int32 CheckIO( Item ioreq )
```

`CheckIO()` returns 0 if the I/O operation is still in progress. It returns greater than 0 if the operation completes; it returns a negative error code if something is wrong.

Do not use `CheckIO()` to poll the state of an I/O request. You should use `WaitIO()` if you need to wait for a specific I/O operation to complete, or use `WaitSignal()` or `WaitPort()` if you must wait for one of a number of I/O operations to

complete.

There are many cases where an I/O operation is very short and fast. In these cases, the overhead of notifying your task when the I/O completes becomes significant. The I/O subsystem provides a quick I/O mechanism to help remove this overhead as much as possible.

Quick I/O occurs whenever `SendIO()` returns 1. It tells you that the I/O operation is complete, and that no signal or message will be sent to your task. You can request quick I/O by setting the `IO_QUICK` bit in the `ioi_Flags` field of the IOInfo structure before you call `SendIO()`. `IO_QUICK` is merely a request for quick I/O. It is possible that the system cannot perform the operation immediately. Therefore, check the return value of `SendIO()` to make sure the I/O was done immediately. If it was not done synchronously, you have to use `WaitIO()` to wait for the I/O operation to complete.

The fastest and simplest way to do quick I/O is to use the `DoIO()` function:

```
Err DoIO( Item ioreq, const IOInfo *ioInfo )
```

`DoIO()` works just like `SendIO()`, except that it guarantees that the I/O operation is complete once it returns. You do not need to call `WaitIO()` or `CheckIO()` if you use `DoIO()` on an IOReq. `DoIO()` always requests quick I/O, and if the system is unable to do quick I/O, this function automatically waits for the I/O to complete before returning.

Completion Notification

When an I/O operation is performed asynchronously, the device handling the request always sends a notification to the client task when the I/O operation is complete. As mentioned previously, the notification can be either a signal or a message.

When you create IOReq items with signal notification, the responsible devices send your task the `SIGF_IODONE` signal whenever an I/O operation completes. If you have multiple I/O requests outstanding at the same time, you can use the following to wait for any of the operations to complete.

```
WaitSignal(SIGF_IODONE);
```

When `WaitSignal()` returns, you must call `CheckIO()` on all of the outstanding I/O requests you have to determine which one is complete. Once you find a completed request, call `WaitIO()` with that IOReq to mark the end of the I/O operation.

If you created your IOReq structures with message notification, you will receive a message whenever an I/O operation completes. The message is posted to the message port you specified when you created the IOReq. If you have multiple message-based I/O requests outstanding, you could wait for them using:

```
msgItem = WaitPort(replyPort,0);
```

`WaitPort()` puts your task to sleep until any of the IOReqs complete. Once `WaitPort()` returns, you can look at three fields of the message structure to obtain information about the IOReq that has completed:

```
{
Item msgItem;
Message *msg;

    msg        = (Message *) LookupItem(msgItem);
    ioreq        = (Item) msg->msg_DataPtr;
```

```
    result        = (Err) msg->msg_Result;
    user        = msg->msg_DataSize;
}
```

The `msg_DataPtr` field contains the item number of the IOReq that has completed. The `msg_Result` field contains the result code of the I/O operation. This is the same value that `WaitIO()` would return for this IOReq. Finally, `msg_DataSize` contains the value of the `ioi_User` field from the `IOInfo` structure used to initiate the I/O operation with `SendIO()`.

Reading an IOReq

Once an IOReq returns to a task, the task can read the IOReq for information about the I/O operation. To read an IOReq, a task must get a pointer to the IOReq using the `LookUpItem()` call. For example:

```
    ior = (IOReq *)LookupItem(IOReqItem);
```

Once a task has the address of an IOReq, it can read the values in the different fields of the IOReq. An IOReq data structure is defined as follows:

```
typedef struct IOReq
{
    ItemNode            io;
    MinNode            io_Link;
    struct            Device   *io_Dev;
    struct            IOReq    *(*io_CallBack)(struct IOReq *iorP);
                                    /* call, do not ReplyMsg */
    IOInfo            io_Info;
    int32            io_Actual;    /* actual size of request completed
*/
    uint32            io_Flags;       /* internal to device driver */
    int32            io_Error;       /* any errors from request? */
    int32            io_Extension[2];    /* extra space if needed */
    Item            io_MsgItem;
    Item            io_SigItem;
} IOReq;
```

Fields that offer information to the average task are:

- `io_Info`.ioi_`User`. A copy of the `ioi_User` field from the IOInfo structure supplied by the task when calling `SendIO()` or `DoIO()`. This is a convenient location for the task to store context information associated with the IOReq.
- `io_Actual`. Supplies the size in bytes of the I/O operation just completed. This is a convenient place to find out the size of the last I/O transfer.
- `io_Error`. Contains any errors generated by the device carrying out the I/O operation. The meaning of this value depends on the device that was used. This value is also returned by `DoIO()` or `WaitIO()`.

Continuing I/O

Tasks often have a series of I/O operations to carry out with a device. If so, a task can recycle an IOReq once it's been returned; the task supplies new values in an IOInfo data structure, and then uses `SendIO()` or `DoIO()` to request a new operation.

A task isn't restricted to a single IOReq for a single device. If it's useful, a task can create two or more IOReqs for a device, and work with one IOReq while others are sent to the device or are awaiting action by the task.

Aborting I/O

If an asynchronous I/O operation must be aborted while in process at the device, the issuing task can use this call (defined in *io.h*):

```
Err AbortIO( Item ioreq )
```

`AbortIO()` accepts the item number of the IOReq that is responsible for the operation to be aborted. When executed, it notifies the device that the operation should be aborted. When it is safe, the operation is aborted and the device sets the IOReq's `Io_Error` field to `ABORTED`. The device then returns the IOReq to the task.

A task should always follow the `AbortIO()` call with a `WaitIO()` call so the task will wait for the abort to complete and the IOReq to return.

Finishing I/O

When a task completely finishes I/O with a device, the task should clean up by deleting any IOReqs it created and by closing the device. To delete an IOReq, a task uses this call (found in *io.h*):

```
Err DeleteIOReq( Item ioreq )
```

This call accepts the item number of an IOReq and frees any resources the IOReq used.

To close a device, a task uses the `CloseNameDevice()` call:

```
Err CloseNamedDevice( Item device )
```

Examples

The following two examples show how to use the timer device to read the current system time and make it wait a certain amount of time. See Portfolio Devices, for a complete description of the timer device.

Reading the System Time

Example 1 demonstrates how to use the timer device to read the current system time. The example does the following:

- Opens the timer device
- Creates an IOReq
- Initializes an IOInfo structure
- Calls `DoIO()` to perform the read operation
- Prints out the current time
- Cleans up

Portfolio provides convenience calls that make using the timer device easier, for example, `CreateTimerIOReq()` and `DeleteTimerIOReq()`. This example shows how to communicate with devices in the Portfolio environment. Note that the `SampleSystemTime()` and `SampleSystemTimeTV()` calls provide a more accurate reading of the system time than the calls used in this example.

Example 1: *Reading the current time (timerread.c).*

```
#include "types.h"
#include "string.h"
#include "io.h"
#include "device.h"
#include "item.h"
#include "time.h"
#include "stdio.h"
#include "operror.h"

int main(int32 argc, char **argv)
{
Item    deviceItem;
Item    ioreqItem;
IOReq  *ior;
IOInfo  ioInfo;
```

```
TimeVal tv;
Err     err;

    deviceItem = OpenNamedDevice("timer",0);
    if (deviceItem >= 0)
    {
        ioreqItem = CreateIOReq(0,0,deviceItem,0);
        if (ioreqItem >= 0)
        {
            ior = (IOReq *)LookupItem(ioreqItem);

            memset(&ioInfo,0,sizeof(ioInfo));
            ioInfo.ioi_Command         = CMD_READ;
            ioInfo.ioi_Unit            = TIMER_UNIT_USEC;
            ioInfo.ioi_Recv.iob_Buffer = &tv;
            ioInfo.ioi_Recv.iob_Len    = sizeof(tv);

            err = DoIO(ioreqItem,&ioInfo);
            if (err >= 0)
            {
                printf("Seconds %u, microseconds

                      %u\n",tv.tv_Seconds,tv.tv_Microseconds);
            }
            else
            {
                printf("DoIO() failed: ");
                PrintfSysErr(err);
            }
            DeleteIOReq(ioreqItem);
        }
        else
        {
            printf("CreateIOReq() failed: ");
            PrintfSysErr(ioreqItem);
        }
        CloseNamedDevice(deviceItem);
    }
    else
    {
        printf("OpenNamedDevice() failed: ");
        PrintfSysErr(deviceItem);
    }
```

```
    return 0;
}
```

Using the Timer Device to Wait

Example 2 demonstrates how to use the timer device to wait a certain amount of time. The program does the following:

- Parses the command-line arguments
- Opens the timer device
- Creates an IOReq
- Initializes an IOInfo structure
- Calls `DoIO()` to perform the wait operation
- Cleans up

Note that Portfolio provides convenience calls to make using the timer device easier, for example, `CreateTimerIOReq()`, `DeleteTimerIOReq()`, and `WaitTime()`. This example shows how to communicate with devices in the Portfolio environment.

Example 2: *Waiting for a specified time (timersleep.c).*

```
#include "types.h"
#include "string.h"
#include "io.h"
#include "device.h"
#include "item.h"
#include "time.h"
#include "stdio.h"
#include "operror.h"


int main(int32 argc, char **argv)
{
Item    deviceItem;
Item    ioreqItem;
IOReq  *ior;
IOInfo  ioInfo;
TimeVal tv;
Err     err;

    if (argc == 3)
    {
```

```
    tv.tv_Seconds      = strtoul(argv[1],0,0);
    tv.tv_Microseconds = strtoul(argv[2],0,0);

    deviceItem = OpenNamedDevice("timer",0);
    if (deviceItem >= 0)
    {
        ioreqItem = CreateIOReq(0,0,deviceItem,0);
        if (ioreqItem >= 0)
        {
            ior = (IOReq *)LookupItem(ioreqItem);

            memset(&ioInfo,0,sizeof(ioInfo));
            ioInfo.ioi_Command         = TIMERCMD_DELAY;
            ioInfo.ioi_Unit            = TIMER_UNIT_USEC;
            ioInfo.ioi_Send.iob_Buffer = &tv;
            ioInfo.ioi_Send.iob_Len    = sizeof(tv);

            err = DoIO(ioreqItem,&ioInfo);
            if (err >= 0)
            {
                printf("slept\n");
            }
            else
            {
                printf("DoIO() failed: ");
                PrintfSysErr(err);
            }
            DeleteIOReq(ioreqItem);
        }
        else
        {
            printf("CreateIOReq() failed: ");
            PrintfSysErr(ioreqItem);
        }
        CloseNamedDevice(deviceItem);
    }
    else
    {
        printf("OpenNamedDevice() failed: ");
        PrintfSysErr(deviceItem);
    }
}
else
{
```

```
        printf("Usage: timersleep <num seconds> <num microseconds>\n");
    }

    return 0;
}
```

Portfolio Devices

This chapter lists the standard Portfolio devices and their associated commands and options. It contains the following topics:

Introduction

Some of the devices currently in the 3DO system include:

- Timer device
- File device
- SPORT device
- CD-ROM device (not documented here)
- RAM device (not documented here)
- xbus device (not documented here)

All devices use I/O request items for communicating information between tasks and the device. Refer to The Portfolio I/O Model, for more information on devices and I/O requests.

The first step in the I/O process is for the task to open the device with the `OpenNamedDevice()` call.

A task must also get its own IOReq and IOInfo data structures to pass information to and from the device. The IOReq data structure is created using the call `CreateIOReq()`. The IOInfo data structure is allocated by the task either via a memory allocation call or on the stack.

Note: The IOInfo structure must be cleared of all unused bits before it is used.

After setting the IOInfo fields, a task uses the `SendIO()` or `DoIO()` functions to pass the item number of the I/O request structure and a pointer to the IOInfo structure to the device. The I/O request is passed to the device, and the appropriate action is taken by the device as specified by the IOInfo structure fields.

Communicating With Devices

A task sends commands using the IOInfo data structure to a device. The `IOInfo.ioi_Command` field is set to the command a task sends to a device. In addition to the command, some devices allow the task to specify options to the command. The options are set up by defining the `IOInfo.ioi_CmdOptions` field. The `IOInfo.ioi_Send` and `IOInfo.ioi_Recv` substructures must also be set for the information to be sent to the device and to receive it back from the device. The reference sections that follow provide detailed information about the commands and command options, and send and receive buffers that a task uses to communicate with a device.

The Timer Device

The timer device is a software component that provides a standardized interface to the timing hardware.The 3DO hardware architecture includes a number of clocks and timers.

The timer device has two separate timer units that offer different timing characteristics: the microsecond unit (`TIMER_UNIT_USEC`) and the vertical blank unit (`TIMER_UNIT_VBLANK`). The microsecond timer deals with time quantities using seconds and microseconds, while the vertical blank timer counts time in vertical blank intervals. Vertical blank intervals are discussed in more detail in The Vertical Blank Unit. Both units respond to the same commands.

Using either of the timer units, you can do four basic operations:

- Sample the current time
- Ask the timer to notify you when a given time arrives
- Ask the timer to notify you after a given amount of time passes
- Ask the timer to notify you at regular intervals until you ask it to stop

The following sections describe how to perform these operations with either timer device units.

Note: The timer device currently doesn't provide real-time clocks. That is, the timer starts counting time only when the machine is turned on, it stops counting time when the machine is shut down. Therefore, it is not currently possible to automatically determine the current time of day. This capability may be added to the 3DO architecture in the future.

Working With the Timer Device

Communicating with the two units of the timer device is done using the standard Portfolio I/O commands. To send commands to the timer device, you must complete the following steps:

1. Open the timer device using the `OpenNamedDevice()` function.
2. Create an IOReq structure by calling the `CreateIOReq()` function.
3. Initialize an IOInfo structure that specifies the command and parameters to the command.
4. Send the command to the timer device using either `DoIO()` or `SendIO()`.

The Microsecond Unit

`The microsecond timer unit provides very high-resolution timing.`

Although it has very high short-term accuracy, its time base can drift slightly over extended periods of time.

This microsecond unit of the timer device deals in time quantities using the TimeVal structure, which is defined as:

```
typedef struct timeval
{
    int32 tv_Seconds;         /* seconds */
    int32 tv_Microseconds;     /* and microseconds */
} TimeVal;
```

Reading the Current System Time

To read the current system time, you must initialize an IOInfo structure such as:

```
IOInfo  ioInfo;
TimeVal tv;

memset(&ioInfo,0,sizeof(ioInfo));
ioInfo.ioi_Command                          = CMD_READ
ioInfo.ioi_Unit                           = TIMER_UNIT_USEC;
ioInfo.ioi_Recv.iob_Buffer = &tv;
ioInfo.ioi_Recv.iob_Len                           = sizeof(tv);
```

The `ioi_Command` field is set to `CMD_READ`, indicating the current system time should be read. `ioi_Unit` indicates which unit of the timer device to query. `ioi_Recv.iob_Buffer` points to a TimeVal structure. This pointer is where the timer device stores the current time value. Finally, `ioi_Recv.iob_Len` holds the size of the TimeVal structure.

Once an I/O operation is performed on the timer device using the above `IOInfo`, the timer device puts the current system time in the supplied TimeVal structure, and completes the I/O operation by sending your task a message or a signal, depending on how the I/O request was created.

For a high performance way to read the system's microseconds clock, see High-Performance Timing.

Waiting for a Specific Time to Arrive

A common use for the timer device is to provide automatic notification of the passage of time. For example, if you want a given picture to remain on the display for exactly 1 second, you can send a command to the timer device telling it to send you a signal when 1 second has passed. While you are

waiting for that second to pass, your task can do other work, such as play music, confident in the fact that the timer device will notify it when the appropriate amount of time has passed.

You can ask the timer device to notify you when a specific time arrives. To do this, you must first ask the system what the current time is by sending the device a CMD_READ. Once you know the current time, you can use the AddTimes() and SubTimes() calls, explained below, to calculate the time to receive a notification. Once you have calculated the time to be notified, you can send the TIMERCMD_DELAYUNTIL command to the timer device. You must initialize the IOInfo structure in the following way:

```
IOInfo  ioInfo;
TimeVal tv;

memset(&ioInfo,0,sizeof(ioInfo));
ioInfo.ioi_Command                  = TIMERCMD_DELAYUNTIL;
ioInfo.ioi_Unit                   = TIMER_UNIT_USEC;
ioInfo.ioi_Send.iob_Buffer = &tv;
ioInfo.ioi_Send.iob_Len   = sizeof(tv);
```

The ioi_Command field is set to TIMERCMD_DELAYUNTIL, indicating that the timer will wait until a specific time arrives. ioi_Unit indicates which unit of the timer device to use. ioi_Send.iob_Buffer points to a TimeVal structure. This contains the amount of time to wait. Finally, ioi_Send.iob_Len holds the size of the TimeVal structure.

You can send the I/O request to the timer device using either DoIO() or SendIO(). When using DoIO(), your task is put to sleep until the requested time. If you use SendIO(), then your task is free to continue working while the timer is counting time. When the requested time arrives, the timer device will either send your task the SIGF_IODONE signal, or will send you a message as specified in your I/O request.

Waiting a Specific Amount of Time

Instead of asking the timer to wait until a given time, you can tell it to wait for a fixed amount of time to pass. To achieve this, you follow the procedure in the previous section except that you initialize the IOInfo structure differently. For a specific amount of time, ioi_Command must be set to TIMERCMD_DELAY, and the TimeVal structure you supply must specify an amount of time instead of a certain time.

Getting Repeated Notifications

Often, it is desirable to have a timer automatically generate a signal in regular fixed intervals. The TIMERCMD_METRONOME commands arranges to have a signal sent to your task for an undetermined

length of time at a fixed rate.

```
IOInfo  ioInfo;
TimeVal tv;
int32   signals;

memset(&ioInfo,0,sizeof(ioInfo));
ioInfo.ioi_Command                          = TIMERCMD_METRONOME;
ioInfo.ioi_Unit                           = TIMER_UNIT_USEC;
ioInfo.ioi_CmdOptions     = signals;
ioInfo.ioi_Send.iob_Buffer = &tv;
ioInfo.ioi_Send.iob_Len   = sizeof(tv);
```

The `ioi_Command` field is set to `TIMERCMD_METRONOME` indicating that the timer acts as a metronome, and sends your task signals every time a specified amount of time passes. `ioi_Unit` indicates which unit of the timer device to use. `ioi_CmdOptions` specifies the signal mask that the timer device should use when signalling your task. `ioi_Send.iob_Buffer` points to a TimeVal structure. This contains the amount of time between each signal that the timer device sends your task. Finally, `ioi_Send.iob_Len` holds the size of the TimeVal structure.

Send the I/O request to the timer device using `SendIO()`. Once this is done, the timer device will send a signal to your task every time the specified amount of time passes. To stop the timer device from sending these signals, you must abort the I/O request using the `AbortIO()` call.

The Vertical Blank Unit

The vertical blank timer unit provides a fairly coarse measure of time, but is very stable over long periods of time. It offers a resolution of either 1/60th of a second on NTSC systems or 1/50th of a second on PAL systems. Vertical blanking is a characteristic of raster scan displays, and occurs on a fixed time-scale synchronized with the display hardware.

A *vblank* is the amount of time it takes for the video beam to perform an entire sweep of the display. Given that displays operate at different refresh rates in NTSC (60 Hz) compared to PAL (50 Hz), the amount of time taken by a vblank varies. Since the vblank unit of the timer device deals with time exclusively in terms of vblank units, waiting for a fixed number of vblanks will take different amounts of time on NTSC and PAL.

The advantages of the vertical blank timer are that it remains stable for very long periods of time; it involves slightly less overhead than the microsecond unit; and it is synchronized with the video beam. Being synchronized with the video beam is very important when creating animation sequences.

This vertical blank unit of the timer device deals in time quantities using the VBlankTimeVal structure, which is defined as:

```
typedef struct VBlankTimeVal
{
    uint32 vbltv_VBlankHi32;  * upper 32 bits of vblank counter */
    uint32 vbltv_VBlankLo32;    /* lower 32 bits of vblank counter */
} VBlankTimeVal;
```

Vblanks are counted using a 64-bit counter. This is represented in two 32-bit words. The upper-32 bits, which are the most significant, are stored in the `vbltv_VBlankHi32` field, while the lower-32 bits are stored in the `vbltv_VBlankLo32` field.

Reading the Current System Time

To read the current system time, you must initialize an IOInfo structure such as:

```
IOInfo       ioInfo;
VBlankTimeVal vbltv;

memset(&ioInfo,0,sizeof(ioInfo));
ioInfo.ioi_Command                          = CMD_READ
ioInfo.ioi_Unit                           = TIMER_UNIT_VBLANK;
ioInfo.ioi_Recv.iob_Buffer                            = &vbltv;
ioInfo.ioi_Recv.iob_Len                             = sizeof(vbltv);
```

`ioi_Command` is set to `CMD_READ`, indicating that the current system time should be read. `ioi_Unit` indicates which unit of the timer device to query. `ioi_Recv.iob_Buffer` points to a VBlankTimeVal structure. This is where the timer device stores the current vblank count. Finally, `ioi_Recv.iob_Len` holds the size of the VBlankTimeVal structure.

Once the I/O is complete, the supplied VBlankTimeVal structure is filled with the current vblank count. Given that there are either 50 or 60 vblanks per second, over 800 days worth of vblanks can be stored in `vbltv_VBlankLo32`. Whenever `vbltv_VBlankLo32` exceeds the maximum value it can contain (2^32 - 1), then the value of `vbltv_VBlankHi32` is incremented by 1. This means that the VBlankTimeVal structure being used to store vblank counts can hold up to (2^32 * 800) days, which is probably longer than the time remaining before the sun goes supernova.

Waiting for a Specific Time

Similar to the microsecond unit, the vblank unit can wait for a specific time. You specify this time in

terms of vblanks. Unlike the microsecond unit, you do not specify the amount of time to wait using a TimeVal structure. Instead, set the `ioi_Offset` field of the IOInfo structure to the vblank count.

```
IOInfo ioInfo;

memset(&ioInfo,0,sizeof(ioInfo));
ioInfo.ioi_Command                = TIMERCMD_DELAYUNTIL;
ioInfo.ioi_Unit                 = TIMER_UNIT_VBLANK;
ioInfo.ioi_Offset                 = vblankCountToWaitUntil;
```

Waiting for a Specific Amount of Time

The vblank unit can wait for a given number of vblanks, that is, a specific amount of time. This is done by initializing an IOInfo structure such as:

```
IOInfo ioInfo;

memset(&ioInfo,0,sizeof(ioInfo));
ioInfo.ioi_Command                  = TIMERCMD_DELAY;
ioInfo.ioi_Unit                 = TIMER_UNIT_VBLANK;
ioInfo.ioi_Offset                 = numberOfVBlanksToWait;
```

It is important to understand that the timer counts vblanks in a very strict way. Whenever the video beam reaches a known location on the display, the vblank counter is incremented. So if you ask the timer to wait for 1 vblank while the beam is near the trigger location, the I/O request will be returned in less than 1/60th or 1/50th of a second.

The `WaitVBL()` function is a wrapper function that initializes an `IOInfo` structure using the `TIMERCMD_DELAY` command, and sends it to the timer device.

Getting Repeated Notifications

Similar to the microsecond unit, the vblank unit can automatically send you signals at specified intervals. You specify the interval time in terms of vblanks. Unlike with the microsecond unit, you do not specify the amount of time between signals using a TimeVal structure. Instead, set the `ioi_Offset` field of the IOInfo structure to the vblank count between signals.

```
IOInfo  ioInfo;
int32   signals;

memset(&ioInfo,0,sizeof(ioInfo));
ioInfo.ioi_Command                          = TIMERCMD_METRONOME;
```

```
ioInfo.ioi_Unit                         = TIMER_UNIT_VBLANK;
ioInfo.ioi_CmdOptions     = signals;
ioInfo.ioi_Offset        = vblanksBetweenSignals;
```

You should send the I/O request to the timer device using `SendIO()`. Once this is done, the timer device will send a signal to your task every time the specified number of vblanks occurs. To stop the timer device from sending you these signals, you must abort the I/O request using the `AbortIO()` call.

High-Performance Timing

It is sometimes necessary to measure very short time intervals with very high accuracy. This is especially useful when trying to measure the performance of various pieces of code. Although using the timer device and the `CMD_READ` command gives fairly accurate readings, the overhead involved in doing device I/O is often enough to skew the results of small time intervals.

Portfolio provides the `SampleSystemTime()` and `SampleSystemTimeTV()` functions that allow low-overhead sampling of the system clock. These functions are based on the same hardware clock as the microsecond unit of the timer device, and so refer to the same time base.

`SampleSystemTime()` returns the current seconds counter of the system clock. The function also returns the current microseconds counter in the R1 CPU register of the ARM processor. This value is only available if you program in assembly language. To get both values in C, you must use the `SampleSystemTimeTV()` function, which puts the current values of the seconds and microseconds counters in the TimeVal structure that you supply.

Time Arithmetic

The following calls calculate or compare time values.

- AddTimes(). Adds two time values, yielding the total time for both. This call is useful for time calculation.
- CompareTimes(). Compares two time values. This call helps you determine what happens when.
- SubTimes(). Subtracts one time value from another, yielding the time difference between the two.
- TimeLaterThan(). Returns whether a time value comes before another time value.
- TimeLaterThanOrEqual(). Returns whether a time value comes before or at the same time as another time value.

Simplified Timer Device Interface

The following calls create I/O requests and wait fixed amounts of time. These are simple convenience calls that interface to the timer device for you. All of these routines use the microsecond timer unit.

- CreateTimerIOReq(). Creates an I/O request for communication with the timer device.
- DeleteTimerIOReq(). Frees any resources used in a previous call to `CreateTimerIOReq()`.
-
- WaitUntil(). Puts the current context to sleep until the system clock reaches a given time.

The SPORT Device

The 3DO hardware includes a minimum of 1 MB of VRAM. VRAM is a special type of memory that offers a very wide interface bus, which allows for efficient implementation of display buffers.

SPORT stands for Serial PORT, and refers specifically to the VRAM's serial port, not to an external RS-232 serial port. VRAM has special hardware that lets you copy pages of memory, clear pages of memory, and replicate pages of memory at an amazing rate. The SPORT device provides access to these capabilities. It is ideal for clearing a display to a specific color, or setting a display to a static background picture.

Working With the SPORT Device

The SPORT device always operates on pages of VRAM. The size of VRAM pages is returned by the `GetPageSize(MEMTYPE_VRAM)` function call. All operations performed by the SPORT device must start on an even page boundary and be a multiple of the page size length. The current VRAM page size is 2 KB, but you must not rely on this fact. Always use the `GetPageSize()` function to get the actual page size.

Communicating with the SPORT device is done with the standard Portfolio I/O commands. The SPORT device only has a single unit, unit 0. To send commands to the SPORT device, you must complete the following steps:

1. Open the SPORT device using the `OpenNamedDevice()` function.
2. Create an IOReq structure by calling the `CreateIOReq()` function.
3. Initialize an IOInfo structure that specifies the command and parameters of the command.
4. Send the command to the SPORT device using either `DoIO()` or `SendIO()`.

Copying VRAM Pages

The `SPORTCMD_COPY` command copies pages of VRAM to other pages of VRAM. The copy operation always occurs during vertical blanking because the serial port on the VRAM is always in use when video is displayed. When vertical blanking occurs, the serial port becomes available for other duties such as copying and cloning pages.

To use `SPORTCMD_COPY`, initialize an IOInfo structure such as:

```
IOInfo ioInfo;
```

```
memset(&ioInfo,0,sizeof(ioInfo));
ioInfo.ioi_Command                          = SPORTCMD_COPY;
ioInfo.ioi_Offset                             = mask;
ioInfo.ioi_Send.iob_Buffer                      = sourceAddress;
ioInfo.ioi_Send.iob_Len                       = numBytes;
ioInfo.ioi_Recv.iob_Buffer                      = destinationAddress;
ioInfo.ioi_Recv.iob_Len                       = numBytes;
```

The `sourceAddress` field points to the source data to be copied, while the `destinationAddress` field points to the address where the data is copied. Both of these addresses must be in VRAM, and both must fall on even VRAM page boundaries. `numBytes` indicates the number of bytes to copy. This value must be an even multiple of the VRAM page size. Finally, `mask` is a 32-bit value that determines which bits of each word of data are copied. Every on bit indicates a bit that will be copied to the destination. Every off bit indicates that the corresponding bit in the destination will remain unchanged.

Don't use a mask value other than 0xffffffff because future hardware implementations may impose serious performance penalties for using masks with other values.

Replicating VRAM Pages

The `SPORTCMD_CLONE` command replicates a page of VRAM to a series of other pages. This is useful when creating wallpaper backgrounds. The cloning operation always occurs during vertical blanking because the serial port on the VRAM is always in use when video is being displayed. When vertical blanking occurs, the serial port becomes available for other duties such as copying and cloning pages.

To use `SPORTCMD_CLONE`, you must initialize an IOInfo structure such as:

```
IOInfo ioInfo;

memset(&ioInfo,0,sizeof(ioInfo));
ioInfo.ioi_Command                          = SPORTCMD_CLONE;
ioInfo.ioi_Offset                             = mask;
ioInfo.ioi_Send.iob_Buffer                      = sourcePageAddress;
ioInfo.ioi_Send.iob_Len                       = pageSize;
ioInfo.ioi_Recv.iob_Buffer                      = destinationAddress;

ioInfo.ioi_Recv.iob_Len                       = numBytes;
```

The `sourcePageAddress` points to the source page to be replicated, while the `destinationAddress` field points to the address where the data is copied. Both of these addresses must be in VRAM, and both must fall on even VRAM page boundaries. `pageSize` is the size of a single VRAM page, as returned by `GetPageSize(MEMTYPE_VRAM)`. `numBytes` indicates the number of

bytes to replicate. This value is typically calculated as (`numPages * pageSize`). This value must be an even multiple of the VRAM page size. Finally, `mask` is a 32-bit value that determines which bits of each word of data are put on the destination pages. Every on bit indicates a bit that will be copied to the destination. Every off bit indicates that the corresponding bit in the destination will remain unchanged.

Don't use a mask value other than 0xffffffff because future hardware implementations may impose serious performance penalties for using masks with other values.

Setting VRAM Pages to a Fixed Value

The `FLASHWRITE_CMD` command sets the value of a range of VRAM pages. Unlike the copy and clone operations described above, this command does not operate in the vertical blank area and occurs immediately.

To use `FLASHWRITE_CMD`, you must initialize an IOInfo structure such as:

```
IOInfo ioInfo;

memset(&ioInfo,0,sizeof(ioInfo));
ioInfo.ioi_Command                          = FLASHCMD_WRITE;
ioInfo.ioi_CmdOptions    = mask;
ioInfo.ioi_Offset       = value;
ioInfo.ioi_Send.iob_Buffer                          = pageAddress;
ioInfo.ioi_Send.iob_Len                           = numBytes;
ioInfo.ioi_Recv.iob_Buffer                          = pageAddress;
ioInfo.ioi_Recv.iob_Len                           = numBytes;
```

The `pageAddress` field points to the pages of VRAM to be set to a fixed value. This address must be in VRAM, and must fall on an even VRAM page boundary. The `numBytes` field indicates the number of bytes to affect. This value must be an even multiple of the VRAM page size. `value` is the value to which the words of data within the pages should be set. Finally, `mask` is a 32-bit value that determines which bits of each word of data are affected. Every on bit indicates a bit that will be affected. Every off bit indicates that the corresponding bit will remain unchanged.

Don't use a mask value other than 0xffffffff because future hardware implementations may impose serious performance penalties for using masks with other values.

The File Pseudo-Device

The File folio, in cooperation with the various file systems, provides the services needed to access filesystem-oriented external storage; for example, reading files from a CD-ROM, or writing files to the 3DO NVRAM (non-volatile RAM).

When you open a file using the File folio `OpenDiskFile()` function, the item that is returned serves as a handle to the file. The `Item` is of type Device. This enables communication to the filesystem that controls this file using the standard Portfolio I/O model.

When you send a command to a file device, the file system responsible for the file wakes up and executes the command. Therefore, the file device is only a pseudo-device, serving as a gateway to the underlying filesystem.

The File folio provides many high-level functions to control file systems. For example, you can create files using the `CreateFile()` function, or you can delete files using the `DeleteFile()` function. These File folio functions are merely wrappers for file system commands. The functions internally create file devices and send commands to the device to perform work.

This section explains the various commands you can send to a file device. It is often much easier to use the higher-level File folio functions to interact with the file system. However, there are some operations that can only be performed by interfacing to the file device directly.

Getting Filesystem Status

Once you open a file, you can obtain information about the filesystem the file resides on. This is done by using the `FILECMD_FSSTAT` command.

```
IOInfo        ioInfo;
FileSystemStat fsStat;

memset(&ioInfo,0,sizeof(ioInfo));
ioInfo.ioi_Command                      = FILECMD_FSSTAT;
ioInfo.ioi_Recv.iob_Buffer                      = &fsStat;
ioInfo.ioi_Recv.iob_Len                    = sizeof(FileSystemStat);
```

Once the command returns successfully, you can look at the fields in the fsStat structure for the status information. The `fst_BitMap` field of the FileSystemStat structure indicates which fields in the rest of the structure are valid and can be examined. Different file systems cannot always provide all the information in a FileSystemStat structure. For example, if the `FSSTAT_SIZE` bit is set in `fst_BitMap`, it means the `fst_Size` field of the FileSystemStat structure is valid.

The fields in the FileSystemStat structure are:

- **fst_BitMap.** Indicates which fields of the structure contain valid data.
- **fst_CreateTime.** Indicates when the filesystem was created. This field is currently never valid and always contains 0.
- **fst_BlockSize.** Indicates the nominal size of data blocks in the filesystem. To determine the block size to use when reading and writing files, query the file's status and extract the block size from that information.
- **fst_Size.** Indicates the total size of the filesystem in blocks.
- **fst_MaxFileSize.** Indicates the maximum size of a file in the filesystem
- **fst_Free.** Indicates the total number of blocks currently not in use on the filesystem
- **fst_Used.** Indicates the total number of blocks currently in use on the filesystem.

Getting File Status

Once you open a file, you can obtain information about the file by using the `CMD_STATUS` command.

```
IOInfo    ioInfo;
FileStatus fStat;

memset(&ioInfo,0,sizeof(ioInfo));
ioInfo.ioi_Command                      = CMD_STATUS;
ioInfo.ioi_Recv.iob_Buffer              = &fStat;
ioInfo.ioi_Recv.iob_Len                 = sizeof(FileStatus);
```

Once the command completes successfully, you can look at the fields in the fStat structure for the status information. Fields of interest are:

- **fs.ds_DeviceBlockSize.** The block size to use when reading or writing this file.
- **fs.ds_DeviceBlockCount.** The number of blocks of data in the file.
- **fs_ByteCount.** The number of bytes currently in the file.

Creating and Deleting Files

To create a file, use the File folio's `CreateFile()` function. You supply the path name of the file to create, and the function does the necessary work to create a new file entry on the filesystem.

To delete an existing file, use the File folio's `DeleteFile()` function. You supply it the path name of the file

to delete, and the function does the necessary work to remove the file's entry from the filesystem.

Allocating Blocks for a File

Before you can write data to a file, you must allocate enough free blocks in the file to hold the data to be written. This is done by using the FILECMD_ALLOCBLOCKS command.

```
IOInfo ioInfo;

memset(&ioInfo,0,sizeof(ioInfo));
ioInfo.ioi_Command                = FILECMD_ALLOCBLOCKS;
ioInfo.ioi_Offset                = numBlocks;
```

The numBlocks variable contains the number of blocks by which to extend the file. If this value is negative, the size of the file is reduced by the specified number of blocks.

Writing Data to a File

You must use the CMD_WRITE command to write data to a file. All write operations must be performed in full blocks. The size of the blocks can be obtained by sending a CMD_STATUS command to the file device. The write operation must also be aligned on a block boundary within the file.

```
IOInfo ioInfo;

memset(&ioInfo,0,sizeof(ioInfo));
ioInfo.ioi_Command                                = CMD_WRITE;
ioInfo.ioi_Offset                                 = blockOffset;
ioInfo.ioi_Send.iob_Buffer                        = dataAddress;
ioInfo.ioi_Send.iob_Len                           = numBytes;
```

The blockOffset field indicates the offset in blocks from the beginning of the file where the data is to be written. The dataAddress value points to the data that is to be written. Finally, the numBytes value indicates the number of bytes to write out. This value must be an even multiple of the block size for the file.

Marking the End of a File

Once you are done writing data to a file, you must mark the end of the file using the FILECMD_SETEOF command. This command tells the filesystem how many useful bytes of data are in the file. Because you can only transfer data in terms of blocks, sending this command tells the filesystem how many bytes of the last written block are useful bytes.

```
IOInfo ioInfo;
```

```
memset(&ioInfo,0,sizeof(ioInfo));
ioInfo.ioi_Command                              = FILECMD_SETEOF;
ioInfo.ioi_Offset                               = numBytesInFile;
```

Reading Data From a File

Reading information from a file is done much the same way you write information to a file. You must supply the offset in blocks where to start reading data, and the number of bytes of data to read.

```
IOInfo ioInfo;

memset(&ioInfo,0,sizeof(ioInfo));
ioInfo.ioi_Command                                  = CMD_READ;
ioInfo.ioi_Offset                                   = blockOffset;
ioInfo.ioi_Recv.iob_Buffer                          = dataAddress;
ioInfo.ioi_Recv.iob_Len                             = numBytes;
```

The `blockOffset` value indicates the offset in blocks from the beginning of the file from which data is read. `dataAddress` indicates an address in memory where the data will be copied once read from the filesystem. Finally, `numBytes` contains the number of bytes to read. This number must be an even multiple of the block size of the file.

Getting Directory Information

The `OpenDiskFile()` function opens a directories. You can then use the `FILECMD_READDIR` command to scan the directory and obtain information about files in the directory.

```
IOInfo        ioInfo;
DirectoryEntry dirEntry;

memset(&ioInfo,0,sizeof(ioInfo));
ioInfo.ioi_Command                                  = FILECMD_READDIR;
ioInfo.ioi_Offset                                   = fileNum;
ioInfo.ioi_Recv.iob_Buffer                          = &dirEntry;
ioInfo.ioi_Recv.iob_Len                             = sizeof(DirectoryEntry);
```

The `fileNum` value indicates the number of the file within the directory to read. You start with file 0, and keep going until the I/O cannot be completed because there are no more files. The dirEntry structure will be filled out with information about the specified file.

Getting the Path of a File

Once you have an open file, you may need to determine the exact path to reach this file. The

`FILECMD_GETPATH` command determines the exact path to an open file.

```
IOInfo ioInfo;

char   path[FILESYSTEM_MAX_PATH_LEN];

memset(&ioInfo,0,sizeof(ioInfo));
ioInfo.ioi_Command                      = FILECMD_GETPATH;
ioInfo.ioi_Recv.iob_Buffer              = path;
ioInfo.ioi_Recv.iob_Len                 = sizeof(path);
```

Once the command completes successfully, the path variable contains a complete unambiguous path to reach the file. You can then use this path to issue another `OpenDiskFile()` command or a `DeleteFile()` command.

Function Calls

I/O operations use the following calls. See Kernel Folio Calls, in the *3DO System Programmer's Reference* for complete details on these calls.

Opening and Closing Devices

The following calls open and close devices:

- `CloseNamedDevice()` Closes a device previously opened with `OpenNamedDevice()`.
- `FindDevice()` Returns the item number of a device.
- `FindAndOpenDevice()` Finds and opens a device item.
- `OpenNamedDevice()` Opens a named device.

Creating, Deleting, and Creating Pointers for IOReqs

The following calls handle IO requests:

- `CreateIOReq()` Creates an I/O request.
- `DeleteIOReq()` Deletes an I/O request.
- `LookupItem()` Gets a pointer to an item.

Controlling the I/O Process

The following calls handle I/O processes:

- `AbortIO()` Aborts an I/O operation.
- `CheckIO()` Checks if an asynchronous I/O request has completed.
- `DoIO()` Performs synchronous I/O.
- `SendIO()` Requests asynchronous I/O.
- `WaitIO()` Waits for an asynchronous I/O request to complete.

Kernel Folio Calls

This chapter describes the kernel folio procedure calls in alphabetical order. The list provides a quick summary of each procedure and the page number of the first page of its description.

- CreateUniqueSemaphore Creates a semaphore with a unique name.
- DeleteIOReq Deletes an I/O request.
- DeleteItem Deletes an item.
- DeleteMemDebug Releases memory debugging resources.
- DeleteMsg Deletes a message.
- DeleteMsgPort Deletes a message port.
- DeleteSemaphore Deletes a semaphore.
- DeleteThread Deletes a thread.
- DoIO Performs synchronous I/O.
- DumpMemDebug Dumps memory allocation debugging information.
- DumpNode Prints contents of a node.
- DumpTagList Prints contents of a tag list.
- exit Exits from a task or thread.
- ffs Finds the first set bit.
- FindAndOpenDevice Finds a device by name and opens it.
- FindAndOpenFolio Finds a folio by name and opens it.
- FindAndOpenItem Finds an item by type and tags and opens it.
- FindAndOpenNamedItem Finds an item by name and opens it.
- FindDevice Finds a device by name.
- FindFolio Finds a folio by name.
- FindItem Finds an item by type and tags.
- FindLSB Finds the least-significant bit.
- FindMSB Finds the highest-numbered bit.
- FindMsgPort Finds a message port by name.
- FindNamedItem Finds an item by name.
- FindNamedNode Finds a node by name.
- FindNodeFromHead Returns a pointer to a node appearing at a given ordinal position from the head of the list.
- FindNodeFromTail Returns a pointer to a node appearing at a given ordinal position from the tail of the list.
- FindSemaphore Finds a semaphore by name.
- FindTagArg Looks through a tag list for a specific tag.
- FindTask Finds a task by name.
- FindVersionedItem Finds an item by name and version.
- FirstNode Gets the first node in a list.
- free Frees memory from malloc().
- FreeMem Frees memory from AllocMem().
- FreeMemList Frees a memory list.
- FreeMemToMemList Frees a private block of memory.

- FreeMemToMemLists Returns memory to the free pool.
- FreeSignal Frees signals.
- GetBankBits Finds out which VRAM bank contains a memory location.
- GetCurrentSignals Gets the currently received signal bits.
- GetFolioFunc Returns a pointer to a folio function.
- GetMemAllocAlignment Gets the memory-allocation alignment size
- GetMemTrackSize Get the size of a block of memory allocated with MEMTYPE_TRACKSIZE.
- GetMemType Gets the type of the specified memory.
- GetMsg Gets a message from a message port.
- GetNodeCount Counts the number of nodes in a list.
- GetNodePosFromHead Gets the ordinal position of a node within a list, counting from the head of the list.
- GetNodePosFromTail Gets the ordinal position of a node within a list, counting from the tail of the list.
- GetPageSize Gets the number of bytes in a memory page.
- GetSysErr Gets the error string for an error.
- GetTagArg Finds a TagArg in list and returns its ta_Arg field.
- GetTaskSignals Gets the currently received signal bits.
- GetThisMsg Gets a specific message.
- InitList Initializes a list.
- InsertNodeAfter Inserts a node into a list after another node already in the list.
- InsertNodeBefore Inserts a node into a list before another node already in the list.
- InsertNodeFromHead Inserts a node into a list.
- InsertNodeFromTail Inserts a node into a list.
- IsEmptyList Checks whether a list is empty.
- IsItemOpened Determines whether a task or thread has opened a given item.
- IsListEmpty Checks whether a list is empty.
- IsMemReadable Determines whether a region of memory is fully readable by the current task.
- IsMemWritable Determines whether a region of memory is fully writable by the current task.
- IsNode Checks that a node pointer is not the head list anchor.
- IsNodeB Checks that a node pointer is not the tail list anchor.
- LastNode Gets the last node in a list.
- LockItem Locks an item.
- LockSemaphore Locks a semaphore.
- LookupItem Gets a pointer to an item.
- malloc Allocates memory.
- MkNodeID Creates an item type value
- NextNode Gets the next node in a list.

- NextTagArg Finds the next TagArg in a tag list.
- OpenItem Opens an item.
- OpenNamedDevice Opens a named device.
- PrevNode Gets the previous node in a list.
- PrintError Prints the error string for an error
- PrintfSysErr Prints the error string for an error.
- ReadHardwareRandomNumber Gets a 32-bit random number.
- RemHead Removes the first node from a list.
- RemNode Removes a node from a list.
- RemTail Removes the last node from a list.
- ReplyMsg Sends a reply to a message.
- ReplySmallMsg Sends a reply to a small message.
- SampleSystemTime Samples the system time with very low overhead.
- SampleSystemTimeTV Samples the system time with very low overhead.
- SanityCheckMemDebug Checks all current memory allocations to make sure all of the allocation cookies are intact
- ScanList Walks through all the nodes in a list.
- ScanListB Walks through all the nodes in a list backwards.
- ScavengeMem Returns task's free memory pages to the system memory pool.
- SendIO Requests asynchronous I/O.
- SendMsg Sends a message.
- SendSignal Sends a signal to another task.
- SendSmallMsg Sends a small message.
- SetItemOwner Changes the owner of an item.
- SetItemPri Changes the priority of an item.
- SetNodePri Changes the priority of a list node.
- UniversalInsertNode Inserts a node into a list.
- UnlockItem Unlocks a locked item.
- UnlockSemaphore Unlocks a semaphore.
- WaitIO Waits for an I/O request to complete.
- WaitPort Waits for a message to arrive.
- WaitSignal Waits until a signal is received.
- Yield Give up the CPU to a task of equal priority.

AbortIO

Aborts an I/O operation.

Synopsis

```
Err AbortIO( Item ior )
```

Description

This procedure aborts an I/O operation. If the I/O operation has already completed, calling `AbortIO()` has no effect. If it is not complete, it will be aborted.

A task should call `WaitIO()` immediately after calling `AbortIO()`. When `WaitIO()` returns, the task knows that the I/O operation is no longer active. It can then confirm that the I/O operation was aborted before it was finished by checking the io_Error field of the IOReq structure. If the operation aborted, the value of this field is ER_Aborted.

Arguments

ior
: The item number of the I/O request to abort.

Return Value

The procedure returns 0 if the I/O operation was successfully aborted or an error code (a negative value) if an error occurs. Possible error codes include:

BADITEM
: The ior argument is not an IOReq.

NOTOWNER
: The ior argument is an IOReq but the calling task is not its owner.

Implementation

SWI implemented in kernel folio V20.

Associated Files

io.h
: ANSI C Prototype

See Also

CheckIO(), CreateIOReq(), DeleteIOReq(), DoIO(), SendIO(), WaitIO()

CheckIO

Checks for I/O request completion.

Synopsis

```
int32 CheckIO( Item ior )
```

Description

This procedure checks to see if an I/O request has completed.

Arguments

ior
: The item number of the I/O request to be checked.

Return Value

The procedure returns 0 if the I/O is not complete. It returns > 0 if it is complete. It returns BADITEM if ior is bad.

Implementation

Convenience call implemented in clib.lib V20. Became a folio call in kernel folio V24.

Associated Files

io.h
: ANSI C Prototype

See Also

AbortIO(), CreateIOReq(), DeleteIOReq(), DoIO(), SendIO(), WaitIO()

CreateIOReq

Creates an I/O request.

Synopsis

```
Item CreateIOReq( const char *name, uint8 pri, Item dev, Item mp )
```

Description

This convenience procedure creates an I/O request item.

When you create an I/O request, you must decide how the device will notify you when an I/O operation completes. There are two choices:

- Notification by signal
- Notification by message

With notification by signal, the device will send your task the SIGF_IODONE signal whenever an I/O operation completes. This is a low-overhead mechanism, which is also low on information. When you get the signal, all you know is that an I/O operation has completed. You do not know which operation has completed. This has to be determined by looking at the state of all outstanding I/O requests.

Notification by message involves slightly more overhead, but provides much more information. When you create the I/O request, you indicate a message port. Whenever an I/O operation completes, the device will send a message to that message port. The message will contain the following information:

- msg_Result Contains the io_Error value from the I/O request. This indicates the state of the I/O operation, whether it worked or failed.
- msg_DataPtr Contains the item number of the I/O request that completed.
- msg_DataSize Contains the value of the ioi_User field taken from the IOInfo structure used when initiating the I/O operation.

Arguments

name
: The name of the I/O request (see "Notes").

pri

The priority of the I/O request. For some device drivers, this value determines the order in which I/O requests are processed. When in doubt about what value to use, use 0.

dev
: The item number of the device to which to send the I/O request.

mp
: If a task wants to receive a message when an I/O request is finished, this argument must be the item number of the message port to receive the message. If the task wants to receive a signal when an I/O request is finished instead of a message, this argument must be 0.

Return Value

The procedure returns the item number of the new I/O request or one of the following error codes if an error occurs:

BADITEM
: The mp argument was not zero but did not specify a valid message port.

ER_Kr_ItemNotOpen
: The device specified by the dev argument is not open.

NOMEM
: There was not enough memory to complete the operation.

Implementation

Convenience call implemented in clib.lib V20.

Associated Files

io.h
: ANSI C Prototype

clib.lib
: ARM Link Library

Notes

When you no longer need an I/O request, use `DeleteIOReq()` to delete it.

You can use `FindNamedItem()` to find a I/O request by name. When creating I/O requests, you should assign unique names whenever possible.

The kernel may change the priority of an I/O request to help optimize throughput.

See Also

`DeleteIOReq()`

DeleteIOReq

Deletes an I/O request.

Synopsis

```
Err DeleteIOReq( Item item )
```

Description

This macro deletes an I/O request item. You can use this macro in place of `DeleteItem()` to delete the item. If there was any outstanding I/O with this IOReq, it will be aborted first.

Arguments

item
: The item number of the I/O request to delete.

Return Value

The macro returns 0 if the I/O request was successfully deleted or an error code (a negative value) if an error occurs.

Implementation

Macro implemented in io.h V20.

Associated Files

io.h
: ANSI C Macro

See Also

`CreateIOReq()`

DoIO

Performs synchronous I/O.

Synopsis

```
Err DoIO( Item ior, const IOInfo *ioiP )
```

Description

This procedure is the most efficient way to perform synchronous I/O (I/O in which the task waits for the I/O request to complete before continuing). It automatically requests quick I/O (see below), sends the I/O request, and then puts the calling task into wait state until the request is complete.

`DoIO()` automatically sets the IO_QUICK flag in the IOInfo structure. This flag specifies quick I/O, which works as follows: The kernel tries to perform the I/O operation immediately and, if it is successful, it sends back the resulting IOReq immediately and returns 1 as the result code. The calling task can then get the necessary information from the IOReq. If quick I/O is not successful, the kernel performs normal asynchronous I/O and notifies the task with a signal or message when the I/O request is complete. This wakes up the task and DoIO() returns.

Starting with kernel folio V24, this function will automatically sample the io_Error field of the I/O request and return this to you. It is, therefore, no longer necessary to have code such as:

```
err = DoIO(iorItem,&ioinfo);
0r err = DoIO(iorItem,&ioinfo);
    if (err >= 0)
        err = ior->io_Error;
```

You can now just look at the return value of `DoIO()`.

Arguments

ior
: The item number of the I/O request to use.

ioiP
: A pointer to the IOInfo structure containing the I/O command to be executed, input and/or output data, and other information.

Return Value

The procedure returns when the I/O was successful (which means that the IOReq structure has already been returned and its contents can be examined).

Implementation

Convenience call implemented in clib.lib V20. Became a SWI in kernel folio V24.

Associated Files

io.h
: ANSI C Prototype

See Also

AbortIO(), CheckIO(), CreateIOReq(), DeleteIOReq(), SendIO(), WaitIO()

SendIO

Requests asynchronous I/O.

Synopsis

```
Err SendIO( Item ior, const IOInfo *ioiP )
```

Description

This procedure sends an I/O request that is to be executed asynchronously. Because the request is asynchronous, control returns to the calling task as soon as the request is sent (unlike synchronous I/O, where control doesn't return until after the request has been completed).

Call this procedure after creating the necessary IOReq item (for the ior argument, created by calling `CreateIOReq()`) and an IOInfo structure (for the ioiP argument). The IOReq item specifies the device to which to send the request and the reply port, if any, while the IOInfo structure includes the I/O command to be executed and a variety of other information. For descriptions of the IOReq and IOInfo data structures, see the Data Structures and Variable Type's chapter.

To request quick I/O, set the IO_QUICK flag in the ioi_Flags field of the IOInfo structure. Quick I/O works as follows: The kernel tries to perform the I/O operation immediately; if it is successful, it sends back the resulting IOReq item immediately without setting any signal bits or sending a message. If quick I/O was successful, the IO_QUICK bit in the io_Flags field of the IOReq is set. If quick I/O was not successful, the kernel performs normal asynchronous I/O and notifies the task with a signal or message when the I/O request is complete.

The IOInfo structure must be fully initialized before calling this function. You can use the ioi_User field of the IOInfo structure to contain whatever you want. This is a useful place to store a pointer to contextual data that needs to be associated with the I/O request. If you use message-based notification for your I/O requests, the msg_DataSize field of the notification messages will contain the value of ioi_User from the IOInfo structure.

Arguments

ior

The item number of the IOReq structure for the request. This structure is normally created by calling CreateIOReq().

ioiP

A pointer to an IOInfo structure.

Return Value

The procedure returns 1 if the I/O was completed immediately or 0 if the I/O request is still in progress (the task will be notified with either a signal or message when the request is complete, depending on what you specified when you called `CreateIOReq()`). It returns an error code (a negative value) if an error occurs. Possible error codes include:

BADITEM
: The ior argument does not specify an IOReq.

NOTOWNER
: The I/O request specified by the ior argument is not owned by this task.

ER_IONotDone
: The I/O request is already in progress.

BADPTR
: A pointer is invalid: Either the IOInfo structure specified by the ioiP argument is not entirely within the task's memory, the IOInfo receive buffer (specified by the ioi_Recv field in the IOInfo structure) is not entirely within the task's memory, or the IOInfo send buffer (specified by the ioi_Send field in the IOInfo structure) is not in legal memory.

BADIOARG
: One or more reserved I/O flags are set (either reserved flags in the ioi_Flags field of the IOInfo structure or any of the flags in the ioi_Flags2 field of the IOInfo structure).

BADUNIT
: The unit specified by the ioi_Unit field of the IOInfo structure is not supported by this device.

If quick I/O occurs, the IO_QUICK flag is set in the io_Flags field of the IOReq structure.

If the ior and ioiP arguments were valid but an error occurred during the I/O operation, an error code is returned in the io_Error field of the IOReq structure. If `SendIO()` returns 0 and a error occurs during I/O, the IOReq is returned as if it were completed, and it contains the error code in io_Error of the IOReq structure.

Implementation

SWI implemented in kernel folio V20.

Associated Files

io.h
: ANSI C Prototype

Caveats

`SendIO()` returns only BADITEM, NOTOWNER, and ER_IONotDone. All other errors codes are returned in the io_Error field of the IOReq.

See Also

`AbortIO()`, `CheckIO()`, `DoIO()`, `WaitIO()`

WaitIO

Waits for an I/O request to complete.

Synopsis

```
Err WaitIO( Item ior )
```

Description

The procedure puts the calling task into wait state until the specified I/O request completes. When a task is in wait state, it uses no CPU time.

Note: If the I/O request has already completed, the procedure returns immediately. If the I/O request never completes and it is not aborted, the procedure never returns.

Starting with kernel folio V24, this function will automatically sample the io_Error field of the IO request, and return this to you. It is, therefore, no longer necessary to have code such as:

```
err = WaitIO(iorItem);
if (err > = 0)

err = ior->io_Error;
```

You can now just look at the return value of `WaitIO()`.

Arguments

ior
: The item number of the I/O request to wait for.

Return Value

The procedure returns a value greater than or equal to 0 if the I/O request was successful or an error code if an error occurs.

Implementation

Convenience call implemented in clib.lib V20. Became a SWI in kernel folio V24.

Associated Files

io.h
: ANSI C Prototype

clib.lib
: ARM Link Library

See Also

AbortIO(), CheckIO(), DoIO(), SendIO()

AddHead

Adds a node to the head of a list.

Synopsis

```
void AddHead( List *l, Node *n )
```

Description

This procedure adds a node to the head (the beginning) of the specified list.

Arguments

l
: A pointer to the list in which to add the node.

n
: A pointer to the node to add.

Implementation

Folio call implemented in kernel folio V20.

Associated Files

list.h
: ANSI C Prototype

clib.lib
: ARM Link Library

Notes

A node can be included in only one list.

Caveats

Attempting to insert a node into a list while it is a member of another list is not reported as an error, and may confuse the other list.

See Also

AddTail(), InitList(), InsertNodeFromHead(), InsertNodeFromTail(), RemHead(), RemNode(), RemTail(), UniversalInsertNode()

AddTail

Adds a node to the tail of a list.

Synopsis

```
void AddTail( List *l, Node *n )
```

Description

This procedure adds the specified node to the tail (the end) of the specified list.

Arguments

l
: A pointer to the list in which to add the node.

n
: A pointer to the node to add.

Implementation

Folio call implemented in kernel folio V20.

Associated Files

list.h
: ANSI C Prototype

clib.lib
: ARM Link Library

Notes

A node can be included only in one list.

Caveats

Attempting to insert a node into a list while it is a member of another list is not reported as an error, and may confuse the other list.

See Also

AddHead(), InitList(), InsertNodeFromHead(), InsertNodeFromTail(), RemHead(), RemNode(), RemTail(), UniversalInsertNode()

InitList

Initializes a list.

Synopsis

```
void InitList( List *l, const char *name )
```

Description

When you create a List structure, you must initialize it with a call to `InitList()` before using it. `InitList()` creates an empty list by initializing the head (beginning-of-list) and tail (end-of-list) anchors and by providing a name for a list.

Arguments

l
: A pointer to the list to be initialized.

name
: The name of the list, or NULL to get the default name.

Implementation

Folio call implemented in kernel folio V20.

Associated Files

list.h
: ANSI C Prototype

clib.lib
: ARM Link Library

Caveats

GIGO (garbage in, garbage out)

See Also

AddHead(), AddTail(), InsertNodeFromHead(), InsertNodeFromTail(), IsListEmpty(), RemHead(), RemNode(), RemTail(), UniversalInsertNode()

InsertNodeFromHead

Inserts a node into a list.

Synopsis

```
void InsertNodeFromHead( List *l, Node *n )
```

Description

This procedure inserts a new node into a list. The order of nodes in a list is normally determined by their priority. The procedure compares the priority of the new node to the priorities of nodes currently in the list, beginning at the head of the list, and inserts the new node immediately after all nodes whose priority is higher. If the priorities of all the nodes in the list are higher, the node is added at the end of the list.

Arguments

l
: A pointer to the list into which to insert the node.

n
: A pointer to the node to insert.

Implementation

Folio call implemented in kernel folio V20.

Associated Files

list.h
: ANSI C Prototype

clib.lib
: ARM Link Library

Notes

To arrange the nodes in a list by a value or values other than priority, use

`UniversalInsertNode()`.

A node can only be included in one list.

See Also

`AddHead()`, `AddTail()`, `RemHead()`, `RemNode()`, `RemTail()`, `InsertNodeFromTail()`, `UniversalInsertNode()`

RemHead

Removes the first node from a list.

Synopsis

```
Node *RemHead( List *l )
```

Description

This procedure removes the head (first) node from a list.

Arguments

l
: A pointer to the list to be beheaded.

Return Value

The procedure returns a pointer to the node that was removed from the list or NULL if the list is empty.

Implementation

Folio call implemented in kernel folio V20.

Associated Files

list.h
: ANSI C Prototype

clib.lib
: ARM Link Library

See Also

AddHead(), AddTail(), InsertNodeFromTail(), RemNode(), RemTail()

InsertNodeFromTail

Inserts a node into a list.

Synopsis

```
void InsertNodeFromTail( List *l, Node *n )
```

Description

This procedure inserts a new node into a list. The order of nodes in a list is determined by their priority. The procedure compares the priority of the new node to the priorities of nodes currently in the list, beginning at the tail of the list, and inserts the new node immediately before the nodes whose priority is lower. If there are no nodes in the list whose priority is lower, the node is added at the head of the list.

Arguments

l
: A pointer to the list into which to insert the node.

n
: A pointer to the node to insert.

Implementation

Folio call implemented in kernel folio V20.

Associated Files

list.h
: ANSI C Prototype

clib.lib
: ARM Link Library

Notes

To arrange the nodes in a list by a value or values other than priority, use

`UniversalInsertNode()`.

A node can only be included in one list.

See Also

`AddHead()`, `AddTail()`, `RemHead()`, `RemNode()`, `RemTail()`, `InsertNodeFromHead()`, `UniversalInsertNode()`

RemNode

Removes a node from a list.

Synopsis

```
void RemNode( Node *n )
```

Description

This procedure removes the specified node from a list.

Note: If the specified node structure is not in a list you may get an abort.

Arguments

n
: A pointer to the node to remove.

Implementation

Folio call implemented in kernel folio V20.

Associated Files

list.h
: ANSI C Prototype

clib.lib
: ARM Link Library

See Also

RemTail(), AddTail(), InsertNodeFromTail(), RemHead()

RemTail

Removes the last node from a list.

Synopsis

```
Node *RemTail( List *l )
```

Description

This procedure removes the tail (last) node from a list.

Arguments

l
: A pointer to the list structure that will have its tail removed.

Return Value

The procedure returns a pointer to the node that was removed from the list or NULL if the list is empty. The anchor nodes are never returned.

Implementation

Folio call implemented in kernel folio V20.

Associated Files

list.h
: ANSI C Prototype

clib.lib
: ARM Link Library

See Also

AddHead(), AddTail(), InsertNodeFromTail(), RemHead(), RemNode()

UniversalInsertNode

Inserts a node into a list.

Synopsis

```
void UniversalInsertNode( List *l, Node *n, bool (*f)(Node *n,Node *m)
)
```

Description

Every node in a list has a priority (a value from 0 to 255 that is stored in the n_Priority field of the node structure). When a new node is inserted with `InsertNodeFromHead()` or `InsertNodeFromTail()`, the position at which it is added is determined by its priority. The `UniversalInsertNode()` procedure allows tasks to arrange list nodes according to values other than priority.

`UniversalInsertNode()` uses a comparison function provided by the calling task to determine where to insert a new node. It compares the node to be inserted with nodes already in the list, beginning with the first node. If the comparison function returns TRUE, the new node is inserted immediately before the node to which it was compared. If the comparison function never returns TRUE, the new node becomes the last node in the list. The comparison function, whose arguments are pointers to two nodes, can use any data in the nodes for the comparison.

Arguments

l
: A pointer to the list into which to insert the node.

n
: A pointer to the node to insert. This same pointer is passed as the first argument to the comparison function.

f
: A comparison function provided by the calling task that returns TRUE if the node to be inserted (pointed to by the first argument to the function) should be inserted immediately before the node to which it is compared (pointed to by the second argument to the function).

m
: A pointer to the node in the list to which to compare the node to insert.

Implementation

Folio call implemented in kernel folio V20.

Associated Files

list.h
: ANSI C Prototype

clib.lib
: ARM Link Library

Notes

A node can be included only in one list.

See Also

AddHead(), AddTail(), InsertNodeFromHead(), InsertNodeFromTail(), RemHead(), RemNode(), RemTail()

IsListEmpty

Checks whether a list is empty.

Synopsis

```
bool IsListEmpty( List *l )
```

Description

This macro checks whether a list is empty.

Arguments

l
: A pointer to the list to check.

Return Value

The macro returns TRUE if the list is empty or FALSE if it isn't.

Implementation

Macro implemented in list.h V24.

Associated Files

list.h
: ANSI C Macro definition

See Also

FirstNode(), IsNode(), IsNodeB(), LastNode(), NextNode(), PrevNode(), ScanList()

FirstNode

Gets the first node in a list.

Synopsis

```
Node *FirstNode( List *l )
```

Description

This macro returns a pointer to the first node in the specified list. If the list is empty, the macro returns a pointer to the tail (end-of-list) anchor. To determine if the return value is an actual node rather than the tail anchor, call the `IsNode()` procedure.

Example 1: *Use of FirstNode in forward list traversal*

```
for (n = FirstNode(list); IsNode(list, n); n = NextNode( n ))
{    . . .    };
```

Arguments

l
: A pointer to the list from which to get the first node.

Return Value

The macro returns a pointer to first node in the list or, if the list is empty, a pointer to the tail (end-of-list) anchor.

Implementation

Macro implemented in list.h V20.

Associated Files

list.h
: ANSI C Macro

See Also

IsListEmpty(), IsNode(), IsNodeB(), LastNode(), NextNode(), PrevNode(), ScanList()

IsNode

Checks that a node pointer is not the head list anchor.

Synopsis

```
bool IsNode( const List *l, const Node *n )
```

Description

This macro is used to test whether the specified node is an actual node or is the tail (end-of-list) anchor. Use this macro when traversing a list from head to tail. When traversing a list from tail to head, use the `IsNodeB()` macro.

Arguments

l
: A pointer to the list containing the node to check.

n
: A pointer to the node to check.

Return Value

The macro returns TRUE if the node is an actual node or FALSE if it is the tail (end-of-list) anchor.

Note: This macro currently returns TRUE for any node that is not the tail anchor, whether or not the node is in the specified list.

Implementation

Macro implemented in list.h V20.

Associated Files

list.h
: ANSI C Macro Definition

See Also

FirstNode(), IsListEmpty(), IsNodeB(), LastNode(), NextNode(), PrevNode(), ScanList()

IsNodeB

Checks that a node pointer is not the tail list anchor.

Synopsis

```
bool IsNodeB( const List *l, const Node *n )
```

Description

This macro is used to test whether the specified node is an actual node or is the head (beginning-of-list) anchor. Use this macro when traversing a list from tail to head. When traversing a list from head to tail, use the `IsNode()` macro.

Arguments

l
: A pointer to the list containing the node to check.

n
: A pointer to the node to check.

Return Value

The macro returns TRUE if the node is an actual node or FALSE if it is the head (beginning-of-list) anchor.

Note: This macro currently returns TRUE for any node that is not the head anchor, whether or not the node is in the specified list.

Implementation

Macro implemented in list.h V20.

Associated Files

list.h
: ANSI C Macro Definition

See Also

FirstNode(), IsListEmpty(), IsNode(), LastNode(), NextNode(), PrevNode(), ScanList()

LastNode

Gets the last node in a list.

Synopsis

```
Node *LastNode( const List *l )
```

Description

This macro returns a pointer to the last node in a list. If the list is empty, the macro returns a pointer to the head (beginning-of-list) anchor. To determine if the return value is an actual node rather than the head anchor, call the `IsNodeB()` procedure.

Example 1: *Example 15-3 Use of LastNode in reverse list traversal*

```
for (n = LastNode(list); IsNodeB(list, n); n = PrevNode( n ))
{    . . .    };
```

Arguments

l
: A pointer to the list structure to be examined.

Return Value

The macro returns a pointer to last node in the list or, if the list is empty, a pointer to the head (beginning-of-list) anchor.

Implementation

Macro implemented in list.h V20.

Associated Files

list.h
: ANSI C Macro definition

See Also

FirstNode(), IsListEmpty(), IsNode(), IsNodeB(), NextNode(), PrevNode(), ScanList()

NextNode

Gets the next node in a list.

Synopsis

```
Node *NextNode( const Node *n )
```

Description

This macro gets a pointer to the next node in a list. If the current node is the last node in the list, the result is a pointer to the tail (end-of-list) anchor. To determine if the return value is an actual node rather than the tail anchor, call the `IsNode()` procedure.

Example 1: *Use of NextNode in forward list traversal*

```
for (n = FirstNode( l ); IsNode( l, n ); n = NextNode( n ))
{   . . .   };
```

Arguments

n
: Pointer to the current node.

Return Value

The macro returns a pointer to the next node in the list or, if the current node is the last node in the list, to the tail (end-of-list) anchor.

Implementation

Macro implemented in list.h V20.

Associated Files

list.h
: ANSI C Macro definition

Caveats

Assumes that n is a node in a list. If not, watch out.

See Also

FirstNode(), IsListEmpty(), IsNode(), IsNodeB(), LastNode(), PrevNode(), ScanList()

PrevNode

Gets the previous node in a list.

Synopsis

```
Node *PrevNode( const Node *node )
```

Description

This macro returns a pointer to the previous node in a list. If the current node is the first node in the list, the result is a pointer to the head (beginning-of-list) anchor. To determine whether the return value is an actual node rather than the head anchor, use the `IsNodeB()` procedure.

Example 1: *Use of PrevNode in reverse list traversal*

```
for (n = LastNode( l ); IsNodeB( l, n ); n = PrevNode( n ))
{   . . .   };
```

Arguments

node
: A pointer to the current node.

Return Value

The macro returns a pointer to the previous node in the list or, if the current node is the first node in the list, to the head (beginning-of-list) anchor.

Implementation

Macro implemented in list.h V20.

Associated Files

list.h
: ANSI C Macro definition

PrevNode

See Also

FirstNode(), IsListEmpty(), IsNode(), IsNodeB(), LastNode(), ScanList()

ScanList

Walks through all the nodes in a list.

Synopsis

```
ScanList ( const List *1, void *n, <node type>)
```

Description

This macro lets you easily walk through all the elements in a list from the first to the last.

Example 1: *For Example*

```
List       *l;
DataStruct *d;
uint32      i;

i = 0;
ScanList(l,d,DataStruct)
{
    printf("Node %d is called %s\n",i,d->d.n_Name);
    i++;
}
```

Arguments

l
: A pointer to the list to scan.

n
: A variable which will be altered to hold a pointer to every node in the list in succession.

<node type>
: The data type of the nodes on the list. This is used for type casting within the macro.

Implementation

Macro implemented in list.h V22.

Associated Files

list.h
: ANSI C Macro definition

See Also

ScanListB()

ScanListB

Walks through all the nodes in a list backwards.

Synopsis

```
ScanListB(List *l, void *n, )
```

Description

This macro lets you easily walk through all the elements in a list from the last to the first.

Example 1: *For Example*

```
List       *l;
DataStruct *d;

uint32      i;
i = 0;
ScanListB(l,d,DataStruct)

{
printf("Node %d (counting from the end) is called %s=n".o.d-
>d.n_Name=;

i++;

}
```

Arguments

```
l
     A pointer to the list to scan.

n
     A variable which will be altered to hold a pointer to every node
     in the list in succession.

<node type>
     The data type of the nodes on the list. This is used for type
```

casting within the macro.

Implementation

Macro implemented in list.h V24.

Associated Files

list.h
ANSI C Macro definition

See Also

ScanList()

AllocMem

Allocate a memory block of a specific type.

Synopsis

```
void *AllocMem( int32 s, uint32 t )
```

Description

This macro allocates a memory block of a specific type.

Arguments

s
: The size of the memory block to allocate, in bytes.

t
: Flags that specify the type, alignment, and fill characteristics of memory to allocate.

One of the following flags must be set:

MEMTYPE_ANY
: Any memory can be allocated.

MEMTYPE_VRAM
: Allocate only video random-access memory (VRAM).

MEMTYPE_DRAM
: Allocate only dynamic random-access memory (DRAM).

If a block of VRAM must come from a specific VRAM bank, the following flag must be set:

MEMTYPE_BANKSELECT
: Allocate VRAM from a specific VRAM bank. In addition, one of the following two VRAM bank selection flags must be set:

MEMTYPE_BANK1
: Allocate only memory from VRAM bank 1.

MEMTYPE_BANK2
: Allocate only memory from VRAM bank 2.

The following flags are provided for compatibility with future hardware. You can set them in addition to the preceding flags.

MEMTYPE_DMA
: Allocate only memory that is accessible via direct memory access (DMA). Currently, all memory is accessible via DMA, but this may not be true in future hardware. Set this flag if you know the memory must be accessible via DMA.

MEMTYPE_CEL
: Allocate only memory that is accessible to the cel engine. Currently, all memory is accessible to the cel engine, but this may not be true in future hardware. Set this flag if you know the memory will be used for graphics.

MEMTYPE_AUDIO
: Allocate only memory that can be used for audio data (such as digitized sound). Currently, all memory can be used for audio, but this may not be true in future hardware. Set this flag if you know the memory will be used for audio data.

MEMTYPE_DSP
: Allocate only memory that is accessible to the digital signal processor (DSP). Currently, all memory is accessible to the DSP, but this may not be true in future hardware. Set this flag if you know the memory must be accessible to the DSP.

The following flags specify alignment, fill, and other allocation characteristics:

MEMTYPE_FILL
: Set every byte in the memory block to the value of the lower eight bits of the flags. If this flag is not set, the previous contents of the memory block are not changed. (Using 0 as the fill value can be useful for debugging: Any memory that is inadvertently changed can easily be detected.)

MEMTYPE_INPAGE
: Allocate a memory block that does not cross page boundaries.

MEMTYPE_STARTPAGE
: Allocate a memory block that starts on a page boundary.

MEMTYPE_MYPOOL
: Specifies that the memory block must be allocated only from the task's free memory pool. This

means that if there is not sufficient memory in the task's pool, the kernel must not allocate additional memory from the system-wide free memory pool (see Notes).

MEMTYPE_SYSTEMPAGESIZE
: Use the system's protection page size for that type of memory and not the special feature page size. This is necessary for MEMTYPE_VRAM to distinguish between allocations on page boundaries for graphics operations (normally 2 K) and task page sizes (normally 16 K).

Return Value

The procedure returns a pointer to the memory block that was allocated or NULL if the memory couldn't be allocated.

Implementation

Macro implemented in mem.h V20.

Associated Files

mem.h
: C Macro Definition

Notes

Use `FreeMem()` to free a block of memory that was allocated with `AllocMem()`.

If there is insufficient memory in a task's free memory pool to allocate the requested memory, the kernel automatically transfers the necessary pages of additional memory from the system-wide free memory pool to the task's free memory pool. The only exceptions are (1) when there is not enough memory in both pools together to satisfy the request, or (2) when the MEMTYPE_MYPOOL memory flag-which specifies that the memory block must be allocated only from the task's free memory pool-is set.

You can enable memory debugging in your application by compiling your entire project with the MEMDEBUG value defined on the compiler's command-line. Refer to the `CreateMemDebug()` function for more details.

See Also

`AllocMemBlocks()`, `AllocMemFromMemList()`, `AllocMemList()`, `ControlMem()`, `FreeMem()`, `FreeMemList()`, `FreeMemToMemList()`, `malloc()`, `ScavengeMem()`,

AllocMem

AllocMemBlocks

Transfers pages of memory from the system-wide free memory pool.

Synopsis

```
void *AllocMemBlocks( int32 size, uint32 typebits )
```

Description

When there is insufficient memory in a task's free memory pool to allocate a block of memory, the kernel automatically provides additional memory pages from the system-wide free memory pool. Tasks can also get pages from the system-wide free memory pool by calling `AllocMemBlocks()`.

Note: Normal applications do not need to call this procedure. It should only be used by applications that need additional control over the memory-allocation process.

You must set MEMTYPE_TASKMEM in the typebits argument otherwise the memory will not be allocated to the current task.

`AllocMemBlocks()` is different from other memory-allocation procedures:

- The pages of memory that are transferred are not automatically added to the task's free memory pool. To move the memory into its free memory pool, thereby making it available to tasks, the task must call one of the procedures for freeing memory (`FreeMem()`, `FreeMemToMemList()`, or `FreeMemToMemLists())` with the pointer returned by `AllocMemBlocks()` as the argument. (Note that in the memory returned by `AllocMemBlocks()`, the first four bytes specify the amount of memory, in bytes, that was transferred. You should use this value as the size to be freed.)

Arguments

size
: The amount of memory to transfer, in bytes. If the size is not an integer multiple of the page size for the type of memory requested, the system transfers the number of full pages needed to satisfy the request.

typebits
: Flags that specify the type of memory to transfer. These flags can include MEMTYPE_ANY, MEMTYPE_VRAM, MEMTYPE_DRAM, MEMTYPE_BANKSELECT, MEMTYPE_BANK1,

MEMTYPE_BANK2, MEMTYPE_DMA, MEMTYPE_CEL, MEMTYPE_AUDIO, MEMTYPE_DSP, MEMTYPE_TASKMEM. For information about these flags, see the description of AllocMem().
MEMTYPE_TASKMEM must always be set.

Return Value

The procedure returns a pointer to the pages of memory that were transferred or NULL if the memory couldn't be transferred. The first four bytes of the memory specify the amount of memory that was transferred, in bytes.

Implementation

SWI implemented in kernel folio V20.

Associated Files

mem.h
ARM C "swi" declaration

Notes

To return memory to the system-wide free memory pool, use `ScavengeMem()` or `ControlMem()`. `ScavengeMem()` finds pages of memory in the task's free memory pool from which no memory has been allocated and returns those pages to the system-wide memory pool. You can use `ControlMem()` to transfer ownership of memory to the system-wide memory pool.

See Also

`ControlMem()`, `FreeMem()`, `FreeMemToMemList()`, `FreeMemToMemLists()`, `ScavengeMem()`

ControlMem

Controls memory permissions and ownership.

Synopsis

```
Err ControlMem( void *p, int32 size, int32 cmd, Item task )
```

Description

When a task allocates memory, it becomes the owner of that memory. Other tasks cannot write to the memory unless they are given permission by its owner. A task can give another task permission to write to one or more of its memory pages, revoke write permission that was previously granted, or transfer ownership of memory to another task or the system by calling `ControlMem()`.

Each page of memory has a control status that specifies which task owns the memory and which tasks can write to it. Calls to `ControlMem()` change the control status for entire pages. If the p and size arguments (which specify the memory to change) specify any part of a page, the changes apply to the entire page.

A task can grant write permission for a page that it owns (or for some number of contiguous pages) to any number of tasks. To accomplish this, the task must make a separate call to `ControlMem()` for each task that is to be granted write permission.

A task that calls `ControlMem()` must own the memory whose control status it is changing, with one exception: A task that has write access to memory it doesn't own can relinquish its write access (by using MEMC_NOWRITE as the value of the cmd argument). If a task transfers ownership of memory it still retains write access.

Arguments

p
: A pointer to the memory whose control status to change.

size
: The amount of memory for which to change the control status, in bytes. If the size and p arguments specify any part of a page, the control status is changed for the entire page.

cmd
: A constant that specifies the change to be made to the control status; possible values are listed

below.

task
: The item-number task for which to change the control status or zero for global changes (see "Notes").

The possible values of "cmd" are:

MEMC_OKWRITE
: Grants permission to write to this memory to the task specified by the task argument.

MEMC_NOWRITE
: Revokes permission to write to this memory from the task specified by the task argument. If task is 0 revokes write permission for all tasks.

MEMC_GIVE
: If the calling task is the owner of the memory, this transfers ownership of the memory to the task specified by the task argument. If the specified task is 0, it gives the memory back to the system free memory pool.

Return Value

The procedure returns 0 if the change was successful or an error code (a negative value) if an error occurs. Possible error codes include:

BADITEM
: The task argument does not specify a current task.

ER_Kr_BadMemCmd
: The cmd argument is not one of the valid values.

ER_BadPtr
: The p argument is not a valid pointer to memory.

Implementation

SWI implemented in kernel folio V20.

Associated Files

mem.h

ANSI C Prototype

Notes

A task can use `ControlMem()` to prevent itself from writing to memory it owns.

A task must own the memory for its I/O buffers.

A task can use `ControlMem()` to return ownership of memory pages to the system, thereby returning them to the system-wide free memory pool. You can do this by using 0 as the value of the task argument.

A task can use `ControlMem()` to unshare or write protect memory from all other tasks. Specify 0 for the task for this to happen.

We would like to support making a piece of memory writable by all other tasks by using task==0 and MEMC_OKWRITE, but this is not implemented yet.

See Also

`ScavengeMem()`, `AllocMemBlocks()`

ScavengeMem

Returns task's free memory pages to the system memory pool.

Synopsis

```
int32 ScavengeMem( void )
```

Description

This procedure finds pages of memory in the task's free memory pool from which no memory is allocated and returns those pages to the system-wide memory pool.

If there is not enough memory in a task's free memory pool to satisfy a memory-allocation request, `AllocMem()` tries to get reclaim the necessary memory by calling `ScavengeMem()`.

Return Value

The procedure returns the amount of memory that was returned to the system-wide memory pool, in bytes. If no memory was returned, the procedure returns 0.

Implementation

Folio call implemented in kernel folio V20.

Associated Files

mem.h
: ANSI C Prototype

clib.lib
: ARM Link Library

See Also

`AllocMem()`, `AllocMemFromMemLists()`, `FreeMem()`, `FreeMemToMemLists()`

AllocMemFromMemLists

Allocates a memory block from a list of memory lists.

Synopsis

```
void *AllocMemFromMemLists( List *l, int32 size, uint32 memflags)
```

Description

This procedure allocates a memory block of the specified type from a list of free memory lists. Although it is used most often by the kernel, it can also be called by user tasks that need to do their own memory management.

Note: Most applications do not need to do their own memory management. When you use standard memory-allocation procedures like `AllocMem()`, the details of memory management are handled for you by the kernel.

The free memory pools in Portfolio are implemented as lists of memory lists:

- The system-wide free memory pool is the list of memory lists that contain the free memory pages owned by the system.
- A task's free memory pool is a list of memory lists that have been allocated for the task. These include (1) the two memory lists - one for VRAM and one for DRAM -that are allocated automatically for a task when the task is created.

`AllocMemFromMemLists()` allocates memory directly from a particular memory pool. This is in contrast to `AllocMemFromMemList()`, which allocates memory from a specific memory list, typically one that was created by the task for doing its own memory management.

Note: Tasks can only allocate memory from memory lists that belong to them. This means that user tasks cannot use `AllocMemFromMemLists()` to allocate memory directly from the system-wide free memory pool, because the memory in that pool belongs to the system.

If a task requests more memory from its free memory pool than is available, the kernel automatically allocates the necessary additional memory pages from the system-wide free memory pool if they are available. The task gets ownership of the additional memory and, when the memory is later freed, it is added to the task's free memory pool.

Arguments

l
: A pointer to the memory pool from which to allocate the block.

size
: The size of the memory block to allocate, in bytes.

memflags
: Flags that specify the type of memory to allocate. These flags can include MEMTYPE_ANY, MEMTYPE_VRAM, MEMTYPE_DRAM, MEMTYPE_BANKSELECT, MEMTYPE_BANK1, MEMTYPE_BANK2, MEMTYPE_DMA, MEMTYPE_CEL, MEMTYPE_AUDIO, MEMTYPE_DSP, MEMTYPE_FILL, MEMTYPE_INPAGE, MEMTYPE_TRACKSIZE, MEMTYPE_STARTPAGE, and MEMTYPE_MYPOOL. For information about these flags, see the description of AllocMem().

Return Value

The procedure returns a pointer to memory block that was allocated or NULL if the memory couldn't be allocated.

Implementation

Folio call implemented in kernel folio V20.

Associated Files

mem.h
: ANSI C Prototype

clib.lib
: ARM Link Library

Notes

To free a memory block allocated with `AllocMemFromMemLists()`, use the `FreeMemToMemLists()` procedure.

You can enable memory debugging in your application by compiling your entire project with the MEMDEBUG value defined on the compiler's command-line. Refer to the `CreateMemDebug()` function for more details.

See Also

`AllocMemFromMemLists()`

FreeMem

Frees memory from AllocMem().

Synopsis

```
void FreeMem( void *p, int32 size )
```

Description

This macro frees memory that was previously allocated by a call to `AllocMem()`. The size argument specifies the number of bytes to free. The freed memory is automatically returned to the memory list from which it was allocated.

Arguments

p
: A pointer to the memory block to free. This value may be NULL, in which case this function just returns.

size
: The size of the block to free, in bytes. This must be the same size that was specified when the block was allocated. If the memory being freed was allocated using MEMTYPE_TRACKSIZE, this argument should be set to -1.

Implementation

Macro implemented in mem.h V20.

Associated Files

mem.h
: ANSI C Macro definition

Notes

You can enable memory debugging in your application by compiling your entire project with the MEMDEBUG value defined on the compiler's command-line. Refer to the `CreateMemDebug()`

function for more details.

See Also

AllocMem()

FreeMemToMemLists

Returns memory to the free pool.

Synopsis

```
void FreeMemToMemLists( List *l, void *p, int32 size )
```

Description

The procedure returns a block of memory that was allocated with `AllocMemFromMemLists()` to the specified free memory pool (a list of memory lists).

Note: Unless you are trying to move memory from one memory pool to another, you should always free memory to the same pool that you obtained it from.

Arguments

l
: A pointer to the memory pool (a list of memory lists) to which to return the memory block.

p
: A pointer to the memory to free. This value may be NULL, in which case this function just returns.

size
: Number of bytes to free. This must be the same size that was passed to AllocMemFromMemLists() to allocate the block. See FreeMem for additional information.

Implementation

Folio call implemented in kernel folio V20.

Associated Files

mem.h
: ANSI C Prototype

clib.lib
: ARM Link Library

Notes

You can enable memory debugging in your application by compiling your entire project with the MEMDEBUG value defined on the compiler's command-line. Refer to the `CreateMemDebug()` function for more details.

See Also

`AllocMem()`, `AllocMemFromMemLists()`, `FreeMem()`

FreeMemToMemList

Frees a private block of memory.

Synopsis

```
void FreeMemToMemList( MemList *ml, void *p, int32 size )
```

Description

This procedure frees a private block of memory that was previously allocated by a call to `AllocMemFromMemList()`. You can also use it to add memory to a private memory list created by `AllocMemList()`.

Note: The memory you free to a memory list must be either memory that was previously allocated from the list or (when adding new memory to the list) memory of the type specified (by the p argument of `AllocMemList()`) when the memory list was created.

The block of memory is expected to begin and end on nicely aligned bounds; the alignment size can be obtained by calling `GetMemAllocAlignment()`. If the block is not properly aligned, the base is rounded up and the size is rounded down until alignment is achieved.

Arguments

ml
: A pointer to the memory list to which to free the memory.

p
: A pointer to the memory to free. This value may be NULL, in which case this function just returns.

size
: The number of bytes to free. This must be the same size that was passed to AllocMemFromMemList() to allocate the block. See FreeMem for additional information.

Implementation

Folio call implemented in kernel folio V20.

Associated Files

mem.h
: ANSI C Prototype

clib.lib
: ARM Link Library

See Also

`AllocMemFromMemList()`, `AllocMemList()`, `FreeMemList()`, `GetMemAllocAlignment()`

AllocMemFromMemList

Allocates memory from a private memory pool.

Synopsis

```
void *AllocMemFromMemList( MemList *ml, int32 size, uint32 memflags)
```

Description

A task can do its own memory management by creating one or more private memory lists. Use this procedure to allocate a memory block from a private memory list.

Note: Most applications do not need to do their own memory management. When you use standard memory-allocation procedures like `AllocMem()`, the details of memory management are handled for you by the kernel.

To create a private memory list, use the `AllocMemList()` procedure.

Arguments

ml
: A pointer to the private memory list from which to allocate the memory block.

size
: The size of the memory block to allocate, in bytes.

memflags
: Flags that specify the type of memory to allocate. These flags include MEMTYPE_ANY, MEMTYPE_VRAM, MEMTYPE_DRAM, MEMTYPE_BANKSELECT, MEMTYPE_BANK1, MEMTYPE_BANK2, MEMTYPE_DMA, MEMTYPE_CEL, MEMTYPE_AUDIO, MEMTYPE_DSP, MEMTYPE_FILL, MEMTYPE_INPAGE, MEMTYPE_TRACKSIZE, and MEMTYPE_STARTPAGE. For information about these flags, see the description of AllocMem().

Return Value

The procedure returns a pointer to the allocated memory or NULL if the memory couldn't be allocated.

Implementation

Folio call implemented in kernel folio V20.

Associated Files

mem.h
: ANSI C Prototype

clib.lib
: ARM Link Library

Notes

To free a memory block allocated with `AllocMemFromMemList()`, use the `FreeMemToMemList()` procedure.

See Also

`AllocMemList()`, `FreeMemToMemList()`

AllocMemList

Creates a private memory list.

Synopsis

```
MemList *AllocMemList( const void *p, char *name )
```

Description

A task can do its own memory management by creating one or more private memory lists. Use this procedure to allocate a private memory list.

A memory list you create with `AllocMemList()` is initially empty. To add memory to the list, use the `FreeMemToMemList()` procedure.

Note: A single memory list can store either DRAM or VRAM, but not both. The p argument points to an example of the memory the task wants to manage with this MemList.

Arguments

p
: A pointer to a memory address whose memory type (either DRAM or VRAM) is the type you want to store in the list. You control actual type of memory in the list by controlling what you free into the list.

name
: The optional name of the memory list.

Return Value

The procedure returns a pointer to the MemList structure that is created or NULL if an error occurs.

Implementation

Folio call implemented in kernel folio V20.

Associated Files

mem.h
: ANSI C Prototype

clib.lib
: ARM Link Library

Notes

To deallocate a memory list created with `AllocMemList()`, use `FreeMemList()`.

See Also

`AllocMemFromMemList()`, `FreeMemList()`, `FreeMemToMemList()`

FreeMemList

Frees a memory list.

Synopsis

```
void FreeMemList( MemList *ml )
```

Description

This procedure frees a memory list that was allocated by a call to `AllocMemList()`.

Arguments

ml
: A pointer to the memory list to free. This value may be NULL, in which case this function just returns.

Implementation

Folio call implemented in kernel folio V20.

Associated Files

mem.h
: ANSI C Prototype

clib.lib
: ARM Link Library

Caveats

If the list is not empty, any memory it contains is lost.

See Also

`AllocMemList()`

FreeMemList

GetMemAllocAlignment

Gets the memory-allocation alignment size

Synopsis

```
int32 GetMemAllocAlignment( uint32 memflags )
```

Description

This function returns the alignment guaranteed by AllocMem() and friends, and required by FreeMem() and friends.

Arguments

memflags
: Similar to the usage in GetPageSize(); different allocators may be in use for different kinds of memory, which may have different alignment guarantees and restrictions.

Return Value

The return value is the alignment modulus for the memory specified; for example, if AllocMem() guarantees (and FreeMem() requires) long-word alignment, it returns 4. If no alignment is guaranteed or required, it returns 1.

Implementation

Folio call implemented in kernel folio V22.

Associated Files

mem.h
: ANSI C Prototype

clib.lib
: ARM Link Library

Caveat

Note that `malloc()` does not necessarily provide same alignment guarantee as AllocMem(), as `malloc()` reserves some space at the front of the block it gets from AllocMem() to contain the size of the block allocated.

See Also

`GetMemType()`, `AllocMem()`, `FreeMem()`, `malloc()`, `free()`

GetMemType

Gets the type of the specified memory.

Synopsis

```
uint32 GetMemType( const void *p )
```

Description

The procedure returns flags that describe the type of memory at a specified memory location.

Arguments

p
: A pointer to the memory whose type to return.

Return Value

The procedure returns a set of flags that specify the type of memory. For information about these flags, see the description of `AllocMem()`.

Implementation

Convenience call implemented in clib.lib V20.

Associated Files

mem.h
: ANSI C Prototype

clib.lib
: ARM Link Library

See Also

`AllocMem()`

malloc

Allocates memory.

Synopsis

```
void *malloc( int32 size )
```

Description

This procedure allocates a memory block. It is identical to the `malloc()` procedure in the standard C library. The memory is guaranteed only to be memory that is accessible to the CPU; you cannot specify the memory type (such as VRAM), alignment (such as a memory block that begins on a page boundary), or any other memory characteristics.

Note: You should only use `malloc()` when porting existing C programs to Portfolio. If you are writing programs specifically for Portfolio, use `AllocMem()`, which allows you to specify the type of memory to allocate, such as VRAM or memory that begins on a page boundary.

Arguments

size
: Size of the memory block to allocate, in bytes.

Return Value

The procedure returns a pointer to the memory that was allocated or NULL if the memory couldn't be allocated.

Implementation

Convenience call implemented in clib.lib V20.

Associated Files

stdlib.h
: ANSI C Prototype

clib.lib
: ARM Link Library

Notes

You can enable memory debugging in your application by compiling your entire project with the MEMDEBUG value defined on the compiler's command-line. Refer to the `CreateMemDebug()` function for more details.

See Also

`AllocMem()`, `AvailMem()`, `free()`

AvailMem

Gets information about available memory.

Synopsis

```
void AvailMem(MemInfo *minfo, uint32 memflags);
```

Description

This procedure returns information about the amount of memory that is currently available. You can get information about a particular kind of memory by setting the corresponding flags, such as MEMTYPE_VRAM or MEMTYPE_DRAM, in the flags argument. To get information about all memory that is available to the CPU, use MEMTYPE_ANY as the value of the flags argument.

The information about available memory is returned in a MemInfo structure:

Example 1: *Memory Information Structure for*

```
typedef struct MemInfo
{
uint32 minfo_SysFree;
uint32 minfo_SysLargest;
uint32 minfo_TaskFree;
uint32 minfo_TaskLargest;
} MemInfo;
```

The fields contain the following information:

minfo_SysFree
: The amount of memory of the specified memory type in the system-wide free memory pool, in bytes. The pool contains only full pages of memory.

minfo_SysLargest
: The size, in bytes, of the largest series of contiguous pages of the specified memory type in the system-wide free memory pool.

minfo_TaskFree
: The amount of memory of the specified type in the task's free memory pool, in bytes.

minfo_TaskLargest
: The size, in bytes, of the largest contiguous block of the specified memory type that can be allocated from the task's free memory pool.

Arguments

minfo
: A pointer to the MemInfo structure used to return the result.

memflags
: Flags that specify the type of memory to get information about. These flags can include MEMTYPE_ANY, MEMTYPE_VRAM, MEMTYPE_DRAM, MEMTYPE_BANKSELECT, MEMTYPE_BANK1, MEMTYPE_BANK2, MEMTYPE_DMA, MEMTYPE_CEL, MEMTYPE_AUDIO, and MEMTYPE_DSP. For information about these flags, see the description of AllocMem().

Implementation

Library routine implemented in clib.lib V20.

Associated Files

mem.h
: ANSI C Prototype

clib.lib
: ARM Link Library

Caveats

When you call `AvailMem()`, you must request information about only one memory type. Attempting to find out about more than one memory type may produce unexpected results.

If you pass in a garbage minfo pointer, sparks may fly.

The information returned by `AvailMem()` is inherently flawed, since you are existing in a multitasking environment. Memory can be allocated or freed asynchronous to the operation of the task calling `AvailMem()`.

See Also

AllocMem(), free(), FreeMem(), malloc()

---

free

Frees memory from malloc().

Synopsis

```
void free( void *p )
```

Description

The procedure frees a memory block that was allocated by a call to `malloc()`. It is identical to the `free()` procedure in standard C library. The freed memory is automatically returned to the memory list from which it was allocated.

Note: You should only use `malloc()` and `free()` when porting existing C programs to Portfolio. If you are writing programs specifically for Portfolio, use `AllocMem()` and `FreeMem()`, which allow you to specify the type of memory to allocate, such as VRAM or memory that begins on a page boundary.

Arguments

p
: A pointer to the memory to be freed. This pointer may be NULL, in which case this function just returns.

Implementation

Convenience call implemented in clib.lib V20.

Associated Files

stdlib.h
: ANSI C Prototype

clib.lib
: ARM Link Library

Notes

You can enable memory debugging in your application by compiling your entire project with the MEMDEBUG value defined on the compiler's command-line. Refer to the `CreateMemDebug()` function for more details.

Caveats

Trusts that p is a result of a previous `malloc()` call.

See Also

`FreeMem()`, `FreeMemToMemList()`, `FreeMemToMemLists()`, `malloc()`

AllocSignal

Allocates signals.

Synopsis

```
int32 AllocSignal( uint32 sigMask )
```

Description

One of the ways tasks communicate is by sending signals to each other. Signals are 1-bit flags that indicate that a particular event has occurred.

Tasks that send and receive signals must agree on which signal bits to use and the meanings of the signals. Except for system signals, there are no conventions for the meanings of individual signal bits; it is up to software developers to define their meanings.

You allocate bits for new signals by calling `AllocSignal()`. To define one signal at a time - by far the most common case - call `AllocSignal()` with 0 as the argument:

theSignal = AllocSignal(0)

This allocates the next unused bit in the signal word. In the return value, the bit that was allocated is set. If the allocation fails (which happens if all the non-reserved bits in the signal word are already allocated), the procedure returns 0.

In rare cases, you may need to define more than one signal with a single call. You do this by creating a uint32 value and setting any bits you want to allocate for new signals, then calling `AllocSignal()` with this value as the argument. If all the signals are successfully allocated, the bits set in the return value are the same as the bits that were set in the argument.

Signals are implemented as follows:

- Each task has a 32-bit signal mask that specifies the signals it understands. Tasks allocate bits for new signals by calling `AllocSignal()`. The bits are numbered from 0 (the least-significant bit) to 31 (the most-significant bit). Bits 0 through 7 are reserved for system signals (signals sent by the kernel to all tasks); remaining bits can be allocated for other signals. Note: Bit 31 is also reserved for the system. It is set when the kernel returns an error code to a task instead of signals. For example, trying to allocate a system signal or signal number 31.
- A task calls `SendSignal()` to send one or more signals to another task. Each bit set in the

signalWord argument specifies a signal to send. Normally, only one signal is sent at a time.

- When `SendSignal()` is called, the kernel gets the incoming signal word and ORs it into the received signal mask of the target task. If the task was in the wait queue, it compares the received signals with the WaitSignalMask. If there are any bits set in the target, the task is moved from the wait queue to the ready queue.
- If the SIGF_ABORT system signal is sent to the task, the corresponding bit in the task's signal word is automatically set. This signal cannot be masked.
- A task gets incoming signals by calling `WaitSignal()`. If any bits are set in the task's signal word , `WaitSignal()` returns immediately. If no bits are set in the task's signal word, the task remains in wait state until a signal arrives that matches one of the signals the task is waiting for.

The following system signals are currently defined:

SIGF_ABORT
: Informs the task that the current operation has been aborted.

SIGF_IODONE
: Informs the task that an asynchronous I/O request is complete.

SIGF_DEADTASK
: Informs the task that one of its child tasks or threads has been deleted. Note: This signal does not specify which task was deleted. To find this out, the parent task must check to see which of its child tasks still exist.

SIGF_SEMAPHORE
: Informs a task waiting to lock a semaphore that it can do so. Note: This signal is for system internal use only.

Arguments

signalMask
: An uint32 value in which the bits to be allocated are set, or 0 to allocate the next available bit. You should use 0 whenever possible (see Notes).

Return Value

The procedure returns a int32 value with bits set for any bits that were allocated or 0 if not all of the requested bits could be allocated. It returns ILLEGALSIGNAL if the signalMask specified a reserved signal.

Implementation

SWI implemented in kernel folio V20.

Associated Files

task.h
    ARM C Prototype

Notes

Use `FreeSignal()` to deallocate signals.

Future versions of Portfolio may define additional system signals. To help ensure that the signal bits you allocate in current applications do not conflict with future system signals, you should always use 0 as the value of the signalMask argument when allocating signals. If you must allocate specific signal bits (which is discouraged), use bits 17-31.

See Also

`FreeSignal()`, `GetCurrentSignals()`, `SendSignal()`, `WaitSignal()`

FreeSignal

Frees signals.

Synopsis

```
Err FreeSignal( uint32 sigMask )
```

Description

This procedure frees one or more signal bits allocated by `AllocSignal()`. The freed bits can then be reallocated.

For information about signals, see the description of the `AllocSignal()` procedure and the "Communicating Among Tasks" chapter in the 3DO Portfolio Programmer's Guide.

Arguments

sigMask
: A 32-bit value in which any signal bits to deallocate are set. The bits are numbered from 0 (the least-significant bit) to 31 (the most-significant bit). Bits 0 through 7 and bit 31 cannot be freed.

Return Value

The procedure returns 0 if the signal(s) were freed successfully or an error code if an error occurs.

Implementation

SWI implemented in kernel folio V20.

Associated Files

task.h
: ANSI C Prototype

See Also

`AllocSignal()`, `WaitSignal()`, `SendSignal()`

WaitSignal

Waits until a signal is received.

Synopsis

```
int32 WaitSignal( uint32 sigMask )
```

Description

This procedure puts the calling task into wait state until any of the signal(s) specified in the sigMask have been received. When a task is in wait state, it uses no CPU time.

When `WaitSignal()` returns, bits set in the result indicate which of the signal(s) the task was waiting for were received since the last call to `WaitSignal()`. (The SIGF_ABORTED bit is also set if that signal was received, even if it is not in the signal mask.) If the task was not waiting for certain signals, the bits for those signals remain set in the task's signal word, and all other bits in the signal word are cleared.

See `AllocSignal()` for a description of the implementation of signals.

Arguments

sigMask
: A mask in which bits are set to specify the signals the task wants to wait for.

Return Value

The procedure returns a mask that specifies which of the signal(s) a task was waiting for have been received or an error code (a negative value) if an error occurs. Possible error codes include:

ILLEGALSIGNAL
: One or more of the signal bits in the sigMask argument was not allocated by the task.

Implementation

SWI implemented in kernel folio V20.

Associated Files

task.h
ANSI C Prototype

Notes

Because it is possible for tasks to send signals in error, it is up to tasks to confirm that the actual event occurred when they receive a signal.

For example, if you were waiting for SIGF_IODONE and the return value from WaitSignal indicated that the signal was sent, you should still call CheckIO using the IOReq to make sure it is actually done. If it was not done you should go back to WaitSignal.

See Also

`SendSignal()`

SendSignal

Sends a signal to another task.

Synopsis

```
Err SendSignal( Item task, uint32 sigMask )
```

Description

This procedure sends one or more signals to the specified task. See the description of `AllocSignal()` for more information about signals.

It is an error for a user task to send a system signal or a signal that has not been allocated by the receiving task.

Arguments

task
: The item number of the task to send signals to. If this parameter is 0, then the signals are sent to the calling task. This is sometimes useful to set initial conditions.

sigMask
: The signals to send.

Return Value

The procedure returns 0 if successful or an error code (a negative value) if an error occurs. Possible error codes include:

BADPRIV
: The task attempted to send a system signal.

ILLEGALSIGNAL
: The task attempted to send a signal to a task that was not allocated by that task, or bit 32 in the sigMask argument (which is reserved by the system software) was set.

Implementation

SWI implemented in kernel folio V20.

Associated Files

task.h
: ANSI C Prototype

See Also

AllocSignal(), FreeSignal(), GetCurrentSignals(), WaitSignal()

GetCurrentSignals

Gets the currently received signal bits.

Synopsis

```
int32 GetCurrentSignals( void )
```

Description

This macro returns the signal bits that have been received for the current task. For information about signals, see the description of the `AllocSignal()` procedure and the "Communicating Among Tasks" chapter in the 3DO Portfolio Programmer's Guide.

Return Value

A 32-bit word in which all currently received signal bits are set.

Implementation

Macro implemented in kernel.h V20.

Associated Files

kernel.h
: ANSI C Macro

See Also

`AllocSignal()`, `FreeSignal()`, `SendSignal()`, `WaitSignal()`

CallFolio

Invokes from the folio vector table a folio procedure that doesn't return a value.

Synopsis

```
void CallFolio( const Folio *folio, int32 func, args )
```

Description

This macro allows a task to invoke a folio procedure directly from the folio vector table, thereby bypassing the normal procedure interface. It can be used only for folio procedures that do not return a value. (To invoke a folio procedure that does return a value, use `CallFolioRet()`.) This approach, which is slightly faster than invoking the procedure through the normal interface, is intended for use by the folios themselves, but it can be also be used by applications.

Note: Most tasks should invoke folio procedures in the normal way, using the interfaces described in this manual. Only tasks that require the utmost in speed and efficiency should invoke folio procedures directly from the vector table.

Example of AddTail(listP, nodeP) using CallFolio: CallFolio(KernelBase, KBV_ADDTAIL, (listP, nodeP));

Arguments

folio
: A pointer to the folio item that contains the procedure. Use LookupItem() to get this pointer. For the item number of a folio (which you pass to LookupItem() see the Portfolio Items chapter or call FindAndOpenFolio().

func
: The index of the vector table entry for the procedure. This index (which is always a negative integer, because the table grows backward in memory) is listed in the header file for the folio that contains the procedure.

args
: The arguments for the procedure, separated by commas and enclosed within parentheses.

Implementation

Macro implemented in folio.h V20.

Associated Files

folio.h

See Also

`CallFolioRet()`

CallFolioRet

Invokes from the folio vector table a folio procedure that returns a value.

Synopsis

```
void CallFolioRet( const Folio *folio, int32 func, args, ret, cast )
```

Description

This macro allows a task to invoke a folio procedure directly from the folio vector table, thereby bypassing the normal procedure interface. It can be used only for folio procedures that return a value. (To invoke a folio procedure that does not return a value, use `CallFolio()`.) This approach, which is slightly faster than invoking the procedure through the normal interface, is intended for use by the folios themselves, but it can be also be used by applications.

Note: Most tasks should invoke folio procedures in the normal way, using the interfaces described in this manual. Only tasks that require the utmost in speed and efficiency should invoke folio procedures directly from the vector table.

Example of n = RemTail(l) using CallFolioRet: CallFolioRet(KernelBase, KBV_REMTAIL, (l), n, (Node *));

Arguments

folio
: A pointer to the folio item that contains the procedure. Use LookupItem() to get this pointer. For the item number of a folio (which you pass to LookupItem()), see the Portfolio Items chapter or call FindAndOpenFolio().

func
: The index of the vector table entry for the folio procedure to invoke. This index (a negative integer, because the table grows backward in memory) is listed in the header file for the folio.

args
: The arguments for the procedure, separated by commas and enclosed within parentheses.

ret
: A variable to receive the result from the folio procedure. The result is coerced to the type specified by the cast argument before it is assigned to this variable.

cast

The desired type for the result, surrounded by parentheses. The result returned by the folio procedure is cast to this type.

Implementation

Macro implemented in folio.h V20.

Associated Files

folio.h

See Also

`CallFolio()`

CheckItem

Checks to see if an item exists.

Synopsis

```
void *CheckItem( Item i, uint8 ftype, uint8 ntype )
```

Description

This procedure checks to see if a specified item exists. To specify the item, you use an item number, an item-type number, and the item number of the folio in which the item type is defined. If all three of these values match those of the item, the procedure returns a pointer to the item.

Arguments

i
: The item number of the item to be checked.

ftype
: The item number of the folio that defines the item type. (This is the same value that is passed to the MkNodeID() macro.) For a list of folio item numbers, see the Portfolio Item's chapter.

ntype
: The item-type number for the item. (This is the same value that is passed to the MkNodeID() macro.) For a list of item-type numbers, see the Portfolio Item's chapter.

Return Value

If the item exists (and the values of all three arguments match those of the item), the procedure returns a pointer to the item. If the item does not exist, the procedure returns NULL.

Implementation

Folio call implemented in kernel folio V20.

Associated Files

item.h

ANSI C Prototype

clib.lib

ARM Link Library

See Also

CreateItem(), FindItem(), FindNamedItem(), FindVersionedItem(), LookupItem(), MkNodeID()

CreateItem

Creates an item.

Synopsis

```
Item CreateItem( int32 ct, TagArg *p )
Item CreateItemVA( int32 ct, uint32 tags, ... );
```

Description

This macro creates an item.

Note: There are convenience procedures for creating most types of items (such as `CreateMsg()` to create a message and `CreateIOReq()` to create an I/O request). You should use `CreateItem()` only when no convenience procedure is available or when you need to supply additional arguments for the creation of the item beyond what the convenience routine provides.

Arguments

ct
: Specifies the type of item to create. Use MkNodeID() to generate this value.

p
: A pointer to an array of tag arguments. The tag arguments can be in any order. The last element of the array must be the value TAG_END. If there are no tag arguments, this argument must be NULL. For a list of tag arguments for each item type, see the Portfolio Item's chapter.

Return Value

The procedure returns the item number of the new item or an error code if an error occurs.

Implementation

Macro implemented in item.h V20.

Associated Files

item.h

ANSI C Macro

Notes

When you no longer need an item created with `CreateItem()`, use `DeleteItem()` to delete it.

See Also

`CheckItem()`, `CreateIOReq()`, `CreateMsg()`, `CreateMsgPort()`, `CreateSemaphore()`, `CreateSmallMsg()`, `CreateThread()`, `DeleteItem()`

CreateMsg

Creates a standard message.

Synopsis

```
Item CreateMsg( const char *name, uint8 pri, Item mp )
```

Description

One of the ways tasks communicate is by sending messages to each other. This procedure creates an item for a standard message (a message where any data to be communicated to the receiving task is contained in a data block allocated by the sending task). You can use this procedure in place of `CreateItem()` to create the item.

To create a buffered message (a message that includes an internal buffer for sending data to the receiving task), use `CreateBufferedMsg()`. To create a small message (a message that can contain up to eight bytes of data), use `CreateSmallMsg()`.

Arguments

name
: The name of the message (see "Notes").

pri
: The priority of the message. This determines the position of the message in the receiving task's message queue and thus, how soon it is likely to be handled. A larger number specifies a higher priority.

mp
: The item number of the message port at which to receive the reply.

Return Value

The procedure returns the item number of the message or an error code (a negative value) if an error occurs.

Implementation

Convenience call implemented in clib.lib V20.

Associated Files

msgport.h
: ANSI C Prototype

clib.lib
: ARM Link Library

Notes

The same message item can be reused for any number of messages. When you are finished with a message item, use `DeleteMsg()` to delete it.

You can use `FindNamedItem()` to find a message item by name. When creating messages, you should assign unique names whenever possible.

See Also

`CreateMsgPort()`, `CreateBufferedMsg()`, `CreateSmallMsg()`, `DeleteMsg()`, `DeleteMsgPort()`, `SendMsg()`

CreateMsgPort

Creates a message port.

Synopsis

```
Item CreateMsgPort( const char *name, uint8 pri, uint32 signal )
```

Description

This convenience procedure creates a message port item. It also creates a message queue for the port in system RAM. You can use this procedure in place of `CreateItem()` to create the item.

Arguments

name
: The name of the message port (see "Notes").

pri
: The priority of the message port. The priority has no effect on the way message ports operate, but it does determine the order of the message ports in the global list of message ports and thus, the order in which a message ports are found.

signal
: Specifies the bit in the signal word that is set when a message arrives at this port. If zero is supplied a new signal will be allocated. (See the description of AllocSignal() for information about allocating signal bits.)

Return Value

The procedure returns the item number of the new message port or an error code if an error occurs.

Implementation

Convenience call implemented in clib.lib V20.

Associated Files

msgport.h
: ANSI C Prototype

clib.lib
: ARM Link Library

Notes

When you no longer need a message port use `DeleteMsgPort()` to delete it.

You can use `FindMsgPort()` to find a message port by name. When creating message ports you should assign unique names whenever possible.

See Also

`CreateMsg()`, `DeleteMsg()`, `DeleteMsgPort()`, `SendMsg()`, `SendSmallMsg()`

DeleteMsg

Deletes a message.

Synopsis

```
Err DeleteMsg( Item it )
```

Description

This macro deletes the specified message and any resources (including the internal data buffer for a buffered message) that were allocated for it.

Arguments

it
: The item number of the message to be deleted.

Return Value

The macro returns a value greater than or equal to 0 if successful or an error code if an error occurs.

Implementation

Macro implemented in msgport.h V20.

Associated Files

msgport.h
: ANSI C Macro

See Also

`CreateMsg()`

DeleteMsgPort

Deletes a message port.

Synopsis

```
Err DeleteMsgPort( Item it )
```

Description

This macro deletes the specified message port.

Arguments

it
: The item number of the message port to be deleted.

Return Value

The macro returns a value greater than or equal to 0 if successful or an error code if an error occurs.

Implementation

Macro implemented in msgport.h V20.

Associated Files

msgport.h
: ANSI C Macro

See Also

CreateMsg(), CreateMsgPort()

SendMsg

Sends a message.

Synopsis

```
Err SendMsg( Item mp, Item msg, const void *dataptr, int32 datasize )
```

Description

This procedure sends a message to the specified message port. (To send a small message use `SendSmallMsg()`.

The message is queued on the message port's list of messages according to its priority. Messages that have the same priority are queued on first come, first served basis.

Whether a message is standard or buffered is determined by the procedure you use to create the message: `CreateMsg()` creates a standard message (one whose message data belongs to the sending task; the receiving task reads from this block), while CreateBufferedMsg creates a buffered message (one in which a buffer for the message data is included in the message). For standard messages, the dataptr and datasize arguments refer to the data block owned by the message sender. For buffered messages, the dataptr and datasize arguments specify the data to be copied into the message's internal buffer before the message is sent.

If the message is a buffered message, `SendMsg()` checks the size of the data block to see whether it will fit in the message's buffer. If it won't fit, `SendMsg()` returns an error.

Arguments

mp
: The item number of the message port to which to send the message.

msg
: The item number of the message to send.

dataptr
: A pointer to the message data, or NULL if there is no data to include in the message. For a standard message, the pointer specifies a data block owned by the sending task, which can later be read by the receiving task. For a buffered message, the pointer specifies a block of data to be copied into the message's internal data buffer.

datasize
: The size of the message data, in bytes, or 0 if there is no data to include in the message.

Return Value

The procedure returns 0 if the message was sent successfully or an error code if an error occurs.

Implementation

SWI implemented in kernel folio V20.

Associated Files

msgport.h
: ANSI C Prototype

See Also

GetMsg(), GetThisMsg(), ReplyMsg(), ReplySmallMsg(), SendSmallMsg(), WaitPort()

GetMsg

Gets a message from a message port.

Synopsis

```
Item GetMsg( Item mp )
```

Description

This procedure gets the first message in the message queue for the specified message port and removes the message from the queue.

Arguments

mp
: The item number of the message port from which to get the message.

Return Value

The procedure returns the item number of the first message in the message queue or an error code (a negative value) if an error occurs. Possible error codes include the following:

0
: The queue is empty.

BADITEM
: The mp argument does not specify a message port.

NOTOWNER
: The message port specified by the mp argument is not owned by this task.

Implementation

SWI implemented in kernel folio V20.

Associated Files

msgport.h

ANSI C Prototype

See Also

DeleteMsg(), GetThisMsg(), ReplyMsg(), ReplySmallMsg(), SendMsg(), SendSmallMsg(), WaitPort()

GetThisMsg

Gets a specific message.

Synopsis

```
Item GetThisMsg( Item message )
```

Description

This procedure gets a specific message and removes it from the message queue that contains it, if any.

A task that is receiving messages can use `GetThisMsg()` to get an incoming message. Unlike `GetMsg()`, which gets the first message in a specific message queue, `GetThisMsg()` can get any message from any of the task's message queues.

A task that has sent a message can use `GetThisMsg()` to get it back. If the receiving task hasn't already taken the message from its message queue, `GetThisMsg()` removes it from the queue. If the message is not on any MsgPort a zero is returned.

Arguments

message
: The item number of the message to get.

Return Value

The procedure returns the item number of the message or an error code (a negative value) if an error occurred. Possible error codes include:

BADITEM
: The message argument does not specify a message.

BADPRIV
: The calling task is not allowed to get this message.

Implementation

SWI implemented in kernel folio V20.

Associated Files

msgport.h
ANSI C Prototype

Caveats

Prior to V21, if the message was not on any message port this procedure still returned the item number of the message. In V21 and beyond, if the message is not on any port, the procedure returns an error.

See Also

CreateMsg(), GetMsg(), ReplyMsg(), ReplySmallMsg(), SendMsg(), SendSmallMsg(), WaitPort()

ReplyMsg

Sends a reply to a message.

Synopsis

```
Err ReplyMsg( Item msg, int32 result, const void *dataptr,int32
datasize )
```

Description

This procedure sends a reply to a message.

When replying to a message, send back the same message item you received. (Think of this as sending a reply in the same envelope as the original letter.) Note that you must include a result code (which you provide in the result argument).

The meanings of the dataptr and datasize arguments depend on the type of the original message. (You can find out what type of message it is by looking at its msg.n_Flags field. If it is a small message, the value of the field is MESSAGE_SMALL. If it is a buffered message, the value of the field is MESSAGE_PASS_BY_VALUE.)

- If the original message was a small message, the dataptr and datasize are put in the corresponding fields of the message as eight bytes of data.
- If the original message was a standard message, these refer to a data block that your task allocates for returning reply data. For standard messages, the sending task and the replying task must each allocate their own memory blocks for message data. If the message was also buffered, the data is copied into the internal buffer of the message.

Arguments

msg
: The item number of the message. Note: The reply message item must be same message item that was received.

result
: A result code. This code is placed in the msg_Result field of the message data structure before the reply is sent. Note: There are no standard result codes currently defined; the code must be a value whose meaning is agreed upon by the calling and receiving tasks. In general, you should use negative values to specify errors and 0 to specify that no error occurred.

dataptr
: See the "Description" section above for a description of this argument. If the message is not buffered, set this to 0.

datasize
: See the "Description" section above for a description of this argument. If the message is not buffered, set this to 0.

Return Value

The procedure returns 0 if the reply was sent successfully or an error code if an error occurs.

Implementation

SWI implemented in kernel folio V20.

Associated Files

msgport.h
: ANSI C Prototype

Notes

Most programs handle only one type of message per message port. Each type of message moves information in a different way; programs choose a particular message type to meet their specific needs.

See Also

`GetMsg()`, `GetThisMsg()`, `ReplySmallMsg()`, `SendMsg()`, `SendSmallMsg()`, `WaitPort()`

ReplySmallMsg

Sends a reply to a small message.

Synopsis

```
Err ReplySmallMsg( Item msg, int32 result, uint32 val1, uint32val2)
```

Description

This procedure sends a reply to a small message. (To reply to a standard or buffered message, use `ReplyMsg()`.)

This procedure is identical to `ReplyMsg()` except that arguments are cast as (uint32) instead of (void *) and (int32). A small message can return only eight bytes of data: four that you provide in the val1 argument, and four that you provide in the val2 argument.

When replying to a message, send back the same message item you received. (Think of this as sending a reply in the same envelope as the original letter.) You can include an optional result code (which you provide in the result argument). Note that no standard result codes are currently defined; the result code must be a value whose meaning is agreed upon by the sending and receiving tasks. For consistency, use negative numbers for errors.

Arguments

msg
: The item number of the message. Note: The reply message item must be same message item that was received.

result
: A result code. This code is placed in the msg_Result field of the message data structure before the reply is sent.

val1
: The first four bytes of data for the reply. This data is put into the msg_DataPtr field of the message structure.

val2
: The last four bytes of data for the reply. This data is put into the msg_DataSize field of the message structure.

Return Value

The procedure returns 0 if the reply was sent successfully or an error code if there was an error.

Implementation

SWI implemented in kernel folio V20.

Associated Files

msgport.h
ANSI C Prototype

See Also

GetMsg(), GetThisMsg(), ReplyMsg(), SendMsg(), SendSmallMsg(), WaitPort()

SendSmallMsg

Sends a small message.

Synopsis

```
Err SendSmallMsg( Item mp, Item msg, uint32 val1, uint32 val2 )
```

Description

This procedure sends a small message (a message that contains no more than eight bytes of data) to the specified message port. (To send a standard or buffered message, use `SendMsg()`.) This routine is identical to `SendMsg()`. It is provided only to avoid having to cast uint32 to the types needed by `SendMsg()`.

Arguments

mp
: The item number of the message port to which to send the message

msg
: The item number of the message to send.

val1
: The first four bytes of message data. This data is put into the msg_DataPtr field of the message structure.

val2
: The last four bytes of message data. This data is put into the msg_DataSize field of the message structure.

Return Value

The procedure returns 0 if the message is sent successfully or an error code if an error occurs.

Implementation

SWI implemented in kernel folio V20.

Associated Files

msgport.h
    ANSI C Prototype

See Also

GetMsg(), GetThisMsg(), ReplyMsg(), ReplySmallMsg(), SendMsg(), WaitPort()

WaitPort

Waits for a message to arrive.

Synopsis

```
Item WaitPort( Item mp, Item msg )
```

Description

The procedure puts the calling task into wait state until a message is received at the specified message port. When a task is in wait state, it uses no CPU time.

The task can wait for a specific message by providing the item number of the message in the message argument. To wait for any incoming message, the task uses 0 as the value of the message argument.

Note: If the desired message is already in the message queue for the specified message port, the procedure returns immediately. If the message never arrives, the procedure may never return.

When the message arrives, the task is moved from the wait queue to the ready queue, the item number of the message is returned as the result, and the message is removed from the message port.

Arguments

mp
: The item number of the message port to check for incoming messages.

msg
: The item number of the message to wait for. If this argument is 0, control is returned to the task when any message arrives at the specified message port.

Return Value

The procedure returns the item number of the incoming message or an error code if an error occurs.

Implementation

Folio call implemented in kernel folio V20. Became an SWI in kernel folio V24.

Associated Files

msgport.h
: ANSI C Prototype

clib.lib
: ARM Link Library

See Also

GetMsg(), ReplyMsg(), SendMsg()

CreateBufferedMsg

Creates a buffered message.

Synopsis

```
Item CreateBufferedMsg( const char *name, uint8 pri, Item mp,uint32
datasize )
```

Description

One of the ways tasks communicate is by sending messages to each other. This procedure creates an item for a buffered message (a message that includes an internal buffer for sending data to the receiving task). You can use this procedure in place of `CreateItem()` to create the message.

Arguments

name
: The name of the message (see "Notes").

pri
: The priority of the message. This determines the position of the message in the receiving task's message queue and thus, how soon it is likely to be handled. A larger number specifies a higher priority.

mp
: The item number of the message port at which to receive the reply.

datasize
: The maximum size of the message's internal buffer, in bytes.

Return Value

The procedure returns the item number of the message or an error code if an error occurs.

Implementation

Convenience call implemented in clib.lib V20.

Associated Files

msgport.h
: ANSI C Prototype

clib.lib
: ARM Link Library

Notes

The advantage of using a buffered message instead of a standard message is that the sending task doesn't need to keep the data block containing the message data after the message is sent. All the necessary data is included in the message.

The same message item can be resent any number of times. When you are finished with a message item, use `DeleteMsg()` to delete it.

You can use `FindNamedItem()` to find a message item by name. When creating messages, you should assign unique names whenever possible.

Caveats

The priority value (set with the pri argument) is not used consistently.

See Also

`CreateMsg()`, `CreateMsgPort()`, `CreateSmallMsg()`, `DeleteMsg()`, `DeleteMsgPort()`, `SendMsg()`

CreateSmallMsg

Creates a small message.

Synopsis

```
Item CreateSmallMsg( const char *name, uint8 pri, Item mp )
```

Description

This convenience procedure creates the item for a small message (a message that can contain up to eight bytes of data). You can use this procedure in place of `CreateItem()` to create the item.

To create a standard message (a message in which any data to be communicated to the receiving task is contained in a data block allocated by the sending task), use `CreateMsg()`. To create a buffered message (a message that includes an internal buffer for sending data to the receiving task), use `CreateBufferedMsg()`.

Arguments

name
: The name of the message (see "Notes").

pri
: The priority of the message. This determines the position of the message in the receiving task's message queue and thus, how soon it is likely to be handled. A larger number specifies a higher priority.

mp
: The item number of the message port for the return message.

Return Value

The procedure returns the item number of the message that was created or an error code if an error occurs.

Implementation

Convenience call implemented in clib.lib V20.

Associated Files

msgport.h
: ANSI C Prototype

clib.lib
: ARM Link Library

Notes

The same message item can be resent any number of times. When you are finished with a message item, use `DeleteMsg()` to delete it.

You can use `FindNamedItem()` to find a message by name. When naming messages, you should assign unique names whenever possible.

See Also

`CreateMsg()`, `CreateMsgPort()`, `CreateBufferedMsg()`, `DeleteMsg()`, `DeleteMsgPort()`, `SendMsg()`

CreateSemaphore

Creates a semaphore.

Synopsis

```
Item CreateSemaphore( const char *name, uint8 pri )
```

Description

This convenience procedure creates a semaphore item with the specified name and priority. You can use this procedure in place of `CreateItem()` to create the semaphore.

For information about semaphores, which are used to control access to shared resources, see the Sharing System Resources chapter in the 3DO Portfolio Programmer's Guide.

Arguments

name
: The name of the semaphore (see "Notes").

pri
: The priority of the semaphore; use 0 for now.

Return Value

The procedure returns the item number of the semaphore or an error code if an error occurs.

Implementation

Convenience call implemented in clib.lib V20.

Associated Files

semaphore.h
: ANSI C Prototype

clib.lib

ARM Link Library

Notes

When you no longer need a semaphore, use `DeleteSemaphore()` to delete it.

You can use `FindSemaphore()` to find a semaphore by name. When creating semaphores, you should assign unique names whenever possible.

See Also

`DeleteSemaphore()`, `LockSemaphore()`, `UnlockSemaphore()`

DeleteSemaphore

Deletes a semaphore.

Synopsis

```
Err DeleteSemaphore( Item s )
```

Description

This macro deletes a semaphore.

For information about semaphores, which are used to control access to shared resources, see the Sharing System Resources chapter in the 3DO Portfolio Programmer's Guide.

Arguments

s
: The item number of the semaphore to be deleted.

Return Value

The macro returns a value greater than or equal to 0 if successful or an error code if an error occurred.

Implementation

Macro implemented in semaphore.h V20.

Associated Files

semaphore.h
: ANSI C Macro

Notes

As with all items, the item number of a deleted semaphore is not reused. If a task specifies the item number of a deleted semaphore, the kernel informs the task that the item no longer exists.

See Also

`CreateSemaphore()`

LockSemaphore

Locks a semaphore.

Synopsis

```
int32 LockSemaphore( Item s, uint32 flags )
```

Description

This procedure locks a semaphore. For information about semaphores, which are used to control access to shared resources, see the "Sharing System Resources" chapter in the 3DO System Programmer's Guide.

Arguments

s
: The number of the semaphore to be locked.

flags
: Semaphore flags.

Only one flag is currently defined:

SEM_WAIT
: If the item to be locked is already locked, put the calling task into wait state until the item is available.

Return Value

The procedure returns 1 if the semaphore was successfully locked. If the semaphore is already locked and the SEM_WAIT flag is not set (which indicates that the calling task does not want to wait until the semaphore is available), the procedure returns 0. If an error occurs, the procedure returns an error code.

Implementation

SWI implemented in kernel folio V20.

This is the same SWI as `LockItem()`.

Associated Files

semaphore.h
    ANSI C Prototype

Caveats

There are currently no asynchronous locks or shared read locks.

See Also

`FindSemaphore()`, `UnlockSemaphore()`

FindSemaphore

Finds a semaphore by name.

Synopsis

```
Item FindSemaphore( const char *name )
```

Description

This macro finds a semaphore with the specified name. The search is not case-sensitive (that is, the kernel does not distinguish uppercase and lowercase letters in semaphore names). You can use this macro in place of `FindNamedItem()` to find a semaphore.

For information about semaphores, which are used to control access to shared resources, see the "Sharing System Resources" chapter in the 3DO Portfolio Programmer's Guide.

Arguments

name
: The name of the semaphore to find.

Return Value

The macro returns the item number of the semaphore that was found or an error code if an error occurs.

Implementation

Macro implemented in semaphore.h V20.

Associated Files

semaphore.h
: ANSI C Macro

See Also

`CreateSemaphore()`

UnlockSemaphore

Unlocks a semaphore.

Synopsis

```
Err UnlockSemaphore( Item s )
```

Description

This is an alternate entry point for `UnlockItem()`. This procedure unlocks the specified semaphore. For information about semaphores, which are used to control access to shared resources, see the "Sharing System Resource's" chapter in the 3DO Portfolio Programmer's Guide.

Arguments

s
: The item number of the semaphore to unlock.

Return Value

The procedure returns 0 if successful or an error code (a negative value) if an error occurs. Possible error codes include:

NOTOWNER
: The semaphore specified by the s argument is not owned by the task.

Implementation

SWI implemented in kernel folio V20.

This is the same SWI as `UnlockItem()`.

Associated Files

semaphore.h
: ANSI C Prototype

See Also

LockSemaphore(), LockItem(), UnlockItem()

LockItem

Locks an item.

Synopsis

```
int32 LockItem( Item s, uint32 flags )
```

Description

This procedure locks an item.

Note: Currently, semaphores are the only items that can be locked. You can lock semaphores by calling `LockSemaphore()`.

Arguments

s
: The number of the item to be locked.

flags
: Semaphore flags.

Only one flag is currently defined:

SEM_WAIT
: If the item to be locked is already locked, put the calling task into wait state until the item is available.

Return Value

The procedure returns 1 if the item was successfully locked. If the item is already locked and the SEM_WAIT flag is not set (which indicates that the calling task does not want to wait until the item is available), the procedure returns 0. If an error occurs, the procedure returns an error code.

Implementation

SWI implemented in kernel folio V20.

This is the same SWI as `LockSemaphore()`

Associated Files

item.h
: ANSI C Prototype

See Also

`DeleteItem()`, `UnlockItem()`

DeleteItem

Deletes an item.

Synopsis

```
Err DeleteItem( Item i )
```

Description

This procedure deletes the specified item and frees any resources (including memory) that were allocated for the item.

Note: There are convenience procedures for deleting most types of items (such as `DeleteMsg()` to delete a message and `DeleteIOReq()` to delete an I/O request). You should use `DeleteItem()` only if you used `CreateItem()` to create the item. If you used a convenience routine to create an item, you must use the corresponding convenience routine to delete the item.

Arguments

i
: Number of the item to be deleted.

Return Value

The procedure returns a value greater than of equal to 0 if successful or an error code if an error occurs.

Implementation

SWI implemented in kernel folio V20.

Associated Files

item.h
: ANSI C Prototype

Notes

The item number of a deleted item is not reused. If a task specifies the item number of a deleted item, the kernel informs the task that the item no longer exists.

Tasks can only delete items that they own. If a task transfers ownership of an item to another task, it can no longer delete the item. The lone exception to this is the task itself. You can always commit sepaku.

When a task dies, the kernel automatically deletes all of the items in its resource table.

See Also

CheckItem(), CreateItem(), DeleteIOReq(), DeleteMsg(), DeleteMsgPort(), DeleteThread(), exit()

DeleteThread

Deletes a thread.

Synopsis

```
Err DeleteThread( Item x )
```

Description

This procedure deletes a thread.

Arguments

x
: The item number of the thread to be deleted.

Return Value

The procedure returns a value greater than or equal to 0 if the thread was successfully deleted or an error code if an error occurs.

Implementation

Convenience call implemented in clib.lib V20.

Associated Files

task.h
: ANSI C Prototype

clib.lib
: ARM Link Library

Notes

Although threads may be deleted as soon as the parent task has finished using it by calling `DeleteThread()`, threads are automatically deleted when they return, call `exit(),` or when their

parent task is deleted or dies.

Threads share memory with their parent tasks. This means that:

- Memory that was allocated by a thread also belongs to the parent task.
- Memory that was allocated by a thread is not automatically deallocated when the thread is deleted but instead remains the property of the parent task.

If you create a thread using `CreateThread()`, you must delete it using `DeleteThread()`, which does special cleanup work.

See Also

`CreateThread()`, `exit()`

CreateThread

Creates a thread.

Synopsis

```
Item CreateThread( const char *name, uint8 pri, void (*code) (),int32
stacksize )
```

Description

This procedure creates a thread. The resulting thread belongs to the calling task.

Arguments

name
: The name of the thread to be created (see "Notes").

pri
: The priority of the thread. This can be a value from 11 to 199 (0 to 10 and 200 to 255 can only be assigned by system software). A larger number specifies a higher priority. For all tasks, including threads, the highest priority task - if there is only one task with that priority - is always the task that is executed. If more than one task in the ready queue has the highest priority, the kernel divides the available CPU time among the tasks by providing each task with a time quantum.

code
: A pointer to the code that the thread executes.

stacksize
: The size of the thread's stack, in bytes (see "Notes").

Return Value

The procedure returns the item number of the thread or an error code if an error occurs.

Implementation

Convenience call implemented in clib.lib V20.

Associated Files

task.h
: ANSI C Prototype

clib.lib
: ARM Link Library

Notes

There is no default size for a thread's stack. To avoid stack overflow errors, the stack must be large enough to handle any possible uses. One way to find the proper stack size for a thread is to start with a very large stack, reduce its size until a stack overflow occurs, and then double its size.

When you no longer need a thread, use `DeleteThread()` to delete it. Alternatively, the thread can return or call `exit()`.

You can use `FindTask()` to find a thread by name. When naming threads, you should assign unique names whenever possible.

See Also

`DeleteThread()`, `exit()`

---

exit

Exits from a task or thread.

Synopsis

```
void exit( int status )
```

Description

This procedure deletes the calling task or thread. If the CREATETASK_TAG_MSGFROMCHILD tag was set when creating the calling task, then the status is sent to the parent process through a message.

Arguments

status
: The status to be returned to the parent of the calling task or thread. Negative status is reserved for system use.

Return Value

This procedure never returns.

Implementation

Folio call implemented in kernel folio V20.

Associated Files

stdlib.h
: ANSI C Prototype

Notes

When tasks (including threads) have finished their work, they should call `exit()` to die gracefully. The system will call `exit()` on behalf of the task that returns from the top-level routine. `exit()` does necessary clean-up work when a task is finished, including the cleanup required for a thread created by the `CreateThread()` library routine.

exit

See Also

CreateThread(), DeleteItem(), DeleteThread()

UnlockItem

Unlocks a locked item.

Synopsis

```
Err UnlockItem( Item s )
```

Description

This procedure unlocks the specified item.

Note: Currently, semaphores are the only items that can be locked. You can also unlock semaphores by calling `UnlockSemaphore()`.

Arguments

s
: The item number of the item to be unlocked.

Return Value

The procedure returns 0 if successful or an error code (a negative value) if an error occurs. Possible error codes include:

NOTOWNER
: The item specified by the s argument is not owned by the task.

Implementation

SWI implemented in kernel folio V20.

This is the same SWI as `UnlockSemaphore()`.

Associated Files

item.h
: ANSI C Prototype

See Also

DeleteItem(), LockItem(), UnlockSemaphore()

FindItem

Finds an item by type and tags.

Synopsis

```
Item FindItem( int32 cType, TagArg *tp )
Item FindItemVA( int32 cType, uint32 tags, ...)
```

Description

This procedure finds an item of the specified type whose tag arguments match those pointed to by the tp argument. If more than one item of the specified type has matching tag arguments, the procedure returns the item number for the first matching item.

Arguments

cType
: Specifies the type of the item to find. Use MkNodeID() to create this value.

tp
: A pointer to an array of tag arguments. The tag arguments can be in any order. The array can contain some, all, or none of the possible tag arguments for an item of the specified type; to make the search more specific, include more tag arguments. The last element of the array must be the value TAG_END. If there are no tag arguments, this argument must be NULL. For a list of tag arguments for each item type, see the Portfolio Item's chapter.

Return Value

The procedure returns the number of the first item that matches or an error code if it can't find the item or if an error occurs.

Implementation

SWI implemented in kernel folio V20.

Associated Files

item.h

ANSI C Prototype

Caveats

The procedure currently ignores most tag arguments.

See Also

`CheckItem()`, `FindNamedItem()`, `FindVersionedItem()`, `LookupItem()`, `MkNodeID()`

FindNamedItem

Finds an item by name.

Synopsis

```
Item FindNamedItem( int32 ctype, const char *name )
```

Description

This procedure finds an item of the specified type whose name matches the name argument. The search is not case-sensitive (that is, the kernel does not distinguish uppercase and lowercase letters in item names).

Arguments

ctype
: The type of the item to find. Use MkNodeID() to create this value.

name
: The name of the item to find.

Return Value

The procedure returns the number of the item that was found. It returns an error code if an error occurs.

Implementation

Convenience call implemented in clib.lib V20.

Associated Files

item.h
: ANSI C Prototype

clib.lib
: ARM Link Library

See Also

FindItem(), FindVersionedItem()

FindVersionedItem

Finds an item by name and version.

Synopsis

```
Item FindVersionedItem( int32 ctype, const char *name, uint8 vers,
uint8 rev )
```

Description

This procedure finds an item of a specified type by its name, version number, and revision number. These values are required tag arguments for all items. If all three values match, the procedure returns the item number of the matching item.

Arguments

ctype
: The type of the item to find. Use MkNodeID() to create this value.

name
: The name of the item to find.

vers
: The version number of the item to find.

rev
: The revision number of the item to find.

Return Value

The procedure returns the number of the item that matches or an error code if an error occurs.

Implementation

Convenience call implemented in clib.lib V20.

Associated Files

item.h
: ANSI C Prototype

clib.lib
: ARM Link Library

Caveats

This currently returns a match if the version number and revision number are greater than or equal to the values specified.

See Also

`CheckItem()`, `FindItem()`, `FindNamedItem()`, `LookupItem()`

LookupItem

Gets a pointer to an item.

Synopsis

```
void *LookupItem( Item i )
```

Description

This procedure finds an item by its item number and returns the pointer to the item.

Note: Because items are owned by the system, user tasks cannot change the values of their fields. They can, however, read the values contained in the public fields.

Arguments

i
: The number of the item to look up.

Return Value

The procedure returns the pointer to the item or NULL if the item does not exist.

Implementation

Folio call implemented in kernel folio V20.

Associated Files

item.h
: ANSI C Prototype

clib.lib
: ARM Link Library

See Also

CheckItem(), CreateItem(), FindItem(), FindNamedItem(), FindVersionedItem()

MkNodeID

Creates an item type value

Synopsis

```
int32 MkNodeID( uint8 a , uint8 b )
```

Description

This macro creates an item type value, a 32-bit value that specifies an item type and the folio in which the item type is defined. This value is required by other procedures that deal with items, such as `CreateItem()` and `FindItem()`.

Arguments

a
: The item number of the folio in which the item type of the item is defined. For a list of folio item numbers, see the "Portfolio Items" chapter.

b
: The item type number for the item. For a list of item type numbers, see the "Portfolio Items" chapter.

Return Value

The macro returns an item type value or an error code (a negative value) if an error occurs.

Implementation

Macro implemented in nodes.h V20.

Associated Files

nodes.h
: ANSI C Macro definition

See Also

CreateItem(), FindItem(), FindNamedItem(), FindVersionedItem()

ClearCurrentSignals

Clears some received signal bits.

Synopsis

```
Err ClearCurrentSignals( int32 sigMask )
```

Description

This macro resets the requested signal bits of the current task to 0.

Arguments

sigMask
: A 32-bit word indicating which signal bits should be cleared.

Return Value

Returns > = 0 for success or a negative error code for failure.

Implementation

Macro implemented in kernel.h V24.

Associated Files

kernel.h
: ANSI C Macro definition

See Also

AllocSignal(), FreeSignal(), SendSignal(), WaitSignal()

CloseItem

Closes a system item.

Synopsis

```
Err CloseItem( Item i )
```

Description

System items are items that are created automatically by the operating system, such as folios and device drivers, and thus do not need to be created by tasks. To use a system item, a task first opens it by calling `OpenItem()`. When a task is done using a system item, it calls `CloseItem()` to inform the operating system that it is no longer using the item.

Arguments

i
: Number of the item to close.

Return Value

The procedure returns 0 if the item was closed successfully or an error code (a negative value) if an error occurs. Possible error codes include:

BADITEM
: The i argument is not an item.

ER_Kr_ItemNotOpen
: The i argument is not an opened item.

Implementation

SWI implemented in kernel folio V20.

Associated Files

item.h

ANSI C Prototype

See Also

`OpenItem()`

OpenItem

Opens an item.

Synopsis

```
Item OpenItem( Item FoundItem, void *args )
```

Description

This procedure opens an item for access.

Arguments

item
: The number of the item to open. To find the item number of a system item, use FindItem(), FindDevice(), FindFolio(), or FindNamedItem().

args
: A pointer to an array of tag arguments. Currently, there are no system items that require tag arguments; for these and for any future items you're not sending tag arguments to, this argument must be NULL. If you send tag arguments (for any items in the future that use them), the tag arguments can be in any order, and the last element of the array must be the value TAG_END. For a list of tag arguments for each item type, see the "Portfolio Items" chapter.

Return Value

The procedure returns the number of the item that was opened or an error code if an error occurs.

Note: The item number for the opened item is not always the same as the item number returned by `FindItem()`. When accessing the opened item you should use the return from `OpenItem()`, not the return from `FindItem()`. You can also call the `FindAndOpenItem()` function for to do both a search and an open operation in an atomic manner.

Implementation

SWI implemented in kernel folio V20.

Associated Files

item.h
ANSI C Prototype

Notes

To close an opened item, use `CloseItem()`.

See Also

`CloseItem()`, `FindItem()`

CloseNamedDevice

Closes a device previously opened with OpenNamedDevice().

Synopsis

```
Err CloseNamedDevice(Item devItem);
```

Description

This macro closes a device that was previously opened via a call to `OpenNamedDevice()`.

Arguments

devItem
: The device's item, as returned by OpenNamedDevice().

Return Value

The procedure returns 0 if the item was closed successfully or an error code (a negative value) if an error occurs. Possible error codes include:

BADITEM
: The devItem argument is not an item.

ER_Kr_ItemNotOpen
: The devItem argument is not an opened item.

Implementation

Macro implemented in device.h V22.

Associated Files

device.h
: ANSI C Macro

See Also

OpenItem(), CloseItem(), FindItem()

CountBits

Counts the number of bits set in a word.

Synopsis

```
int CountBits(uint32 mask);
```

Description

CountBits examines the longword passed in as a parameter and returns the number of bits that are turned on in the word, using an algorithm derived from a note in MIT's HAKMEM.

Arguments

mask
: The data word to count the bits of.

Return Value

The number of bits that were turned on in the supplied value.

Implementation

Convenience call implemented in clib.lib V21.

Associated Files

string.h
: ANSI C Prototype

clib.lib
: ARM Link library

See Also

FindMSB(), FindLSB()

CountBits

FindMSB

Finds the highest-numbered bit.

Synopsis

```
int FindMSB( uint32 bits )
```

Description

This procedure finds the highest-numbered bit that is set in the argument. The least-significant bit is bit number 1, and the most-significant bit is bit 32.

Arguments

bits
: The 32-bit word to check.

Return Value

The procedure returns the number of the highest-numbered bit that is set in the argument. If no bits are set, the procedure returns 0.

Implementation

Convenience call implemented in clib.lib V20.

Associated Files

clib.lib
: ARM Link Library

See Also

FindLSB(), ffs()

FindLSB

Finds the least-significant bit.

Synopsis

```
int FindLSB( uint32 mask )
```

Description

FindLSB() finds the lowest-numbered bit that is set in the argument. The least-significant bit is bit number 1, and the most-significant bit is bit 32.

Arguments

flags
: The 32-bit word to check.

Return Value

FindLSB() returns the number of the lowest-numbered bit that is set in the argument (for example, 1 if the value of the argument is 1 or 3 or 7, 3 if the value of the argument is 4 or 12). If no bits are set, the procedure returns 0.

Implementation

Convenience call implemented in clib.lib V20.

Associated Files

clib.lib
: ARM Link Library

See Also

FindMSB(), ffs()

---

ffs

Finds the first set bit.

Synopsis

```
int ffs( uint32 mask )
```

Description

ffs finds the lowest-numbered bit that is set in the argument. The least-significant bit is bit number 1, and the most-significant bit is bit 32.

Arguments

flags
: The 32-bit word to check

Return Value

ffs returns zero if no bits are set in the parameter, or the bit index of the first bit is set in the parameter. For example, ffs returns 1 if the value of the argument is 1 or 3 or 7; and 3 if the value of the argument is 4 or 12. If no bits are set, ffs returns 0.

Implementation

Convenience call implemented in clib.lib V20.

Associated Files

clib.lib
: ARM Link Library

Notes

This entry point is provided for portability and compatibility only; code written specifically for Portfolio should use FindLSB() or FindMSB() to reduce any ambiguity over whether the least-significant or most-significant bit is desired.

ffs

See Also

FindMSB(), FindLSB()

CreateMemDebug

Initializes memory debugging package.

Synopsis

```
Err CreateMemDebug(uint32 controlFlags, const TagArg *args);
```

Description

This function creates the needed data structures and initializes them as needed for memory debugging. This function actually only does anything if MEMDEBUG is defined, otherwise it is a NOP.

Memory debugging provides a general-purpose mechanism to track and validate all memory allocations done by an application. Using memory debugging, you can easily determine where memory leaks occur within a title, and find illegal uses of the memory subsystem.

To enable memory debugging in a title, do the following:

- Add a call to `CreateMemDebug()` as the first instruction in the `main()` routine of your program.
- Add calls to `DumpMemDebug()` and `DeleteMemDebug()` as the last instructions in the `main()` routine of your program.
- Recompile your entire project with MEMDEBUG defined on the compiler's command-line (for the ARM compiler, this is done by adding _DMEMDEBUG).
- Link your code with memdebug.lib

With these steps taken, all memory allocations done by your program will be tracked, and specially managed. On exiting your program, any memory left allocated will be displayed to the console, along with the line number and source file where the memory was allocated from.

In addition, the memory debugging code makes sure that illegal or dangerous uses of memory are detected and flagged. Most messages generated by the debugging code indicate the offending source file and line within your source code where the problem originated.

When all options are turned on, the debugging code will check and report the following problems:

- memory allocations with a size of 0
- memory free with a bogus memory pointer
- memory free with a size not matching the size used when the memory was allocated

- NULL memory list passed to `AllocMemFromMemLists()` or `FreeMemToMemLists()`
- cookies on either side of all memory allocations are checked to make sure they are not altered from the time a memory allocation is made to the time the memory is released. This would indicate that something is writing beyond the bounds of allocated memory.

When MEMDEBUG is defined during compilation, all standard memory allocation calls are automatically vectored through the debugging code. This includes even memory allocation calls made inside of previously compiled.lib files you might be linking with. However, you can get better debugging information if you recompile everything in your project, including .lib files, with MEMDEBUG defined.

By calling the `DumpMemDebug()` function at any time within your program, you can get a detailed listing of all memory currently allocated, showing from which source line and source file the allocation occurred.

Link order can be important when using memdebug.lib. The library should come before clib.lib, but after everything else. That is, the link order should be:

[many .lib files] memdebug.lib clib.lib

Arguments

controlFlags
: A set of bit flags controlling various options of the memory debugging code.

MEMDEBUGF_ALLOC_PATTERNS
: When this flag is specified, it instructs the debugging code to fill newly allocated memory with the constant MEMDEBUG_ALLOC_PATTERN. Doing so will likely cause your program to fail in some way if it tries to read newly allocated memory without first initializing it. Note that this option has no effect if memory is allocated using the MEMTYPE_FILL memory flag.

MEMDEBUGF_FREE_PATTERNS
: When this flag is specified, it instructs the debugging code to fill memory that is being freed with the constant MEMDEBUG_FREE_PATTERN. Doing so will likely cause your program to fail in some way if it tries to read memory that has been freed.

MEMDEBUG_PAD_COOKIES
: When this flag is specified, it causes the debugging code to put special memory cookies in the bytes before and 16 bytes after every block of memory it allocates. When a memory block is freed, the cookies are checked to make sure that they have not been altered. This option makes sure that your program is not writing outside the bounds of memory it allocates.

MEMDEBUG_DEBUG_ON_ERRORS

When this flag is specified, the debugging code will automatically invoke the debugger if any error is detected. Errors includes such things as mangled pad cookies, incorrect size for a FreeMem() call, etc. Normally, the debugging code simply prints out the error to the console and keeps executing.

MEMDEBUG_ALLOW_OWNER_SWITCH

This flag tells the memory debugging code that it is OK for memory to be allocated by one thread and freed by a different thread. Normally, this condition would be flagged as an error.

MEMDEBUGF_CHECK_ALLOC_FAILURES

When this flag is specified, the debugging code will emit a message whenever a memory allocation call fails due to lack of memory. This can be useful to track down where in a title memory is running out.

MEMDEBUGF_KEEP_TASK_DATA

The debugging code maintains some thread-specific statistics about memory allocations performed by that thread. This information gets displayed by DumpMemDebug(). Whenever all of the memory allocated by a thread is freed, the data structure holding the statistics for that thread automatically gets freed by the debugging code. This is undesirable if you wish to dump out statistics of the code just before a title exits. Specifying this flag causes the data structure not to be freed, making the statistical information available to DumpMemDebug().

MEMDEBUGF_USE_VRAM

The debugging code needs to allocate private memory to track information about every memory allocation call made. This flag tells the debugging code to allocate this private memory using MEMTYPE_VRAM instead of MEMTYPE_ANY. This is useful if you need as much DRAM as possible, and do not want the debugging code to take any away from you.

args

A pointer to an array of tag arguments containing extra data for this function. This must currently always be NULL.

Return Value

Returns > = 0 if successful or a negative error code. Current possible error codes are:

MEMDEBUG_ERR_BADTAG

An undefined tag was supplied. Currently the arg's parameter to this function should always be NULL.

MEMDEBUG_ERR_BAD

FLAGS
: An undefined flag bit was set in the control Flags parameter to this function.

Implementation

Library call implemented in memdebug.lib V24.

Associated Files

mem.h, memdebug.lib

Notes

You should make sure to turn off memory debugging prior to creating the final version of your program. Enabling memory debugging incurs an overhead of currently 32 bytes per allocation made. If you use the MEMDEBUGF_PAD_COOKIES option, this overhead grows to 64 bytes per allocation.

In addition, specifying the MEMDEBUGF_ALLOC_PATTERNS and MEMDEBUGF_FREE_PATTERNS options will slow down memory allocations and free operations, due to the extra work of filling the memory with the patterns.

When reporting errors to the console, the memory debugging subsystem will normally print the source file and line number where the error occurred. When using link libraries which have not been recompiled with MEMDEBUG defined, the memory debugging subsystem will still be able to track the allocations, but will not report the source file or line number where the error occurred. It will report < unknown source file > instead.

Caveat

If you link your program with memdebug.lib, you must call `CreateMemDebug()` at the beginning of your program. Any attempt to allocate memory prior to calling `CreateMemDebug()` will fail.

See Also

`DeleteMemDebug()`, `DumpMemDebug()`, `SanityCheckMemDebug()`

DeleteMemDebug

Releases memory debugging resources.

Synopsis

```
Err DeleteMemDebug(void);
```

Description

Deletes any resources allocated by `CreateMemDebug()`. This function only actually does anything if MEMDEBUG is defined, otherwise it is a NOP.

This function also frees any memory allocated that has not been freed yet.

Return Value

Returns > = 0 if successful, or a negative error code if not.

Implementation

Library call implemented in memdebug.lib V24.

Associated Files

mem.h, memdebug.lib

See Also

`CreateMemDebug()`, `DumpMemDebug()`, `SanityCheckMemDebug()`

DumpMemDebug

Dumps memory allocation debugging information.

Synopsis

```
Err DumpMemDebug(const TagArg *args);
```

Description

This function outputs a table showing all memory currently allocated through the memory debugging code. This table shows the allocation size, address, as well as the source file and the source line where the allocation took place.

This function also outputs statistics about general memory allocation patterns. This includes the number of memory allocation calls that have been performed, the maximum number of bytes allocated at any one time, current amount of allocated memory, etc. All this information is displayed on a per-thread basis, as well as globally for all threads.

To use this function, the memory debugging code must have been previously initialized using the `CreateMemDebug()` function.

Arguments

args
: A pointer to an array of tag arguments containing extra data for this function. This must currently always be NULL.

Return Value

Returns > = 0 if successful, or a negative error code if not. Current Possible error code is:

MEMDEBUG_ERR_BADTAG
: An undefined tag was supplied. Currently the arg's parameter to this function should always be NULL.

Implementation

Library routine implemented in memdebug.lib V24.

Associated Files

mem.h, memdebug.lib

See Also

CreateMemDebug(), DeleteMemDebug(), SanityCheckMemDebug()

SanityCheckMemDebug

Checks all current memory allocations to make sure all of the allocation cookies are intact

Synopsis

```
Err SanityCheckMemDebug(const TagArg *args);
```

Description

This function checks all current memory allocations to see if any of the memory cookies have been corrupted. This is useful when trying to track down at which point in a program's execution memory cookies are being trashed.

The routine will also complain about any memory allocated by a task or thread that doesn't exist anymore. This only happens if the MEMDEBUGF_ALLOW_OWNER_SWITCH flag was not specified when calling `CreateMemDebug()`.

Arguments

args
: A pointer to an array of tag arguments containing extra data for this function. This must currently always be NULL.

Return Value

Returns > = 0 if successful or a negative error code if not. Current possible error code is.

MEMDEBUG_ERR_BADTAG
: An undefined tag was supplied. Currently the args parameter to this function should always be NULL.

Implementation

Library routine implemented in memdebug.lib V24.

Associated Files

mem.h, memdebug.lib

See Also

CreateMemDebug(), DeleteMemDebug(), DumpMemDebug()

CreateUniqueMsgPort

Creates a message port with a unique name.

Synopsis

```
Item CreateUniqueMsgPort( const char *name, uint8 pri, uint32 signal )
```

Description

This convenience procedure creates a message port item. It also creates a message queue for the port in system RAM. You can use this procedure in place of `CreateItem()` to create the item.

This function works much like `CreateMsgPort()`, except that it guarantees that no other message port item of the same name already exists. And once this port created, no other port of the same name will be allowed to be created.

Arguments

name
: The name of the message port (see "Notes").

pri
: The priority of the message port. The priority has no effect on the way message ports operate, but it does determine the order of the message ports in the global list of message ports and thus, the order in which message ports are found.

signal
: Specifies the bit in the signal word that is set when a message arrives at this port. If zero is supplied a new signal will be allocated. (See the description of AllocSignal() for information about allocating signal bits.)

Return Value

The procedure returns the item number of the new message port or an error code if an error occurs. If a port of the same name already existed when this call was made, the ER_Kr_UniqueItemExists error will be returned.

Implementation

Convenience call implemented in clib.lib V24.

Associated Files

msgport.h
: ANSI C Prototype

clib.lib
: ARM Link Library

Notes

When you no longer need a message port, use `DeleteMsgPort()` to delete it.

You can use `FindMsgPort()` to find a message port by name.

See Also

`CreateMsg()`, `DeleteMsg()`, `DeleteMsgPort()`, `SendMsg()`, `SendSmallMsg()`

CreateUniqueSemaphore

Creates a semaphore with a unique name.

Synopsis

```
Item CreateUniqueSemaphore( const char *name, uint8 pri )
```

Description

This convenience procedure creates a semaphore item with the specified name and priority. You can use this procedure in place of `CreateItem()` to create the semaphore.

This function works much like `CreateSemaphore()`, except that it guarantees that no other semaphore item of the same name already exists. And once this semaphore created, no other semaphore of the same name is allowed to be created.

For information about semaphores, which are used to control access to shared resources, see the Sharing System Resources chapter in the 3DO Portfolio Programmer's Guide.

Arguments

name
: The name of the semaphore (see "Notes").

pri
: The priority of the semaphore; use 0 for now.

Return Value

The procedure returns the item number of the semaphore or an error code if an error occurs. If a semaphore of the same name already existed when this call was made, the ER_Kr_UniqueItemExists error will be returned.

Implementation

Convenience call implemented in clib.lib V24.

Associated Files

semaphore.h
: ANSI C Prototype

clib.lib
: ARM Link Library

Notes

When you no longer need a semaphore, use `DeleteSemaphore()` to delete it.

You can use `FindSemaphore()` to find a semaphore by name.

See Also

`DeleteSemaphore()`, `LockSemaphore()`, `UnlockSemaphore()`

DumpNode

Prints contents of a node.

Synopsis

```
void DumpNode (const Node *node, const char *banner)
```

Description

This function prints out the contents of a Node structure for debugging purposes.

Arguments

node
: The node to print.

banner
: Descriptive text to print before the node contents. May be NULL.

Implementation

Convenience call implemented in clib.lib V24.

Associated Files

list.h
: ANSI C Prototype

clib.lib
: ARM Link Library

DumpTagList

Prints contents of a tag list.

Synopsis

```
void DumpTagList (const TagArg *tagList, const char *desc)
```

Description

This function prints out the contents of a TagArg list.

Arguments

tagList
: The list of tags to print.

desc
: Description of tag list to print. Can be NULL.

Implementation

Convenience call implemented in clib.lib V24.

Associated Files

tags.h
: ANSI C Prototype

clib.lib
: ARM Link Library

See Also

NextTagArg(), GetTagArg()

NextTagArg

Finds the next TagArg in a tag list.

Synopsis

```
TagArg *NextTagArg( const TagArg **tagList );
```

Description

This function iterates through a tag list, skipping and chaining as dictated by control tags. There are three control tags:

TAG_NOP
: Ignores that single entry and moves to the next one.

TAG_JUMP
: Has a pointer to another array of tags.

TAG_END
: Marks the end of the tag list.

This function only returns TagArgs which are not system tags. Each call returns either the next TagArg you should examine, or NULL when the end of the list has been reached.

Arguments

tagList
: This is a pointer to a storage location used by the iterator to keep track of its current location in the tag list. The variable that this parameter points to should be initialized to point to the first TagArg in the tag list, and should not be changed thereafter.

Return Value

This function returns a pointer to a TagArg structure, or NULL if all the tags have been visited. None of the control tags are ever returned to you, they are handled transparently by this function.

Example

Example 1: *Example for NextTagArg*

```
void WalkTagList(const TagArg *tags)
{
TagArg *state;
TagArg *currentTag;

    state = tags;
        while ((tag = NextTagItem(&state)) != NULL)
    {
            switch (tag->ta_Tag)
            {
                case TAG1: // process this tag
                           break;

                case TAG2: // process this tag
                           break;

                default  : // unknown tag, return an error
                       break;
        }
        }
}
```

Implementation

Folio call implemented in kernel folio V24.

Associated Files

tags.h

See Also

FindTagArg()

FindTagArg

Looks through a tag list for a specific tag.

Synopsis

```
TagArg *FindTagArg( const TagArg *tagList, uint32 tag );
```

Description

This function scans a tag list looking for a TagArg structure with a ta_Tag field equal to the tag parameter. The function always returns the last TagArg structure in the list which matches the tag. Finally, this function deals with the various control tags such as TAG_JUMP and TAG_NOP.

Arguments

tagList
: The list of tags to scan.

tag
: The value to look for.

Return Value

This function returns a pointer to a TagArg structure with a value of ta_Tag that matches the tag parameter, or NULL if no match can be found. The function always returns the last tag in the list which matches.

Implementation

Folio call implemented in kernel folio V24.

Associated Files

tags.h

See Also

NextTagArg()

GetTagArg

Finds a TagArg in list and returns its ta_Arg field.

Synopsis

```
TagData GetTagArg (const TagArg *tagList, uint32 tag,TagData defaultValue)
```

Description

This function calls `FindTagArg()` to locate the specified tag. If it is found, it returns the ta_Arg value from the found TagArg. Otherwise, it returns the default value supplied. This is handy when resolving a tag list that has optional tags that have suitable default values.

Arguments

tagList
: The list of tags to scan. Can be NULL.

tag
: The tag ID to look for.

defaultValue
: Default value to use when specified tag isn't found in tagList.

Return Value

ta_Arg value from found TagArg or defaultValue.

Implementation

Convenience call implemented in clib.lib V24.

Examples

```
void dosomething (const TagArg *tags)
{
    uint32 amplitude = (uint32)GetTagData (tags, MY_TAG_AMPLITUDE, (TagData)0x7fff);
    .
    .
    .
}
```

Caveats

It's a good idea to always use casts for the default value and result. Don't assume anything about the type definition of

TagData other than that it is a 32-bit value.

Associated Files

tags.h
: ANSI C Prototype

clib.lib
: ARM Link Library

See Also

FindTagArg(), NextTagArg()

FindAndOpenDevice

Finds a device by name and opens it.

Synopsis

```
Item FindAndOpenDevice( const char *name )
```

Description

This macro finds a device whose name matches the name argument. The search is not case-sensitive (that is, the kernel does not distinguish uppercase and lowercase letters in device names). If the device is found, it is opened.

Arguments

name
: The name of the device to find.

Return Value

The macro returns the item number of the device or an error code if an error occurs.

Implementation

Macro implemented in device.h V24.

Associated Files

device.h
: ANSI C Macro

See Also

OpenNamedDevice(), FindAndOpenItem()

OpenNamedDevice

Opens a named device.

Synopsis

```
Item OpenNamedDevice( const char *name, a )
```

Description

This macro opens a device that you specify by name.

Arguments

name
: The name of the device to open.

a
: This argument, which is not currently used, must be 0.

Return Value

The procedure returns the item number for the opened device or an error code if an error occurs.

Implementation

Macro implemented in device.h V20.

Associated Files

device.h
: ANSI C Macro definition

See Also

CloseNamedDevice(), OpenItem(), CloseItem(), FindItem()

FindAndOpenItem

Finds an item by type and tags and opens it.

Synopsis

```
Item FindAndOpenItem( int32 cType, TagArg *tp )
Item FindAndOpenItemVA( int32 cType, uint32 tags, ... );
```

Description

This procedure finds an item of the specified type whose tag arguments match those pointed to by the tp argument. If more than one item of the specified type has matching tag arguments, the procedure returns the item number for the first matching item. When an item is found, it is automatically opened and prepared for use.

Arguments

cType
: Specifies the type of the item to find. Use MkNodeID() to create this value.

tp
: A pointer to an array of tag arguments. The tag arguments can be in any order. The array can contain some, all, or none of the possible tag arguments for an item of the specified type; to make the search more specific, include more tag arguments. The last element of the array must be the value TAG_END. If there are no tag arguments, this argument must be NULL. For a list of tag arguments for each item type, see the Portfolio Item's chapter.

Return Value

The procedure returns the number of the first item that matches or an error code if it can't find the item or if an error occurs.The returned item is already opened, so there is no need to call `OpenItem()` on it. You should call `CloseItem()` on the supplied item when you are done with it.

Implementation

SWI implemented in kernel folio V24.

Associated Files

item.h

ANSI C Prototype

Notes

To close an opened item, use `CloseItem()`.

See Also

`CheckItem()`, `FindNamedItem()`, `FindVersionedItem()`, `LookupItem()`, `MkNodeID()`, `OpenItem()`, `FindItem()`

FindAndOpenFolio

Finds a folio by name and opens it.

Synopsis

```
Item FindAndOpenFolio( const char *name )
```

Description

This macro finds a folio whose name matches the name argument. The search is not case-sensitive (that is, the kernel does not distinguish uppercase and lowercase letters in folio names). You can use this macro in place of `FindAndOpenNamedItem()` to find a folio and open it.

Arguments

name
: The name of the folio to find.

Return Value

The macro returns the item number of the folio or an error code if an error occurs.

Implementation

Macro implemented in folio.h V24.

Associated Files

folio.h
: ANSI C Macro

See Also

`CallFolio()`, `CallFolioRet()`, `FindAndOpenNamedItem()`

FindAndOpenNamedItem

Finds an item by name and opens it.

Synopsis

```
Item FindAndOpenNamedItem( int32 ctype, const char *name )
```

Description

This procedure finds an item of the specified type whose name matches the name argument. The search is not case-sensitive (that is, the kernel does not distinguish uppercase and lowercase letters in item names). When an item is found, it is automatically opened and prepared for use.

Arguments

ctype
: The type of the item to find. Use MkNodeID() to create this value.

name
: The name of the item to find.

Return Value

The procedure returns the number of the item that was opened. It returns an error code if an error occurs.

Implementation

Convenience call implemented in clib.lib V24.

Associated Files

item.h
: ANSI C Prototype

clib.lib
: ARM Link Library

See Also

FindAndOpenItem()

FindDevice

Finds a device by name.

Synopsis

```
Item FindDevice( const char *name )
```

Description

This macro finds a device whose name matches the name argument. The search is not case-sensitive (that is, the kernel does not distinguish uppercase and lowercase letters in device names).

Arguments

name
: The name of the device to find.

Return Value

The macro returns the item number of the device or an error code if an error occurs.

Implementation

Macro implemented in device.h V20.

Associated Files

device.h
: ANSI C Macro

See Also

OpenNamedDevice(), FindItem()

FindFolio

Finds a folio by name.

Synopsis

```
Item FindFolio( const char *name )
```

Description

This macro finds a folio whose name matches the name argument. The search is not case-sensitive (that is, the kernel does not distinguish uppercase and lowercase letters in folio names). You can use this macro in place of `FindNamedItem()` to find a folio.

Arguments

name
: The name of the folio to find.

Return Value

The macro returns the item number of the folio or an error code if an error occurs.

Implementation

Macro implemented in folio.h V20.

Associated Files

folio.h
: ANSI C Macro

See Also

`CallFolio()`, `CallFolioRet()`

FindMsgPort

Finds a message port by name.

Synopsis

```
Item FindMsgPort( const char *name )
```

Description

This macro finds a message port with the specified name. The search is not case-sensitive (that is, the kernel does not distinguish uppercase and lowercase letters in message port names). You can use this macro in place of `FindNamedItem()` to find a message port.

Arguments

name
: The name of the message port to find.

Return Value

The macro returns the item number of the message port that was found or an error code if an error occurs.

Implementation

Macro implemented in msgport.h V20.

Associated Files

msgport.h
: ANSI C Macro

See Also

`CreateMsgPort()`

FindNamedNode

Finds a node by name.

Synopsis

```
Node *FindNamedNode( const List *l, const char *name )
```

Description

This procedure searches a list for a node with the specified name. The search is not case-sensitive (that is, the kernel does not distinguish uppercase and lowercase letters in node names).

Arguments

l
: A pointer to the list to search.

name
: The name of the node to find.

Return Value

The procedure returns a pointer to the node structure or NULL if the named node couldn't be found.

Implementation

Folio call implemented in kernel folio V20.

Associated Files

list.h
: ANSI C Prototype

clib.lib
: ARM Link Library

See Also

FirstNode(), LastNode(), NextNode(), PrevNode()

FindNodeFromHead

Returns a pointer to a node appearing at a given ordinal position from the head of the list.

Synopsis

```
Node *FindNodeFromHead(const List *l, uint32 position);
```

Description

This function scans the supplied list and returns a pointer to the node appearing in the list at the given ordinal position. NULL is returned if the list doesn't contain that many items.

Arguments

l
: A pointer to the list to scan for the node.

position
: The node position to look for.

Return Value

A pointer to the node found, or NULL if the list doesn't contain enough nodes.

Implementation

Convenience call implemented in clib.lib V24.

Associated Files

list.h
: ANSI C Prototype

clib.lib
: ARM Link Library

Caveats

GIGO (\Qgarbage in, garbage out)

See Also

AddHead(), AddTail(), InsertNodeFromHead(), InsertNodeFromTail(), RemHead(), RemNode(), RemTail()

FindNodeFromTail

Returns a pointer to a node appearing at a given ordinal position from the tail of the list.

Synopsis

```
Node *FindNodeFromTail(const List *l, uint32 position);
```

Description

This function scans the supplied list and returns a pointer to the node appearing in the list at the given ordinal position counting from the end of the list. NULL is returned if the list doesn't contain that many items.

Arguments

l
: A pointer to the list to scan for the node.

position
: The node position to look for, relative to the end of the list.

Return Value

A pointer to the node found, or NULL if the list doesn't contain enough nodes.

Implementation

Convenience call implemented in clib.lib V24.

Associated Files

list.h
: ANSI C Prototype

clib.lib
: ARM Link Library

Caveats

GIGO (garbage in, garbage out)

See Also

AddHead(), AddTail(), InsertNodeFromHead(), InsertNodeFromTail(), RemHead(), RemNode(), RemTail()

FindTask

Finds a task by name.

Synopsis

```
Item FindTask( const char *name )
```

Description

This macro finds a task with the specified name. The search is not case-sensitive (that is, the kernel does not distinguish uppercase and lowercase letters in task names). You can use this macro in place of `FindNamedItem()` to find a task.

Arguments

name
: The name of the task to find.

Return Value

The macro returns the item number of the task that was found or an error code if an error occurs.

Implementation

Macro implemented in task.h V20.

Associated Files

task.h
: ANSI C Macro

See Also

`CreateThread()`

GetBankBits

Finds out which VRAM bank contains a memory location.

Synopsis

```
uint32 GetBankBits( const void *a )
```

Description

In the current 3DO hardware, 2 MB of video random-access memory (VRAM) is divided into two banks, each containing 1 MB of VRAM. Fast VRAM memory transfers using the SPORT bus can take place only between memory locations in the same bank. Use this procedure to find out which VRAM bank contains a specified memory location.

When a task allocates VRAM, the memory block contains memory from one VRAM bank or the other; it never contains memory from both banks. Thus, if any memory location in a block is in one VRAM bank, the entire block is guaranteed to be in the same bank.

Arguments

a
: A pointer to the memory to be examined.

Return Value

The procedure returns a set of memory flags in which the following flags can be set:

MEMTYPE_BANKSELECT
: Set if the memory is contained in VRAM.

MEMTYPE_BANK1
: Set if the memory is in the primary VRAM Bank.

MEMTYPE_BANK2
: Set if the memory is in the secondary VRAM bank.

If the memory is not in VRAM, the procedure returns 0.

Implementation

Macro implemented in mem.h V20.

Associated Files

mem.h
: ANSI C Macro

See Also

`GetMemType()`

GetFolioFunc

Returns a pointer to a folio function.

Synopsis

```
FolioFunc GetFolioFunc (const Folio *folio, int32 func)
```

Description

This macro returns the address of a folio function from a folio vector table.

Arguments

folio

A pointer to the folio item that contains the procedure. Use LookupItem() to get this pointer. For the item number of a folio (which you pass to LookupItem()), see the "Portfolio Items" chapter or call FindAndOpenFolio().

func

The index of the vector table entry for the procedure. This index (which is always a negative integer, because the table grows backward in memory) is listed in the header file for the folio that contains the procedure.

Implementation

Macro implemented in folio.h V24.

Associated Files

folio.h

See Also

CallFolio(), CallFolioRet()

GetMemTrackSize

Get the size of a block of memory allocated with MEMTYPE_TRACKSIZE.

Synopsis

```
int32 GetMemTrackSize( const void *p )
```

Description

This function returns the size that was used to allocate a block of memory. The block of memory must have been allocated using the MEMTYPE_TRACKSIZE flag, otherwise this function will return garbage.

Arguments

p
: Pointer obtained from a system memory allocation routine. The block of memory must have been allocated using the MEMTYPE_TRACKSIZE flag, otherwise the value returned by this function will be random.

Return Value

The function returns the size, in bytes, of the memory block. This size corresponds to the size provided to the system memory allocation routine when the block was first allocated.

Implementation

Folio call implemented in kernel folio V24.

Associated Files

mem.h
: ANSI C Prototype

clib.lib
: ARM Link Library

Caveats

This function will not return the correct value for allocations made using MEMTYPE_STARTPAGE and using a private MemList structure. Allocations done with `AllocMem()` will always work, since they use the task's memory lists and not any private MemList.

See Also

`AllocMem()`, `FreeMem()`

GetNodeCount

Counts the number of nodes in a list.

Synopsis

```
uint32 GetNodeCount(const List *l);
```

Description

This function counts the number of nodes currently in the list.

Arguments

l
: A pointer to the list to count the nodes of.

Return Value

The number of nodes in the list.

Implementation

Convenience call implemented in clib.lib V24.

Associated Files

list.h
: ANSI C Prototype

clib.lib
: ARM Link Library

Caveats

GIGO (garbage in, garbage out)

See Also

AddHead(), AddTail(), InsertNodeFromHead(), InsertNodeFromTail(), RemHead(), RemNode(), RemTail()

GetNodePosFromHead

Gets the ordinal position of a node within a list, counting from the head of the list.

Synopsis

```
int32 GetNodePosFromHead(const List *l, const Node *n);
```

Description

This function scans the supplied list looking for the supplied node, and returns the ordinal position of the node within the list. If the node doesn't appear in the list, the function returns -1.

Arguments

l
: A pointer to the list to scan for the node.

n
: A pointer to a node to locate in the list.

Return Value

The ordinal position of the node within the list counting from the head of the list, or -1 if the node isn't in the list. The first node in the list has position 0, the second node has position 1, etc.

Implementation

Convenience call implemented in clib.lib V24.

Associated Files

list.h
: ANSI C Prototype

clib.lib
: ARM Link Library

Caveats

GIGO (garbage in, garbage out)

See Also

AddHead(), AddTail(), InsertNodeFromHead(), InsertNodeFromTail(), RemHead(), RemNode(), RemTail()

GetNodePosFromTail

Gets the ordinal position of a node within a list, counting from the tail of the list.

Synopsis

```
int32 GetNodePosFromTail(const List *l, const Node *n);
```

Description

This function scans the supplied list backward looking for the supplied node, and returns the ordinal position of the node within the list. If the node doesn't appear in the list, the function returns -1.

Arguments

l
: A pointer to the list to scan for the node.

n
: A pointer to a node to locate in the list.

Return Value

The ordinal position of the node within the list counting from the tail of the list, or -1 if the node isn't in the list. The last node in the list has position 0, the second to last node has position 1, etc.

Implementation

Convenience call implemented in clib.lib V24.

Associated Files

list.h
: ANSI C Prototype

clib.lib
: ARM Link Library

Caveats

GIGO (garbage in, garbage out)

See Also

AddHead(), AddTail(), InsertNodeFromHead(), InsertNodeFromTail(), RemHead(), RemNode(), RemTail()

GetPageSize

Gets the number of bytes in a memory page.

Synopsis

```
int32 GetPageSize( uint32 memflags )
```

Description

This procedure gets the number of bytes in a page of the specified type of memory.

Arguments

memflags
: Flags that specify the type of memory whose page size to get. These flags can include MEMTYPE_ANY, MEMTYPE_VRAM, MEMTYPE_DRAM, MEMTYPE_BANKSELECT, MEMTYPE_BANK1, MEMTYPE_BANK2, MEMTYPE_DMA, MEMTYPE_CEL, MEMTYPE_AUDIO, MEMTYPE_DSP, and MEMTYPE_SYSTEMPAGESIZE. For information about these flags, see the description of AllocMem().

Return Value

The procedure returns the size, in bytes, of a page of the specified type of RAM. If there is no memory of the specified type, the procedure returns 0.

Implementation

Folio call implemented in kernel folio V20.

Associated Files

mem.h
: ANSI C Prototype

clib.lib
: ARM Link Library

Caveats

If you do not specify the type of memory you're asking about, the page size you get may be for memory you're not interested in. The page size for a particular piece of memory may be not what you expected if the amount of memory requested causes the allocator to jump to a different type of memory that can satisfy your request, but which may have a different page size. To ensure that you get the actual page size of the memory you're interested, call `GetMemType()` to get the memory flags for the memory, then call `GetPageSize()` with those flags as the memflags argument.

See Also

`GetMemType()`

GetSysErr

Gets the error string for an error.

Synopsis

```
int32 GetSysErr( char *buff, int32 buffsize, Err err )
```

Description

This procedure returns a character string that describes an error code. The resulting string is placed in a buffer.

The procedure interprets all the fields in the err argument and returns corresponding text strings for all of them. If an error-text table is not found for the specific error, the routine uses numbers for the final error code value.

Arguments

buff
: A pointer to a buffer to hold the error string.

buffsize
: The size of the error-text buffer, in bytes. This should be 128.

err
: The error code whose error string to get.

Return Value

The procedure returns the number of bytes in the description string, or a negative error code if a bad buffer pointer is supplied.

Implementation

Folio call implemented in kernel folio V20.

Associated Files

operror.h
: ANSI C Prototype

clib.lib
: ARM Link Library

See Also

`PrintfSysErr()`

PrintfSysErr

Prints the error string for an error.

Synopsis

```
void PrintfSysErr( Err err )
```

Description

This procedure prints the error string for an error to the debugging terminal for a 3DO development system. This has the same effect as using `Printf()` to print the string constructed by `GetSysErr()`. The string is not displayed on production machines.

To copy an error string into a buffer instead of sending it the console, use `GetSysErr()`.

Arguments

err
: An error code.

Implementation

Convenience call implemented in clib.lib V20.

Associated Files

operror.h
: ANSI C Prototype

clib.lib
: ARM Link Library

See Also

`GetSysErr()`

GetTaskSignals

Gets the currently received signal bits.

Synopsis

```
int32 GetTaskSignals( Task *t )
```

Description

This macro returns the signal bits that have been received by the specified task. For information about signals, see the description of the `AllocSignal()` procedure and the "Communicating Among Tasks" chapter in the 3DO Portfolio Programmer's Guide.

Return Value

A 32-bit word in which all currently received signal bits for the task are set.

Implementation

Macro implemented in task.h V21.

Associated Files

task.h
: ANSI C Macro

See Also

`AllocSignal()`, `FreeSignal()`, `SendSignal()`, `WaitSignal()`

InsertNodeAfter

Inserts a node into a list after another node already in the list.

Synopsis

```
void InsertNodeAfter(Node *oldNode, Node *newNode);
```

Description

This function lets you insert a new node into a list, AFTER another node that is already in the list.

Arguments

oldNode
: The node after which to insert the new node.

newNode
: The node to insert in the list.

Implementation

Convenience call implemented in clib.lib V24.

Associated Files

list.h
: ANSI C Prototype

clib.lib
: ARM Link Library

Notes

A node can be included only in one list.

Caveats

GIGO (garbage in, garbage out)

See Also

AddHead(), AddTail(), InsertNodeFromHead(), InsertNodeFromTail(), RemHead(), RemNode(), RemTail(), InsertNodeBefore()

InsertNodeBefore

Inserts a node into a list before another node already in the list.

Synopsis

```
void InsertNodeBefore(Node *oldNode, Node *newNode);
```

Description

This function lets you insert a new node into a list, BEFORE another node that is already in the list.

Arguments

oldNode
: The node before which to insert the new node.

newNode
: The node to insert in the list.

Implementation

Convenience call implemented in clib.lib V24.

Associated Files

list.h
: ANSI C Prototype

clib.lib
: ARM Link Library

Notes

A node can be included only in one list.

Caveats

GIGO (garbage in, garbage out)

See Also

AddHead(), AddTail(), InsertNodeFromHead(), InsertNodeFromTail(), RemHead(), RemNode(), RemTail(), InsertNodeAfter()

IsEmptyList

Checks whether a list is empty.

Synopsis

```
bool IsEmptyList( List *l )
```

Description

This macro checks whether a list is empty.

Arguments

l
: A pointer to the list to check.

Return Value

The macro returns TRUE if the list is empty or FALSE if it isn't.

Implementation

Macro implemented in list.h V20.

Associated Files

list.h
: ANSI C Macro definition

See Also

FirstNode(), IsNode(), IsNodeB(), LastNode(), NextNode(), PrevNode(), ScanList(), IsListEmpty()

IsItemOpened

Determines whether a task or thread has opened a given item.

Synopsis

```
Err IsItemOpened( Item task, Item i )
```

Description

This function determines whether a task or thread has currently got an item opened.

Arguments

task
: The task or thread to inquire about. For the current task, use:

```
KernelBase->kb_CurrentTask>
```

i
: The number of the item to verify.

Return Value

This function returns >=0 if the item was opened, or a negative error code if it is not, or if the parameters are bogus.

Implementation

Folio call implemented in kernel folio V24.

Associated Files

item.h
: ANSI C Prototype

clib.lib
: ARM Link Library

See Also

OpenItem(), CloseItem(), CheckItem(), LookupItem()

IsMemReadable

Determines whether a region of memory is fully readable by the current task.

Synopsis

```
bool IsMemReadable( const void *p, int32 size )
```

Description

This function considers the described block of the address space in relation to the known locations of RAM in the system, and returns TRUE if the block is entirely contained within RAM, and FALSE otherwise.

The low 512 (0x200) bytes of the address space have been declared to be "not memory" for all reasonable purposes; this function will therefore return FALSE for any block of memory that uses any portion of low memory.

Arguments

p
: A pointer to the start of the block.

size
: The number of bytes in the block.

Return Value

This function returns TRUE if the block is entirely readable by the current task, or FALSE if any part of the block is not.

Implementation

Folio call implemented in kernel folio V24.

Associated Files

mem.h

ANSI C Prototype

clib.lib
ARM Link Library

See Also

`IsMemWritable()`

IsMemWritable

Determines whether a region of memory is fully writable by the current task.

Synopsis

```
bool IsMemWritable( const void *p, int32 size )
```

Description

This function considers the described block of the address space in relation to the pages of memory the current task has write access to. This function returns TRUE If the current task can write to the memory block, and FALSE if it cannot.

Arguments

p
: A pointer to the start of the block.

size
: The number of bytes in the block.

Return Value

This function returns TRUE if the block can be written to by the calling task, and FALSE if it cannot.

Implementation

Folio call implemented in kernel folio V24.

Associated Files

mem.h
: ANSI C Prototype

clib.lib
: ARM Link Library

See Also

IsMemReadable()

PrintError

Prints the error string for an error

Synopsis

```
void PrintError( char *who, char *what, char *whom, Err err )
```

Description

On DEVELOPMENT builds, this macro calls a procedure that prints the error string for an error to the Macintosh console for a 3DO development system. This has the same effect as using `printf()` to print the who, what, and whom strings with the string constructed by `GetSysErr()`.

On non-DEVELOPMENT builds, this macro expands to an empty statement, reducing code size and avoiding pulling in lots of extraneous junk from the support libraries.

To copy an error string into a buffer instead of sending it the console, use `GetSysErr()`.

In the task called "shell", the C statement

PrintError((char *)0, "open", filename, errno)

might result in something like this:

+=============== Terminal ================ | FFS-Severe-System-extended-No such file | shell: unable to open "badfile"

Arguments

who
: A string that identifies the task, folio, or module that encountered the error; if a NULL pointer or a zero-length string is passed, the name of the current task is used (or, if there is no current task, "notask").

what
: A string that describes what action was being taken that failed. Normally, the words "unable to" are printed before this string to save client data space; if the first character of the "what" string is a backslash, these words are not printed.

whom
: A string that describes the object to which the described action is being taken (for instance, the name of a file). It is printed surrounded by quotes; if a NULL pointer or zero length string is passed, the quotes are not printed.

err
: A (normally negative) number denoting the error that occurred, which will be decoded with GetSysErr. If not negative, then no decoded error will be printed.

Implementation

Macro implemented in the operror.h V21.

Associated Files

operror.h
: ANSI C Prototype

clib.lib
: ARM Link Library

See Also

`PrintfSysErr()` , `GetSysErr()`

ReadHardwareRandomNumber

Gets a 32-bit random number.

Synopsis

```
uint32 ReadHardwareRandomNumber( void )
```

Description

The procedure returns a random number generated by the hardware. This is useful for providing a seed to the software random-number generator.

Return Value

The procedure returns an unsigned 32-bit random number. The number can be any integer from 0 to (2^32 - 1), inclusive.

Implementation

SWI implemented in kernel folio V20.

Associated Files

hardware.h
ANSI C Prototype

SampleSystemTime

Samples the system time with very low overhead.

Synopsis

```
uint32 SampleSystemTime( void )
```

Description

This function samples the current system time, and returns it to you. The primary return value is the seconds count of the system clock. A secondary value available in the r1 register on ARM processors is the microseconds count of the system clock. For a C-friendly version of this call, see `SampleSystemTimeTV()`.

This function has very low overhead and is meant for high-accuracy timings.

The time value returned by this function corresponds to the time maintained by the TIMERUNIT_USEC unit of the timer device.

Return Value

This function returns the seconds count of the system clock. It also returns the microseconds count in the r1 ARM processor register.

Implementation

SWI implemented in kernel folio V24.

Associated Files

time.h

See Also

`SampleSystemTimeTV()` ANSI C Prototype

SampleSystemTimeTV

Samples the system time with very low overhead.

Synopsis

```
void SampleSystemTimeTV( TimeVal *time )
```

Description

This function records the current system time in the supplied TimeVal structure. This is a very low overhead call giving a very high-accuracy timing.

The time value returned by this function corresponds to the time maintained by the TIMERUNIT_USEC unit of the timer device.

Arguments

time
: A pointer to a TimeVal structure which will receive the current system time.

Implementation

Convenience call implemented in clib.lib V24.

Associated Files

time.h
: ANSI C Prototype

clib.lib
: ARM Link Library

See Also

SampleSystemTime()

SetItemOwner

Changes the owner of an item.

Synopsis

```
Err SetItemOwner( Item i, Item newOwner )
```

Description

This procedure makes another task the owner of an item. A task must be the owner of the item to change its ownership. The item is removed from the current task's resource table and placed in the new owner's resource table.

You normally use this procedure to keep a resource available after the task that created it terminates.

Arguments

i
: The item number of the item to give to a new owner.

newOwner
: The item number of the new owner task.

Return Value

The procedure returns 0 if ownership of the item is changed or an error code if an error occurs.

Implementation

SWI implemented in kernel folio V20.

Associated Files

item.h
: ANSI C Prototype

Caveats

This is not implemented for all items that should be able to have their ownership changed.

See Also

`CreateItem()`, `FindItem()`, `DeleteItem()`

SetItemPri

Changes the priority of an item.

Synopsis

```
int32 SetItemPri( Item i, uint8 newpri )
```

Description

This procedure changes the priority of an item. Some items in the operating system are maintained in lists; for example, tasks and threads. Some lists are ordered by priority, with higher-priority items coming before lower-priority items. When the priority of an item in a list changes, the kernel automatically rearranges the list of items to reflect the new priority. The item is moved immediately before the first item whose priority is lower.

A task must own an item to change its priority. A task can change its own priority even if it does not own itself.

Arguments

i
: The item number of the item whose priority to change.

newpri
: The new priority for the item.

Return Value

The procedure returns the previous priority of the item or an error code if an error occurs.

Implementation

SWI implemented in kernel folio V20.

Associated Files

item.h

ANSI C Prototype

Notes

For certain item types, such as devices, the kernel may change the priority of an item to help optimize throughput.

Caveats

This function is not implemented for all known item types.

See Also

`CreateItem()`

SetNodePri

Changes the priority of a list node.

Synopsis

```
uint8 SetNodePri( Node *n, uint8 newpri )
```

Description

This procedure changes the priority of a node in a list. The kernel arranges lists by priority, with higher-priority nodes coming before lower-priority nodes. When the priority of a node changes, the kernel automatically rearranges the list to reflect the new priority. The node is moved immediately before the first node whose priority is lower.

Arguments

n
: A pointer to the node whose priority to change.

newpri
: The new priority for the node. This can be a value from 0 to 255.

Return Value

The procedure returns the previous priority of the node.

Implementation

Folio call implemented in kernel folio V20.

Associated Files

list.h
: ANSI C Prototype

clib.lib
: ARM Link Library

Caveats

GIGO (garbage in, garbage out)

See Also

`InsertNodeFromHead`(), `InsertNodeFromTail`(), `UniversalInsertNode`()

Yield

Give up the CPU to a task of equal priority.

Synopsis

```
void Yield( void )
```

Description

In Portfolio, high-priority tasks always have precedence over lower priority tasks. Whenever a high priority task becomes ready to execute, it will instantly interrupt lower priority tasks and start running. The lower priority tasks do not get to finish their time quantum, and just get put into the system's ready queue for future scheduling.

If there are a number of tasks of equal priority which are all ready to run, the kernel does round-robin scheduling of these tasks. This is a process by which each task is allowed to run for a fixed amount of time before the CPU is taken away from it, and given to another task of the same priority. The amount of time a task is allowed to run before being preempted is called the task's "quantum".

The purpose of the `Yield()` function is to let a task voluntarily give up the remaining time of its quantum. Since the time quantum is only an issue when the kernel does round-robin scheduling, it means that `Yield()` actually only does something when there are multiple ready tasks at the same priority. However, since the yielding task does not know exactly which task, if any, is going to run next, `Yield()` should not be used for implicit communication amongst tasks. The way to cooperate amongst tasks is using signals, messages, and semaphores.

In short, if there are higher-priority tasks in the system, the current task will only run if the higher-priority tasks are all in the wait queue. If there are lower-priority tasks, these will only run if the current task is in the wait queue. And if there are other tasks of the same priority, the kernel automatically cycles through all the tasks, spending a quantum of time on each task, unless a task calls `Yield()`, which will cut short its quantum.

If there are no other ready tasks of the same priority as the task that calls `Yield()`, then that task will keep running as if nothing happened.

`Yield()` is only useful in very specific circumstances. If you plan on using it, think twice. In most cases it is not needed, and using it will result in a decrease in performance.

Implementation

SWI implemented in kernel folio V20.

Associated Files

task.h
: ANSI C Prototype

See Also

`WaitSignal()`

Tasks and Threads

Overview

This chapter provides the background and necessary programming details to create and run tasks and threads. Tasks and threads, although similar, function differently in certain key respects. For details about the calls discussed, see Kernel Folio Calls, in *3DO System Programmer's Reference*.

This chapter contains the following topics:

What Is a Task?

Tasks can be viewed as independent programs, with their own unique memory space. Tasks can spawn other tasks. These spawned tasks (child tasks) quit when the spawning task (parent task) quits, unless the parent task explicitly transfers ownership of the child task to another task before it quits. Although parent and child tasks are tied together when it's time to quit, they don't share memory. They can't share common data structures, and the child task must allocate its own memory pages, which, if its memory requirements are small, can be wasteful.

Threads are a different kind of spawned or child task, which is more tightly bound to the parent task than a standard child task. A thread shares the parent task's memory. The ownership of a thread cannot be transferred to a task that is outside the parent's task family. When the parent dies, each of its threads dies with it because the threads are considered resources of the parent task.

The calls described in this chapter handle both tasks and threads. To use them, include the task.h header file in your application. This file contains the definitions for the tag arguments and data structures, plus the prototypes, for task-related functions.

Starting and Ending Tasks

A user task on a CD can start on boot-up if it's named "LaunchMe." The operating system loads on boot-up and runs the executable code in LaunchMe, which starts a task that becomes the child of the shell-a user task. Once the user task is running, it can spawn child tasks in a manner similar to someone running a program from a terminal-the original user task uses a filesystem call to supply the filename of a file containing the executable code of a second task. The kernel launches the second task as a child of the first task.

Creating a Task

Each task is an item in the system. Tasks are created through the `CreateItem()` function call, a low-level call that returns the item number of the new task. (`CreateItem()` is discussed in Managing Items.)

Each task has a TCB (task control block) associated with it. The TCB data structure is created and maintained by the kernel; a user task can't write to the TCB directly. The settings of the various fields in the TCB are determined by the kind of task (privileged or non-privileged), the parameters passed to the kernel when the task is started, and changes made to the task while it runs. The kernel uses the TCB to allocate and control task resources, memory usage, and so on.

While you can create a task using `CreateItem()`, it's easier to use the following File folio calls which in turn call `CreateItem()`:

```
Item LoadProgram( char *path )
Item LoadProgramPrio( char *path, int32 prio )
```

`LoadProgram()` launches a child task that has the same priority as the launching, or parent, task. The `path` argument for `LoadProgram()` contains the pathname for the file containing the executable code for the task. This parameter can also include arguments for the application, if arguments are required. Specifying the `path` argument is very much like specifying a program to run, with arguments, on a UNIX system.

`LoadProgramPrio()` allows you to specify a priority for the task. As with `LoadProgram(),` you include a `path` argument that specifies your program file and any arguments that you wish to pass to the task when it begins. The second argument, `priority`, specifies a new priority for the child task so it can run at a higher or lower priority than the parent task.

Both `LoadProgram()` and `LoadProgramPrio()` return the item number of the newly started task.

Example 1 shows how to set up a text string for your program and launch it:

Example 1: *Setting up a text string and launching it as a task.*

```
char *cmdline = "mysrc/testprogram arg1 arg2"
Item    task_id
...
{
...
task_id = LoadProgram(cmdline);
...
}
```

Setting Priority

If you use `LoadProgramPrio()`, you should get the parent task's priority before setting up a new priority value for the child task. You can read the current task's priority by reading the `n_Priority` field of the ItemNode data structure that's embedded in the TCB of each task- `CURRENTTASK->t.n_Priority` as shown in Example 2:

Example 2: *Getting a parent task's priority.*

```
/* Determine the priority of the current task. */
Priority = CURRENTTASK->t.n_Priority;
```

The Life of a Task

Child tasks created through `LoadProgram()` or `LoadProgramPrio()` allocate completely separate memory areas for the parent task and its child tasks. Once a child task is launched, it remains in the system as an item until:

- The parent task naturally ends (the program completes its function and dies)
- The parent task quits (if it still owns the child task)
- The task that owns the child task explicitly deletes it using `DeleteItem()`

The launching task remains the owner of the child task throughout the life of the child task unless it passes ownership to another task.

Ending a Task

Typically, a child task ends when its parent task ends. It's good practice to end a child task, before the parent task quits, if you no longer need it to free up resources. To end a task, use `DeleteItem(),` passing in the task's item number. An error code (a negative number) is returned if an error occurs.

```
Err DeleteItem( Item i )
```

Alternatively, a child task can end itself by calling `exit()` or just return:

```
void exit (int status)
```

The function `exit()` never returns.

Note: The return or exit status can be passed on to the parent task through an exit message if the child was created with `CREATETASK_TAG_MSGFROMCHILD`.

Deleting the child task lets the kernel clean up all the task's allocated resources, freeing that space for other tasks.

Managing Items

This chapter provides the programming details necessary to create or open items, manage items and, when finished, delete or close items. For more information about the function calls in this chapter, see see Kernel Folio Calls, in *3DO System Programmer's Reference*.

This chapter contains the following topics:

About Items

Items are system-maintained handles for shared resources. Each item contains a globally unique identification number and a pointer to the resource. Items can refer to any one of many different system components such as data structures, I/O devices, folios, tasks, and so on. You need to understand and use the item function calls in order to use these various system components.

The procedure for working with an item is the same, regardless of the item type. The procedure is as follows:

1. Define the necessary parameters (tag arguments) for the item you want.
2. Create or open the item using the tag arguments.
3. Use the item.
4. Delete or close the item when you're finished with it.

Portfolio provides a set of kernel calls that manage items while they're in use. These calls allow a task to check for the existence of an item, find the item number of a named object, get a pointer to an item, lock and unlock an item, and change the item's priority and owner.

To create items for you application, you need to include the following header files:

- *kernelnodes.h*
- *nodes.h*
- *folio.h*
- *item.h*
- *types.h*

These header files declare structures, types, parameters, routine definitions, and other definitions needed for items. You also need the include file for some items that are associated with that folio. For example, to create an item used by the Audio folio, you'll have to include the *audio.h* file.

The kernel creates and keeps track of items in system memory. When the kernel creates an item, it creates a node for that item using the ItemNode data structure:

```
typedef struct ItemNode
{
    struct ItemNode *n_Next; /*pointer to next itemnode in */

                                /* list */
    struct ItemNode *n_Prev; /*pointer to previous itemnode in
```

```
                                /* list */
    uint8 n_SubsysType;    /* what folio manages this node */
    uint8 n_Type;         /* what type of node for the folio */
    uint8 n_Priority;     /* queueing priority */
    uint8 n_Flags;         /* misc flags, see below */
    int32 n_Size;         /* total size of node including hdr */
    char *n_Name;         /* ptr to null terminated string or NULL*/
    uint8 n_Version;     /* version of of this itemnode */
    uint8 n_Revision;     /* revision of this itemnode */
    uint8 n_FolioFlags;    /* flags for this item's folio */
    uint8 n_ItemFlags;    /* additional system item flags */
    Item  n_Item;         /* ItemNumber for this data structure */
    Item  n_Owner;         /* creator, present owner, disposer */
    void *n_ReservedP;    /* Reserved pointer */
} ItemNode, *ItemNodeP;
```

The fields in this data structure define the status of the item and give information about the item. User tasks can't change their values, but they can look at them for information such as the version and revision numbers of an item (useful for items such as folios or devices). `LookupItem()`, described later, returns a pointer to the ItemNode structure for a specific item.

Creating or Opening an Item

To use most items such as message ports and semaphores, a task must first create them. The system, however, supplies some items premade on disk: folios, devices, and drivers. To use a folio or a device, a task opens it instead of creating it from scratch.

The difference between item creation and item opening is important: a created item is owned by the creating task, which can delete the item at any time. A created item is automatically deleted if the owning task quits without deleting it. On the other hand, an opened item is owned by the original creator and is shared among tasks. When a task opens the item, the item is loaded into memory and the kernel registers the opening task as a user. When another task opens the same item, the kernel registers that task as well.

Creating an Item

To create an item, use the `CreateItem()` function call:

```
Item CreateItem( int32 ct, const TagArg *p )
```

`CreateItem()` takes two arguments. The first, `ct`, specifies the type of folio in which the item is used and, the type of item within that folio. This value must always be present-it defines the item type. To generate this value, use the `MkNodeID()` macro (described later), which accepts the folio type and folio item values, and then returns the proper value for `CreateItem()`.

The second argument, `p`, is a pointer to additional arguments that define the characteristics of the item. These arguments, known as tag arguments, are specific to each item type, and are discussed later in the chapter.

`CreateItem()` returns the item number of the newly created item or an error code (a negative value) if an error occurs.

Specifying the Item Type

To come up with the item type value for `CreateItem()`, use `MkNodeID()`:

```
int32 MkNodeID( Int32 folioNum, Int32 itemTypeNum )
```

The macro accepts two values: `folioNum`, which specifies the number of the folio responsible for the item, and `itemTypeNum`, which specifies the item type for the item being created. This item type number is folio-specific. Both values are set by constants defined within the header file for the appropriate folio. For example, audio.h defines `AUDIONODE` as the constant that specifies the Audio folio, and lists audio folio item constants such as `AUDIO_TEMPLATE_NODE`. The Portfolio I/O Model lists the constants for all the folio types and folio items used for each folio. To use those constants, you must be sure to include the .h file of the proper folio. The constants for kernel items, such as message ports or tasks, are in the *kernelnodes.h* file.

User tasks can't create the following items: drivers, devices, firqs, and timers. These items can only be created by privileged tasks.

Tag Arguments

The values of the tag arguments determine the values of the fields in an item's ItemNode structure. `CreateItem()` creates an ItemNode structure for a new item. The call reads the values of the tag arguments, interprets them, and writes the corresponding values in the ItemNode fields.

All item types use a standard set of tag arguments defined in *item.h*. Many items also use additional tag arguments to specify properties that are unique to those item types.

To define values for an item's associated tag arguments, you must first declare an array of type `TagArg`, which is defined in *types.h* as follows:

Example 1: *Defining values for an item's associated tag argument.*

```
typedef struct TagArg
{
    uint32  ta_Tag;
    TagData ta_Arg;
} TagArg, *TagArgP;
```

The first field, `ta_Tag`, is a value that specifies the type of tag argument. The second field, `ta_Arg`, is a 32-bit value that contains the actual argument.

The standard tag argument types for all items are defined in *item.h*. They include:

- **TAG_ITEM_PRI,** sets the priority of an item.
- **TAG_ITEM_NAME**, sets the name of an item.
- **TAG_ITEM_VERSION**, sets the version number of an item.
- **TAG_ITEM_REVISION**, sets the revision number of an item.
- **TAG_ITEM_UNIQUE_NAME**, specifies the item must be uniquely named within its type.
- **TAG_ITEM_END or TAG_END**, ends a list of tag arguments.

The value for each of these arguments is precisely the same as the value for corresponding fields in ItemNode. When an item is created, these values are simply written into the appropriate ItemNode fields.

You'll find custom tag arguments used for each item type in Portfolio Items, in the *3DO System Programmer's Reference*.

When you provide tag arguments for an item, some of the arguments for that item, are required and some are optional. You must provide values for required tag arguments. You must also provide specific values for optional tag arguments, because any values that you do not provide are filled in with default values by the operating system when it creates the item. Defaults aren't guaranteed to be 0, so you must provide the value of 0 if that's what you want set for a tag argument.

If an entire array of tag arguments is optional and you want to use the system default values for all the arguments, then you do not need to declare the TagArg array in your application. Instead, pass NULL as the pointer to the tag arguments argument. When you create a tag arguments array, its general format is:

Example 2: *Tag arguments array.*

```
TagArg            TagArrayName[] =
{
    command tag,    value assigned,
    ...    ...
    TAG_END,    0
};
```

`TagArrayName` is the name of the array for the tag argument values; `command tag` is the name of the tag argument, and `value assigned` is the value assigned to that tag argument. The last tag argument in an array is always called `TAG_END`, and must be set to 0. This signals the end of the tag argument list.

VarArgs Tag Arguments

Most functions in Portfolio that accept tag arguments have a split personality. The standard version of a function has a parameter of type `const TagArg *`. A varargs version of a function, which is denoted with the VA suffix, `CreateItemVA()` for example. The varargs functions offer a much more convenient way to provide the system with tag arguments.

When you use the varargs versions of a function, you do not need to create arrays of TagArg structures to pass to the function. Instead, you can construct a tag list on the stack directly within the function call. Example 3 shows a standard call with TagArg structures:

Example 3: *A standard function call with TagArg structures.*

```
{
TagArg tags[3];

    tags[0].ta_Tag = CREATEMSG_TAG_REPLYPORT;
    tags[0].ta_Arg = (void *)replyPort;
    tags[1].ta_Tag = CREATEMSG_TAG_MSG_IS_SMALL;
    tags[1].ta_Arg = 0;
    tags[2].ta_Tag = TAG_END;

    item = CreateItem(MKNODEID(KERNELNODE,MESSAGENODE),tags);
}
```

Instead of this cumbersome and error prone array, you can build the tag list on the stack using the `CreateItemVA()` function as shown in Example 4:

Example 4: *Using VarArgs to provide tag arguments.*

```
{
  item = CreateItemVA(MKNODEID(KERNELNODE,MESSAGENODE),
                      CREATEMSG_TAG_REPLYPORT,   replyPort,
                      CREATEMSG_TAG_MSG_IS_SMALL, 0,
                      TAG_END);
}
```

As you can see, this method of providing tag arguments is much easier to read and maintain. There is no casting needed, no array to maintain, and so on.

The Item Number

Once an item is created, the call that creates it returns a positive 32-bit number, which is the unique item number for the new item. A task uses this number whenever it needs to refer to the item in later calls-this item number is important, and should be stashed away in a handy variable. If an error occurs, a negative number is returned.

The Easy Way Out

Now that you've seen the details of creating items using `CreateItem(),` you should know that most folios provide higher-level convenience calls that do the same thing for commonly used items in a folio, which makes programming much easier. The convenience calls usually take arguments that define the characteristics of the item, translate arguments into the appropriate tag arguments, and then call a lower-level item creation call to create the item. Some of those calls include:

- `CreateIOReq()` creates an IOReq.
- `CreateScreenGroup()` creates a screen group.
- `CreateMsgPort()` creates a message port.
- `CreateMsg()` creates a message.
- `CreateSemaphore()` creates a semaphore.
- `CreateThread()` creates a thread.

Generally, use a convenience call whenever one is available for the type of item you want to create. You'll find details about these calls in the *3DO System Programmer's Reference* and in the chapters of

this volume that discuss specific folios.

Portfolio Items

This chapter provides lists of item types and system items. The following table is a brief description of each item.

- `Attachment` The binding of a sample or an envelope to an instrument.
- `Bitmap` Frame buffer into which rendering may be performed.
- `Cue` Audio cue item.
- `Envelope` Audio envelope.
- `ErrorText` A table of error messages.
- `File` A handle to a data file.
- `File alias` A character string for referencing the pathname to a file or a directory of files.
- `Folio`A family of function calls.
- `Instrument` DSP instrument item.
- `IOReq` An item used to communicate between a task and an I/O device.
- `Knob` An item for adjusting an instrument's parameters.
- `Locale` A database of international information.
- `Message` The means of sending data from one task to another.
- `Message port` An item through which a task receives messages.
- `Probe` An item to permit the CPU to read the output of a DSP instrument.
- `Sample` A digital recording of a sound.
- `ScreenGroup` A collection of one or more screens.
- `Semaphore` An item used to arbitrate access to shared resources.
- `Task` A process.
- `Template` A description of an audio instrument.
- `Tuning` An item that contains information for tuning an instrument.
- `VDL` A display control object.

Attachment

The binding of a sample or an envelope to an instrument.

Description

An attachment is the item to binds a sample or an envelope to a particular instrument.

An Attachment is associated with precisely one Envelope and one Instrument, or one Sample and one Instrument. An Attachment is said to be an Envelope Attachment if it is attached to an Envelope, or a Sample Attachment if attached to a Sample.

Sample attachments actually come in two flavors: one for input FIFOs and another for output FIFOs, defined by the Hook to which the Sample is attached. Both kinds are considered Sample Attachments and no distinction is made between them.

A single Instrument can have one Envelope Attachment per Envelope hook and one Sample Attachment per Output FIFO. A single Instrument can have multiple Sample Attachments per Input FIFO, but only one will be selected to be played when the instrument is started. This is useful for creating multi-sample instruments, where the sample selected to be played depends on the pitch to be played.

A single Sample or Envelope can have multiple Attachments made to it.

Folio

audio

Item Type

AUDIO_ATTACHMENT_NODE

Create

`AttachSample()`, `AttachEnvelope()`, `CreateItem()`

Delete

`DetachSample()`, `DetachEnvelope()`, `DeleteItem()`

Modify

SetAudioItemInfo()

Use

LinkAttachments(), MonitorAttachment(), ReleaseAttachment(), StartAttachment(), StopAttachment(), WhereAttachment()

Tags

AF_TAG_CLEAR_FLAGS
: (uint32) Modify. Set of AF_ATTF_ flags to clear. Clears every flag for which a 1 is set in ta_Arg. See also AF_TAG_SET_FLAGS. The final state of any flag is determined by the last occurrence of that flag in a AF_TAG_SET_FLAGS or AF_TAG_CLEAR_FLAGS.

AF_TAG_ENVELOPE
: (Item) Create. Envelope Item to attach to Instrument. Exactly one of AF_TAG_ENVELOPE or AF_TAG_SAMPLE must be specified.

AF_TAG_HOOKNAME
: (const char *) Create. The name of the sample or envelope hook in the instrument to attach to. For sample hooks, defaults to the first one listed for each instrument. For envelopes, defaults to "Env".

AF_TAG_INSTRUMENT
: (Item) Create. Instrument or Template item to attach envelope or sample to. Must be specified when creating an Attachment.

AF_TAG_SAMPLE
: (Item) Create. Sample Item to attach to instrument. Exactly one of AF_TAG_ENVELOPE or AF_TAG_SAMPLE must be specified.

AF_TAG_SET_FLAGS
: (uint32) Create, Modify. Set of AF_ATTF_ flags to set. Sets every flag for which a 1 is set in ta_Arg. See also AF_TAG_CLEAR_FLAGS. The final state of any flag is determined by the last occurrence of that flag in a AF_TAG_SET_FLAGS or AF_TAG_CLEAR_FLAGS.

AF_TAG_START_AT
: (int32) Create, Modify. Specifies the point at which to start when the attachment is started (with `StartAttachment()` or `StartInstrument()) .`

For sample attachments, specifies a sample frame number in the sample at which to begin playback.

For envelopes, attachments specifies the segment index at which to start.

AF_TAG_TIME_SCALE
: (ufrac16) Create. Scales all of the times in the attached envelopes by the supplied ufrac16. Defaults to 1.0. Only applies to envelope attachments.

Flags

AF_ATTF_FATLADYSINGS
: If set, causes the instrument to stop when the attachment finishes playing. This bit can be used to mark the one or more envelope(s) or sample(s) that are considered to be the determiners of when the instrument is completely done. For envelopes, the default setting for this flag comes from the AF_ENVF_FATLADYSINGS flag. Defaults to cleared for samples.

AF_ATTF_NOAUTOSTART
: When set, causes `StartInstrument()` to not automatically start this attachment. This allows later starting of the attachment by using `StartAttachment()`. Defaults to cleared (attachment defaults to starting when instrument is started).

Bitmap

Frame buffer into which rendering may be performed.

Description

A Bitmap is the logical destination for all graphics rendering operations. Bitmap items describe the dimensions of the frame buffer, its starting address in RAM, optional clipping area, and other characteristics.

Folio

graphics

Item Type

BITMAPNODE

Create

`CreateBitmap()`
`CreateBitmapVA()`
`CreateItem()`

Delete

`DeleteBitmap()`
`DeleteItem()`

Query

None

Modify

None

Use

CreateScreenGroup()

DisplayOverlay()

DrawCels()
GetPixelAddress()
SetCEControl()
SetCEWatchDog()
SetClipHeightt()
SetClipOrigin()
SetClipWidth()
SetReadAddress()
ReadPixel()
ResetReadAddress()
WritePixel()

Tags

The following tags must be present:

CBM_TAG_WIDTH(int32)
Sets the bitmap width, in pixels.

CBM_TAG_HEIGHT(int32)
Specifies the bitmap height, in pixels.

CBM_TAG_BUFFER(void *)
Specifies the starting address of the bitmap buffer.

The following tags are optional:

CBM_TAG_CLIPWIDTH(int32)
Specifies the width of the clipping region, in pixels. (default: the value of CBM_TAG_WIDTH)

CBM_TAG_CLIPHEIGHT(int32)
Specifies the height of the clipping region, in pixels. (default: the value of CBM_TAG_HEIGHT)

CBM_TAG_CLIPX(int32)
Specifies the left edge of the clipping region, in pixels. (default: 0)

CBM_TAG_CLIPY(int32) Specifies the top edge of the clipping region, in pixels. (default: 0)

CBM_TAG_WATCHDOGCTR(int32)
Specifies the cel engine timeout value, in msec. (default: 1000000)

CBM_TAG_CECONTROL(uint32)
Specifies the CEControl register value to be used when this bitmap is being rendered to by the cel engine. (default: (B15POS_PDC | B0POS_PPMP | CFBDSUB | CFBDLSB_CFBD0 | PDCLSB_PDC0))

Cue

Audio Cue Item.

Description

This Item type is used to receive completion notification from the Audio folio for these kinds of operations:

sample playback completion

envelope completion

audio timer request completion

Each Cue has a signal bit associated with it. Because signals are task-relative, Cues cannot be shared between multiple tasks.

Folio

audio

Item Type

AUDIO_CUE_NODE

Create

`CreateCue()`

`CreateItem()`

Delete

`DeleteCue()`

`DeleteItem()`

Query

GetCueSignal()

Modify

None

Use

AbortTimerCue()

MonitorAttachment()

SignalAtTime()

SleepUntilTime()

Tags

None

See Also

Attachment

Envelope

Audio envelope.

Description

An envelope is a time-variant control signal which can be used to control parameters of sounds that are to change over time (e.g. amplitude, frequency, filter characteristics, modulation amount, etc.).

Envelope Items use a function defined by a set of points in time-level space described by an array of DataTimePairs (defined in audio.h). The function is a continuous set of line segments drawn between the points in the DataTimePair array.

Envelopes are used in conjunction with Instruments that accept the Envelope Item's data and output the control signal (e.g. envelope.dsp).

Folio

audio

Item Type

AUDIO_ENVELOPE_NODE

Create

CreateEnvelope()

CreateItem()

Delete

DeleteEnvelope()

DeleteItem()

Query

None

Modify

SetAudioItemInfo()

Use

AttachEnvelope()

Tags

AF_TAG_ADDRESS
: (const DataTimePair *) Create, Modify. Pointer to array of DataTimePairs used to define the envelope. All of the points in this array must be sorted in increasing order by time. The length of the array is specified with AF_TAG_FRAMES.

AF_TAG_CLEAR_FLAGS
: (uint32) Create, Modify. Set of AF_ENVF_ flags to clear. Clears every flag for which a 1 is set in ta_Arg.

AF_TAG_FRAMES
: (int32) Create, Modify. Number of DataTimePairs in array pointed to by AF_TAG_ADDRESS.

AF_TAG_MICROSPERUNIT
: (int32) Create, Modify. Number of microseconds for each time unit specified in DataTimePairs and time related envelope tags. Defaults to 1000 on creation, which sets each time unit equal to one millisecond.

AF_TAG_RELEASEBEGIN
: (int32) Create, Modify. Index in DataTimePair array for beginning of release loop. -1 indicates no loop, which is the default on creation. If not -1, must <= the value set by AF_TAG_RELEASEEND.

AF_TAG_RELEASEEND
: (int32) Create, Modify. Index in DataTimePair array for end of release loop. -1 indicates no loop, which is the default on creation. If not -1, must >= the value set by AF_TAG_RELEASEBEGIN.

AF_TAG_RELEASEJUMP
: (int32) Create, Modify. Index in DataTimePair array to jump to on release. When set, release

causes escape from normal envelope processing to the specified index without disturbing the current output envelope value. From there, the envelope proceeds to the next DataTimePair from the current value. -1 to disable, which is the default on creation.

AF_TAG_RELEASETIME
(int32) Create, Modify. The time in units used when looping from the end of the release loop back to the beginning. Defaults to 0 on creation.

AF_TAG_SET_FLAGS
(uint32) Create, Modify. Set of AF_ENVF_ flags to set. Sets every flag for which a 1 is set in ta_Arg.

AF_TAG_SUSTAINBEGIN
(int32) Create, Modify. Index in DataTimePair array for beginning of sustain loop. -1 indicates no loop, which is the default on creation. If not -1, <= the value set by AF_TAG_SUSTAINEND.

AF_TAG_SUSTAINEND
(int32) Create, Modify. Index in DataTimePair array for end of sustain loop. -1 indicates no loop, which is the default on creation. If not -1, >= the value set by AF_TAG_SUSTAINBEGIN.

AF_TAG_SUSTAINTIME
(int32) Create, Modify. The time in units used when looping from the end of the sustain loop back to the beginning. Defaults to 0 on creation.

Flags

AF_ENVF_FATLADYSINGS
The state of this flag indicates the default setting for the AF_ATTF_FATLADYSINGS flag whenever this envelope is attached to an instrument.

See Also

Attachment, Instrument, envelope.dsp

Instrument

DSP Instrument Item.

Description

Instrument items are created from Template items. They correspond to actual instances of DSP code running on the DSP. Several Instruments can be connected together to create a patch. Instruments can be played (by starting them), and controlled (by tweaking knobs). An Instrument is monophonic in the sense that it corresponds to a single voice. Multiple voices require creating an Instrument per voice.

Folio

audio

Item Type

AUDIO_INSTRUMENT_NODE

Create

`AllocInstrument()`

`CreateInstrument()` (new for V24)

`CreateItem()`

`LoadInstrument()`

Delete

`DeleteInstrument()` (new for V24)

`DeleteItem()`

`FreeInstrument()`

UnloadInstrument()

Query

GetAudioItemInfo()

Modify

None

Use

AbandonInstrument()

AttachEnvelope()

AttachSample()

BendInstrumentPitch()

ConnectInstruments()

CreateProbe() (new for V24)

DisconnectInstruments()

GetKnobName()

GetNumKnobs()

GrabKnob()

PauseInstrument()

ReleaseInstrument()

ResumeInstrument()

StartInstrument()

StopInstrument()

TuneInstrument()

Tags

AF_TAG_AMPLITUDE
: (uint32) Start. Value to set instrument's Amplitude knob to before starting instrument (for instruments that have an Amplitude knob). Valid range is 0..0x7fff. Has no effect if Amplitude knob is connected to the output from another instrument. This tag is mutually exclusive with AF_TAG_VELOCITY.

AF_TAG_CALCRATE_DIVIDE
: (uint32) (new tag for V24) Create. Specifies the the denominator of the fraction of the total DSP cycles on which this instrument is to run. The only valid settings at this time are 1 to run on all DSP cycles (i.e. e and 2 to run on only 1/2 of the DSP cycles (i.e. execute at 22,050 cycles/sec). Defaults to 1.

AF_TAG_FREQUENCY
: (ufrac16) Start. Value to set Frequency knob to in 16.16 Hertz (for instruments that have a Frequency knob). Has no effect if Frequency knob is connected to the output from another instrument. This tag is mutually exclusive with AF_TAG_PITCH and AF_TAG_RATE.

AF_TAG_PITCH
: (uint32) Start. Value to set Frequency knob (for instruments that have a Frequency knob) expressed as a MIDI note number. The range is 0 to 127; 60 is middle C. For multisample instruments, picks the sample associated with the MIDI pitch number. This tag is mutually exclusive with AF_TAG_FREQUENCY and AF_TAG_RATE.

AF_TAG_PRIORITY
: (uint32) Create, Query. The priority of execution in DSP in the range of 0..255, where 255 is the highest priority. Defaults to 100 on creation.

AF_TAG_RATE
: (uint32) Start. Value to set Frequency knob to in instrument-specific frequency units (e.g. phase increment, proportion of original sample rate) for instruments that have a Frequency knob. Has no effect if Frequency knob is connected to the output from another instrument. This tag is mutually exclusive with AF_TAG_FREQUENCY and AF_TAG_PITCH.

AF_TAG_SET_FLAGS
: (uint32) Create. AF_INSF_ flags to set at creation time. Defaults to all cleared.

AF_TAG_START_TIME
: (AudioTime) Query. Returns the AudioTime value of when the instrument was last started.

AF_TAG_STATUS
: (uint32) Query. Returns the current instrument status: AF_STARTED, AF_RELEASED, AF_STOPPED, or AF_ABANDONED.

AF_TAG_TEMPLATE
: (Item) Create. DSP Template Item used from which to create instrument. Note: this tag cannot be used with `CreateInstrument()`.

AF_TAG_TIME_SCALE
: (ufrac16) Start, Release. Scale times for all Envelopes attached to this Instrument. Original value is derived from the AF_TAG_TIME_SCALE
provided when the Envelope Attachment was made. This value remains in effect until another AF_TAG_TIME_SCALE is passed to `StartInstrument()` or `ReleaseInstrument()`.

AF_TAG_VELOCITY
: (uint32) Start. MIDI note velocity indicating the value to set instrument's Amplitude knob to before starting instrument (for instruments that have an Amplitude knob). Valid range is 0..127. Has no effect if Amplitude knob is connected to the output from another instrument. This tag is mutually exclusive with AF_TAG_AMPLITUDE.

Flags

AF_INSF_AUTOABANDON
: When set, causes instrument to automatically go to the AF_ABANDONED state when stopped (either automatically because of an AF_ATTF_FATLADYSINGS flag, or manually because of a `StopInstrument()` call). Otherwise, the instrument goes to the AF_STOPPED state when stopped. Note that regardless of the state of this flag, instruments are created in the AF_STOPPED state.

See Also

`Template`, `Attachment`, `Knob`, `Probe`

Template

A description of an audio instrument.

Description

A Template is the description of a DSP audio instrument (including the DSP code, resource requirements, parameter settings, etc.) from which Instrument items are created. A Template can either be a standard system instrument template (e.g. envelope.dsp, sampler.dsp, etc) or a custom template created by ARIA.

Folio

audio

Item Type

AUDIO_TEMPLATE_NODE

Create

CreateInsTemplate()

CreateItem()

DefineInsTemplate()

LoadInsTemplate()

Delete

DeleteItem()

UnloadInsTemplate()

Use

AdoptInstrument()

Tags

AF_TAG_ALLOC_FUNCTION
(void *(*)(uint32 memsize, uint32 memflags)) Create. Sets custom memory allocation function to be called during template creation for objects that can be created in user memory. Currently this only applies to samples embedded in ARIA instruments. Defaults to AllocMem(). If you supply a custom allocation function you must also provide a custom free function with AF_TAG_FREE_FUNCTION.

AF_TAG_DEVICE
(Item) Create. Audio device Item for instrument template. 0 indicates the default audio device, the DSP, which is the only valid audio device item at the present time.

AF_TAG_FREE_FUNCTION
(void (*)(void *memptr, uint32 memsize)) Create. Sets custom memory free function to be called during template deletion. Defaults to FreeMem(). If you supply a custom free function you must also provide a custom allocation function with AF_TAG_ALLOC_FUNCTION.

AF_TAG_IMAGE_ADDRESS
(const char *) Create. Specifies a memory location containing a template file image. Must use in conjunction with AF_TAG_IMAGE_LENGTH. Mutually exclusive AF_TAG_NAME.

AF_TAG_IMAGE_LENGTH
(uint32) Create. Specifies number of bytes in template file image pointed to by AF_TAG_IMAGE_ADDRESS.

AF_TAG_LEAVE_IN_PLACE
(bool) Create. When TRUE and used in conjunction with AF_TAG_IMAGE_ADDRESS, causes any user memory objects read from the template file that would otherwise be copied into freshly allocated memory, to be used in-place (i.e. point to a part of the file image instead of allocating more memory). Currently this is only applies to samples embedded in an ARIA instrument. Mutually exclusive with AF_TAG_ALLOC_FUNCTION. See Sample Item and CreateSample() for more details and caveats.

Note that sample data may have to be moved down two bytes to align it properly for Portfolio DMA. This will destroy the sample image so that it cannot be reused.

AF_TAG_NAME
(const char *) Create. Name of template file to load. Mutually exclusive with AF_TAG_IMAGE_ADDRESS.

See Also

Instrument, Attachment

Knob

An item for adjusting an Instrument's parameters.

Description

A Knob is an Item for adjusting an Instrument's parameters.

Folio

audio

Item Type

AUDIO_KNOB_NODE

Create

CreateItem()

GrabKnob()

Delete

DeleteItem()

ReleaseKnob()

Query

GetAudioItemInfo()

Modify

None

Use

TweakKnob()

TweakRawKnob()

Tags

AF_TAG_CURRENT
: (int32) Query. Returns the current raw value of knob.

AF_TAG_DEFAULT
: (int32) Query. Returns the default raw value of knob.

AF_TAG_INSTRUMENT
: (Item) Create. Specifies from which instrument to grab knob.

AF_TAG_MAX
: (int32) Query. Returns maximum raw value of knob.

AF_TAG_MIN
: (int32) Query. Returns minimum raw value of knob.

AF_TAG_NAME
: (const char *) Create, Query. Knob name. On creation, specifies name of knob to grab. On query, returns a pointer to knob's name.

See Also

Instrument, Probe

Probe

An item to permit the CPU to read the output of a DSP instrument.

Description

A probe is an item that allows the CPU to read the output of a DSP instrument. It is sort of like the opposite of a knob. Multiple probes can be connected to a single output. A probe does not interfere with a connection between instruments.

Folio

audio (new for V24)

Item Type

AUDIO_PROBE_NODE

Create

`CreateProbe()`

`CreateItem()`

Delete

`DeleteProbe()`

`DeleteItem()`

Query

None

Modify

None

Use

`ReadProbe()`

Tags

None

ErrorText

A table of error messages.

Description

An error text item is a table of error messages. The kernel maintains a list of these items and uses them to convert a Portfolio error code into a descriptive string.

Folio

kernel

Item Type

ERRORTEXTNODE

Create

CreateItem()

Delete

DeleteItem()

Use

GetSysErr()

Tags

ERRTEXT_TAG_OBJID
: (uint32) Create. This tag specifies the 3 6-bit characters that idenify these errors. You use the Make6Bit() macro to convert from ASCII into the 6-bit character set.

ERRTEXT_TAG_MAXERR
: (uint8) Create. Indicates the number of error strings being defined. This corresponds to the number of entries in the string table.

ERRTEXT_TAG_TABLE

(char **) Create. A pointer to an array of string pointers. These strings will be used when converting an error code into a string. The array is indexed by the lower 8 bits of the error code.

ERRTEXT_TAG_MAXSTR

(uint32) Create. Specifies the maximum string length of any string in the string table.

File

A handle to a data file.

Description

A file item is created by the file folio when a file is opened. It is a private item used to maintain context information about the file being accessed.

Folio

file

Item Type

FILENODE

Create

`OpenDiskFile()`, `ChangeDirectory()`

Delete

`CloseDiskFile()`

OpenDiskFile

Opens a disk file.

Synopsis

```
Item OpenDiskFile( char *path )
```

Description

This function opens a disk file, given an absolute or relative pathname, and returns its item number.

Arguments

path

An absolute or relative pathname (a null- terminated text string) for the file to open, or an alias for the pathname.

Return Value

The function returns the item number of the opened file (which can be used later to refer to the file), or a negative error code if an error occurs.

Implementation

SWI implemented in file folio V20.

Associated Files

filefunctions.h

See Also

CloseDiskFile(), OpenDiskFileInDir(), OpenDiskStreams()

CloseDiskFile

Closes a file.

Synopsis

```
int32 CloseDiskFile( Item fileItem )
```

Description

Closes a disk file that was opened with a call to `OpenDiskFile()` or `OpenDiskFileInDir()`. The specified item may not be used after successful completion of this call.

Arguments

fileItem
: The item number of the disk file to close.

Return Value

The function returns a value greater than or equal to 0 if successful or a negative error code if an error occurs.

Implementation

SWI implemented in file folio V20.

Associated Files

filefunctions.h

See Also

`OpenDiskFile()`, `OpenDiskFileInDir()`

OpenDiskFileInDir

Opens a disk file contained in a specific directory.

Synopsis

```
Item OpenDiskFileInDir( Item dirItem, char *path )
```

Description

Similar to `OpenDiskFile()`, this function allows the caller to specify the item of a directory that should serve as a starting location for the file search.

Arguments

dirItem
: The item number of the directory containing the file.

path
: A pathname for the file that is relative to the directory specified by the dirItem argument.

Return Value

The function returns the item number of the opened file (which can be used later to refer to the file), or a negative error code if an error occurs.

Implementation

SWI implemented in file folio V20.

Associated Files

filefunctions.h

See Also

`OpenDiskFile()`, `OpenDirectoryItem()`, `OpenDirectoryPath()`, `OpenDiskStream()`, `CloseDiskFile()`

OpenDirectoryItem

Opens a directory specified by an item.

```
Directory *OpenDirectoryItem( Item openFileItem )
```

This function opens a directory. It allocates a new directory structure, opens the directory, and prepares for a traversal of the contents of the directory. Unlike `OpenDirectoryPath()`, you specify the file for the directory by its item number rather than by its pathname.

openFileItem
: The item number of the open file to use for the directory. When you later call CloseDirectory(), this file item will automatically be deleted for you, you do not need to call CloseDiskFile().

The function returns a pointer to the directory structure that is created or NULL if an error occurs.

Folio call implemented in file folio V20.

When you are done scanning the directory and call `CloseDirectory()`, the item you gave to `OpenDirectoryItem()` will automatically be closed for you. In essence, when you call `OpenDirectoryItem()`, you are giving away the File item to the file folio, which will dispose of it when the directory is closed.

directoryfunctions.h, filesystem.lib

`OpenDiskFile()`, `OpenDiskFileInDir()`, `OpenDirectoryPath()`, `CloseDirectory()`

OpenDirectoryPath

Opens a directory specified by a pathname.

Synopsis

```
Directory *OpenDirectoryPath( char *thePath )
```

Description

This function opens a directory. It allocates a new directory structure, opens the directory, and prepares for a traversal of the contents of the directory. Unlike `OpenDirectoryItem()`, you specify the file for the directory by its pathname rather than by its item number.

Arguments

thePath
: An absolute or relative pathname (a null- terminated text string) for the file to use for the directory, or an alias for the pathname.

Return Value

The function returns a pointer to a directory structure that is created or NULL if an error occurs.

Implementation

Folio call implemented in file folio V20.

Associated Files

directoryfunctions.h, filesystem.lib

See Also

`OpenDiskFile()`, `OpenDiskFileInDir()`, `OpenDirectoryItem()`, `CloseDirectory()`

CloseDirectory

Closes a directory.

Synopsis

```
void CloseDirectory( Directory *dir )
```

Description

This function closes a directory that was previously opened using `OpenDirectoryItem()` or `OpenDirectoryPath()`. All resources get released.

Arguments

dir
: A pointer to the directory structure for the directory to close.

Implementation

Folio call implemented in file folio V20.

Associated Files

directoryfunctions.h, filesystem.lib

See Also

`OpenDirectoryItem()`, `OpenDirectoryPath()`, `ReadDirectory()`

ReadDirectory

Reads the next entry from a directory.

Synopsis

```
int32 ReadDirectory( Directory *dir, DirectoryEntry *de )
```

Description

This routine reads the next entry from the specified directory. It stores the information from the directory entry into the supplied DirectoryEntry structure. You can then examine the DirectoryEntry structure for information about the entry.

The most interesting fields in the DirectoryEntry structure are:

de_FileName
: The name of the entry.

de_Flags
: This contains a series of bit flags that describe characteristics of the entry. Flags of interest are FILE_IS_DIRECTORY, which indicates the entry is a nested directory and FILE_IS_READONLY, which tells you the file cannot be written to.

de_Type
: This is currently one of FILE_TYPE_DIRECTORY, FILE_TYPE_LABEL, or FILE_TYPE_CATAPULT.

de_BlockSize
: This is the size in bytes of the blocks when reading this entry.

de_ByteCount
: The logical count of the number of useful bytes within the blocks allocated for this file.

de_BlockCount
: The number of blocks allocated for this file.

You can use `OpenDirectoryPath()` and `ReadDirectory()` to scan the list of mounted file systems. This is done by supplying a path of "/" to `OpenDirectoryPath()`. The entries that `ReadDirectory()` returns will correspond to all of the mounted file systems. You can then look at

the de_Flags field to determine if a file system is readable or not.

Arguments

dir

A pointer to the directory structure for the directory.

de

A pointer to a DirectoryEntry structure in which to receive the information about the next directory entry.

Return Value

This function returns zero if successful or a negative error code if an error (such as end-of-directory) occurs.

Implementation

Folio call implemented in file folio V20.

Associated Files

directoryfunctions.h, filesystem.lib

See Also

`OpenDirectoryItem(), OpenDirectoryPath()`

OpenDiskStream

Opens a disk stream for stream-oriented I/O.

Synopsis

```
Stream *OpenDiskStream( char *theName, int32 bSize )
```

Description

This routine allocates a new stream structure, opens the disk file identified by an absolute or relative pathname, allocates the specified amount of buffer space, and initiates an asynchronous read to fill the buffer with the first portion of the file's data. It returns NULL if any of these functions cannot be performed for any reason.

The bSize field specifies the amount of buffer space to be allocated to the file stream. It may be positive (giving the number of bytes to be allocated), zero (indicating that a default allocation of two blocks should be used), or negative (giving the two's complement of the number of blocks worth of memory that should be allocated).

Arguments

theName
: An absolute or relative pathname (a null- terminated text string) for the file to open.

bSize
: The size of the buffer to allocate for the stream. This can be (1) a positive value that specifies the size of the buffer to allocate, in bytes; (2) zero, which specifies to allocate the default buffer (currently two blocks); or (3) a negative value, the two's complement of which specifies the number of blocks of memory to allocate for the buffer.

Return Value

The function returns a pointer to the stream structure for the opened file or NULL if an error occurs.

Implementation

Folio call implemented in file folio V20.

Associated Files

filestreamfunctions.h

See Also

CloseDiskStream(), ReadDiskStream(), SeekDiskStream(), OpenDiskFile()

CloseDiskStream

Closes a disk stream.

Synopsis

```
void CloseDiskStream( Stream *theStream )
```

Description

This function closes a disk stream that was opened with a call to `OpenDiskStream()`. It closes the stream's file, deallocates the buffer memory, and releases the stream structure.

Arguments

theStream
: A pointer to the stream structure for an open file.

Implementation

Folio call implemented in file folio V20.

Associated Files

filestreamfunctions.h, filesystem.lib

See Also

`OpenDiskStream()`, `ReadDiskStream()`, `SeekDiskStream()`

ReadDiskStream

Reads from a disk stream.

Synopsis

```
int32 ReadDiskStream( Stream *theStream, char *buffer,int32 nBytes )
```

Description

This routine transfers the specified number of bytes (or, as many as are left before the end-of-file) from the stream (beginning at the stream's current position) to the specified buffer. It advances the stream position past the bytes that have been transferred, and returns the actual number of bytes transferred. It will initiate an asynchronous read to read additional data into the stream's internal buffer, if possible and appropriate.

Arguments

theStream
: A pointer to the stream structure from which to read.

buffer
: A pointer to the buffer into which to read the data.

nBytes
: The number of bytes to read.

Return Value

The function returns the actual number of bytes read or a negative error code if an error occurs.

Implementation

Folio call implemented in file folio V20.

Associated Files

filestreamfunctions.h, filesystem.lib

See Also

CloseDiskStream(), OpenDiskStream(), SeekDiskStream()

SeekDiskStream

Performs a seek operation on a disk stream.

Synopsis

```
int32 SeekDiskStream( Stream *theStream, int32 offset,enum SeekOrigin
whence )
```

Description

This function does a seek operation on a stream file, thereby changing the I/O byte position within the file. After calling this function, data transfers start with the first byte at the new file position. The whence argument specifies whether the operation is to be relative to either the beginning or end of the file or to the current position in the file. The offset argument specifies the number of bytes of offset relative to the whence position. The result is the actual absolute file position that results from the seek or an error code if anything went wrong.

The offset is specified in any of three ways: absolute (positive) byte offset from the beginning of file (SEEK_SET); relative byte offset from the current position in the file (SEEK_CUR); or absolute (negative) byte offset from the end of the file (SEEK_END).

This function doesn't actually perform the seek, or do any I/O, or invalidate the data in the stream, or abort a read-ahead in progress. It simply stores the desired offset and a "please seek" request into the stream structure. The actual seeking, if any, occurs on the next call to `ReadDiskStream()`.

Seeks are very efficient if they are backward seeks that don't take you outside of the current block, or forward seeks that don't take you outside of the amount of data actually read ahead into the buffer. Short seeks of this nature simply adjust the pointers, and then allow the read to continue from the appropriate place in the buffer.

Seeks that move you backward outside of the current block or forward past the amount of data in the buffer force all of the data in the buffer to be flushed. Data from the sought after portion of the file will be read into the buffer (synchronously) and the read will be completed. As a result, you'll see some latency when you issue the ReadDiskStream call.

If you wish to do a seek-ahead that is, you wish to seek to a specified location in the file but not necessarily read anything immediately the best thing to do is a SeekDiskStream followed immediately by a zero-byte ReadDiskStream. This is a special case that will start the read-ahead for the data you're seeking, and then return without waiting for the I/O to complete. When you issue a non-zero-length

ReadDiskStream some time later, the data will probably have been read in, and there will be no delay in accessing it.

Arguments

theStream
: A pointer to an open Stream. "offset" is a byte offset into the file, and whence is SEEK_SET, SEEK_CUR, or SEEK_END.

offset
: Specified in any of three ways: absolute (positive) byte offset from the beginning of file (SEEK_SET); relative byte offset from the current position in the file (SEEK_CUR); or absolute (negative) byte offset from the end of the file (SEEK_END).

whence
: The anchor point (either the beginning of the file, the current position, or the end of the file) to which the offset should be applied to create the new file position.

Return Value

The function returns the actual (absolute) offset to which the seek occurs or a negative error code if (1) the offset is outside of the legal range; (2) the whence field contains an illegal value; or (3) there was any other problem.

Implementation

Folio call implemented in file folio V20.

Associated Files

filestreamfunctions.h, filesystem.lib

Notes

Seeks that move backward outside the current block or forward past the amount of data in the buffer flush all of the data from the buffer. Data from the sought-after portion of the file is read synchronously into the buffer and the read operation is then completed. As a result, you'll see some latency when you call `ReadDiskStream()`.

To perform a seek-ahead (seeking to an actual physical location in the file but not necessarily read

anything immediately), call `SeekDiskStream()` followed immediately by a zero-byte `ReadDiskStream()`. This is a special case that starts the read-ahead for the data you're seeking. You then return without waiting for the I/O to complete. Later, you can call `ReadDiskStream()` with a non-zero read-length value, and if the data has been read in, you have immediate access to it.

See Also

`CloseDiskStream()`, `OpenDiskStream()`, `ReadDiskStream()`

ChangeDirectory

Changes the current directory.

Synopsis

```
Item ChangeDirectory( char *path )
```

Description

Changes the current task's working directory to the absolute or relative location specified by the path, and returns the item number for the directory.

Arguments

path
: An absolute or relative pathname for the new current directory.

Return Value

The function returns the item number of the new directory or a negative error code if an error occurs.

Implementation

SWI implemented in file folio V20.

Associated Files

filefunctions.h

See Also

GetDirectory()

GetDirectory

Gets the item number and pathname for the current directory.

Synopsis

```
Item GetDirectory( char *pathBuf, int32 pathBufLen )
```

Description

This function returns the item number of the calling task's current directory. If pathBuf is non-NULL, it must point to a buffer of writable memory whose length is given in pathBufLen; the absolute pathname of the current working directory is stored into this buffer.

Arguments

pathBuf
: A pointer to a buffer in which to receive the absolute pathname for the current directory. If you do not want to get the pathname string, use NULL as the value of this argument.

pathBufLen
: The size of the buffer pointed to by the pathBuf argument, in bytes, or zero if you don't provide a buffer.

Return Value

The function returns the item number of the current directory.

Implementation

SWI implemented in file folio V20.

Associated Files

filefunctions.h

See Also

ChangeDirectory()

File alias

A character string for referencing the pathname to a file or a directory of files.

Description

A file alias is a character string for referencing the pathname to a file or directory of files on disk. Whenever the file folio parses pathnames, it looks for path components starting with $, and treats the rest of the component at the name of an alias. The file folio then finds the alias of that name, and extracts a replacement string from the alias to use as the path component.

Folio

file

Item Type

FILEALIASNODE

Create

`CreateAlias()`

Delete

`DeleteItem()`

CreateAlias

Creates a file system alias.

Synopsis

```
Item CreateAlias( char *aliasPath, char *realPath )
```

Description

This function creates an alias for a file. The alias can be used in place of the full pathname for the file. Note that the file system maintains separate alias entries for each task.

Arguments

aliasPath
: The alias name.

realPath
: The substitution string for the alias.

Return Value

The function returns the item number for the file alias that is created, or a negative error code if an error occurs.

Implementation

SWI implemented in file folio V20.

Associated Files

filefunctions.h

See Also

OpenDiskFile(), OpenDiskFileInDir()

Folio

A family of function calls.

Description

A folio is a family of function calls available in the Portfolio operating system.

Folios provide a standard way to package groups of related functions. Portfolio is composed of a number of folios, such as the kernel folio, the graphics folio, and the compression folio. Some folios are permanent, such as the kernel folio, while other are demand-loaded, which means they only exist when they are needed.

You can create folios for your own use. It is a convenient way to share code amongst multiple independant tasks or threads.

Folio

kernel

Item Type

FOLIONODE

Create

CreateItem(), CreateNamedItem(), CreateVersionedItem()

Delete

DeleteItem()

Query

FindItem(), FindFolio(), FindNamedNode(), FindVersionedItem()

Modify

OpenItem(), CloseItem(), FindAndOpenItem(), FindAndOpenFolio()

Tags

CREATEFOLIO_TAG_NUSERVECS

(uint32) Create. This value indicates the number of elements in the function table.

CREATEFOLIO_TAG_USERFUNCS

(void *) Create. A pointer to a table of function pointers. Once the folio is created, clients of the folio will be able to call the functions within this table.

IOReq

An item used to communicate between a task and an I/O device.

Description

An IOReq item is used to carry information from a client task to a device in order to have the device perform some operation. Once the device is stores return information in the IOReq, and returns it to the client task.

Folio

kernel

Item Type

IOREQNODE

Create

CreateIOReq(), CreateItem()

Delete

DeleteIOReq(), DeleteItem()

Query

FindItem()

Modify

SetItemOwner(), SetItemPri()

Use

SendIO(), DoIO(), AbortIO(), WaitIO(), CheckIO()

Tags

CREATEIOREQ_TAG_REPLYPORT

(Item) Create. The item number of a message port.This is where the device will send a message whenever an I/O operation completes. This argument is optional. If you do not specify it, the device will send your task the `SIGF_IODONE` signal instead of a message.

CREATEIO_TAG_DEVICE

(Item) Create. This specifies the item number of the device that this IOReq will be used to communicate with. This item number is obtained by calling OpenNamedDevice() or FindAndOpenNamedDevice().

Locale

A database of international information.

Description

A Locale structure contains a number of definitions to enable a title to operate transparently accross a wide range of cultural and language environments.

Folio

international

Item Type

LOCALENODE

Use

`intlOpenLocale`(), `intlCloseLocale`(), `intlLookupLocale`()

intlOpenLocale

Gains access to a locale item.

Synopsis

```
Item intlOpenLocale(const TagArg *tags);
```

Description

This function returns a locale item. You can then use `intlLookupLocale()` to gain access to the localization information within the locale item. This information enables software to adapt to different languages and customs automatically at run-time, thus creating truly international software.

Arguments

tags
: A pointer to an array of tag arguments containing extra data for this function. This must currently always be NULL.

Return Value

The function returns the item number of a locale. You can use `intlLookupLocale()` to get a pointer to the locale structure, which contains localization information.

Implementation

Macro implemented in intl.h V24.

Associated Files

intl.h

Notes

Once you are finished with the locale item, you should call `intlCloseLocale()`.

See Also

intlCloseLocale(), intlLookupLocale()

intlCloseLocale

Terminates use of a given locale item.

Synopsis

```
Err intlCloseLocale(Item locItem);
```

Description

This function concludes the title's use of the given locale item. After this call is made, the locale item may no longer be used.

Arguments

locItem
: The locale item, as obtained from intlOpenLocale().

Return Value

> = 0
: Item successfully closed.

INTL_ERR_BADITEM
: locItem was not an existing locale item.

INTL_ERR_CANTCLOSEITEM
: At attempt was made to close this locale item more often than it was opened.

Implementation

Macro implemented in intl.h V24.

Associated Files

intl.h

See Also

intlOpenLocale(), intlLookupLocale()

intlLookupLocale

Returns a pointer to a locale structure.

Synopsis

```
Locale *intlLookupLocale(Item locItem);
```

Description

This macro returns a pointer to a locale structure. The structure can then be examined and its contents used to localize titles.

Arguments

locItem
: A locale item, as obtained from intlOpenLocale().

Return Value

The macro returns a pointer to a locale structure, which contains localization information, or NULL if the supplied Item was not a valid locale item.

Implementation

Macro implemented in intl.h V24.

Associated Files

intl.h

Example

```
{
Item    |t;
Locale *loc;
    it = intlOpenLocale(NULL);
    0
```

```
    {
        loc = intlLookupLocale(it);
        // you can now read fields in the Locale structure.
        e
        intlCloseLocale(it);
    }
}
```

See Also

intlOpenLocale(), intlCloseLocale()

Message

The means of sending data from one task to another.

Description

Messages are used to communicate amongst tasks. Messages come in three flavors: small, standard, and buffered. The flavors each carry their own type or amount of information. The small message carries exactly eight bytes of data. Standard messages contain a pointer to some data. Buffered messages contain an arbitrary amount of data within them.

Messages are closely associated with message ports. In order to send a message, a task must indicate a destination message port. Essentially, messages shuttle back and forth between message ports, and tasks can get and put messages from and to message ports.

Folio

kernel

Item Type

MESSAGENODE

Create

CreateMsg(), CreateSmallMsg(), CreateBufferedMsg(), CreateItem()

Delete

DeleteMsg(), DeleteItem()

Query

FindItem(), FindNamedItem(), FindVersionedItem()

Use

ReplyMsg(), ReplySmallMsg(), SendMsg(), SendSmallMsg(), GetThisMsg(), GetMsg(), WaitPort()

Tag Arguments

CREATEMSG_TAG_REPLYPORT

(Item) Create. The item number of the message port to use when replying this message. This must be a port that was created by the same task or thread that is creating the message.

CREATEMSG_TAG_MSG_IS_SMALL

Create. When this tag is present, it specifies that this message should be small. This means that this message should be used with `SendSmallMsg()` and `ReplySmallMsg()`, and can only pass 8 bytes of information.

CREATEMSG_TAG_DATA_SIZE

(uint32) Create. When this tag is present, it specifies that the message should be buffered. The argument of this tag specifies the size of the buffer that should be allocated.

Message port

An item through which a task receives messages.

Description

A message port is an item through which a task receives messages.

Folio

kernel

Item Type

MSGPORTNODE

Create

CreateMsgPort(), CreateItem()

Delete

DeleteMsgPort(), DeleteItem()

Query

FindMsgPort(), FindItem(), FindNamedItem(), FindVersionedItem()

Use

WaitPort(), GetMsg(), SendMsg(), SendSmallMsg(), ReplyMsg(), ReplySmallMsg()

Tags

CREATEPORT_TAG_SIGNAL
(int32) Create. Whenever a message arrives at a message port, the kernel sends a signal to the

owner of the message port. This tag lets you specify which signal should be sent. If this tag is not provided, the kernel will allocate a signal for the message port itself.

CREATEPORT_TAG_USERDATA

(void *) Create. Lets you specify the 32-bit value that gets put into the mp_UserData field of the MsgPort structure. This can be anything you want, and is sometimes useful to idenify a message port.

Sample

A digital recording of a sound.

Description

A Sample Item is a handle to a digital recording of a sound in memory. Samples come in two kinds:

- Ordinary Samples - sample loaded from a sample file, or created from scratch. User memory is used for samples. `CreateSample()` and a myriad of special purpose functions create ordinary samples.
- Delay Lines - special samples suitable for receiving the DMA output from delay instruments (e.g. delaymono.dsp). The memory for delay lines is allocated in system memory. Use `CreateDelayLine()` to create a delay line.

Most sample operations can be performed on both kinds of samples.

Folio

audio

Item Type

AUDIO_SAMPLE_NODE

Create

`CreateDelayLine()`

`CreateItem()`

`CreateSample()`

`DefineSampleHere()`

`LoadSample()`

LoadSampleHere()

ScanSample()

Delete

DeleteDelayLine()

DeleteItem()

UnloadSample()

Query

GetAudioItemInfo()

Modify

SetAudioItemInfo()

Use

AttachSample()

DebugSample()

Tags

Loading:

AF_TAG_NAME
(const char *) Create. Name of sample file to load.

AF_TAG_IMAGE_ADDRESS
(const char *) Create. Specifies a memory location containing a sample file image from which to read sample. Must use in conjunction with AF_TAG_IMAGE_LENGTH.

AF_TAG_IMAGE_LENGTH

(uint32) Create. Specifies number of bytes in sample file image pointed to by AF_TAG_IMAGE_ADDRESS.

AF_TAG_SCAN

(int32) Create. Specifies the maximum number of bytes of sound data to read from the file. This can be used to cause the Audio folio to simply load sample parameters without loading sample data.

AF_TAG_ALLOC_FUNCTION

(void *(*)(uint32 memsize, uint32 memflags)) Create. Sets custom memory allocation function to be called to allocate sample memory while loading sample file. Defaults to `AllocMem()`. If you supply a custom allocation function you must also provide a custom free function with AF_TAG_FREE_FUNCTION.

AF_TAG_FREE_FUNCTION

(void (*)(void *memptr, uint32 memsize)) Create. Sets custom memory free function to free sample memory to be called during sample deletion. Defaults to `FreeMem()`. If you supply a custom free function you must also provide a custom allocation function with AF_TAG_ALLOC_FUNCTION.

AF_TAG_LEAVE_IN_PLACE

(bool) When TRUE, causes sample being read from AF_TAG_IMAGE_ADDRESS to be used in-place instead of allocating more memory to hold the sample data. Mutually exclusive with AF_TAG_ALLOC_FUNCTION. See `CreateSample()` for more details and caveats.

Note that sample data may have to be moved down two bytes to align it properly for Opera DMA. This will destroy the sample image so that it cannot be reused.

AF_TAG_DATA_OFFSET

(uint32) Query. Byte offset in sample file of the beginning of the sample data.

AF_TAG_DATA_SIZE

(uint32) Query. Size of sample data in sample file in bytes. Note that this may differ from the length of the sample as loaded into memory (as returned by AF_TAG_NUMBYTES).

Data:

AF_TAG_ADDRESS

(void *) Create, Query, Modify. Address of sample data.

This tag, and the all the other Data tags, can be used to query or modify the data address/length of ordinary samples. They can only be used to query the address/length of a delay line.

AF_TAG_FRAMES
(uint32) Create, Query, Modify. Length of sample data expressed in frames. In a stereo sample, this would be the number of stereo pairs.

AF_TAG_NUMBYTES
(uint32) Create, Query, Modify. Length of sample data expressed in bytes.

Format:

AF_TAG_CHANNELS
(uint32) Create, Query, Modify. Number of channels (or samples per sample frame). For example: 1 for mono, 2 for stereo. Valid range is 1..255.

AF_TAG_WIDTH
(uint32) Create, Query, Modify. Number of bytes per sample (uncompressed). Valid range is 1..2.

AF_TAG_NUMBITS
(uint32) Create, Query, Modify. Number of bits per sample (uncompressed). Valid range is 1..32. Width is rounded up to the next byte when computed from this tag.

AF_TAG_COMPRESSIONTYPE
(uint32) Create, Query, Modify. 32-bit ID representing AIFC compression type of sample data (e.g. ID_SDX2). 0 for no compression.

AF_TAG_COMPRESSIONRATIO
(uint32) Create, Query, Modify. Compression ratio of sample data. Uncompressed data has a value of 1. 0 is illegal.

Loops:

AF_TAG_SUSTAINBEGIN
(int32) Create, Query, Modify. Frame index of the first frame of the sustain loop. Valid range is 0..NumFrames-1. -1 for no sustain loop. Use in conjunction with AF_TAG_SUSTAINEND.

AF_TAG_SUSTAINEND
(int32) Create, Query, Modify. Frame index of the first frame after the last frame in the sustain loop. Valid range is 1..NumFrames. -1 for no sustain loop. Use in conjunction with AF_TAG_SUSTAINBEGIN.

AF_TAG_RELEASEBEGIN
(int32) Create, Query, Modify. Frame index of the first frame of the release loop. Valid range is

0..NumFrames-1. -1 for no release loop. Use in conjunction with AF_TAG_RELEASEEND.

AF_TAG_RELEASEEND
(int32) Create, Query, Modify. Frame index of the first frame after the last frame in the release loop. Valid range is 1..NumFrames. -1 for no release loop. Use in conjunction with AF_TAG_RELEASEBEGIN.

Tuning:

AF_TAG_BASENOTE
(uint32) Create, Query, Modify. MIDI note number for this sample when played at the original sample rate (as set by AF_TAG_SAMPLE_RATE). Defaults to middle C (60). This defines the frequency conversion reference note for the `StartInstrument()` AF_TAG_PITCH tag.

AF_TAG_DETUNE
(int32) Create, Query, Modify. Amount to detune the MIDI base note in cents to reach the original pitch. Must be in the range of -100 to 100.

AF_TAG_SAMPLE_RATE
(ufrac16) Create, Query, Modify. Original sample rate for sample expressed in 16.16 fractional Hz. Defaults to 44,100 Hz.

AF_TAG_BASEFREQ
(ufrac16) Query. The frequency of the sample, expressed in 16.16 Hz, when played at the original sample rate. This value is computed from the other tuning tag values.

Multisample:

AF_TAG_LOWNOTE
(uint32) Create, Query, Modify. Lowest MIDI note number at which to play this sample when part of a multisample. `StartInstrument()` AF_TAG_PITCH tag is used to perform selection. Valid range is 0 to 127. Defaults to 0.

AF_TAG_HIGHNOTE
(uint32) Create, Query, Modify. Highest MIDI note number at which to play this sample when part of a multisample. Valid range is 0 to 127. Defaults to 127.

Notes

Sample creation tags have a lot mutual interaction. See `CreateSample()` for a complete explanation of this.

Caveats

There's currently no way to enforce that memory pointed to by AF_TAG_ADDRESS or file image memory used with AF_TAG_LEAVE_IN_PLACE is actually of MEMTYPE_AUDIO. Be careful.

All sample data, loop points, and lengths must be quad-byte aligned. For example, a 1-channel, 8-bit sample (which has 1 byte per frame) is only legal when the lengths and loop points are at multiples of 4 frames. For mono ADPCM samples (which have only 4 bits per frame), lengths and loop points must be at multiples of 8 frames. The Audio folio, however, does not report any kind of an error for using non-quad-byte aligned sample data. Sample addresses, lengths, and loop points, are internally truncated to quad-byte alignment when performing the DMA without warning. Misalignments may result in noise at loop points, or slight popping at the beginning and ending of sound playback. It is recommended that you pay very close attention to sample lengths and loop points when creating, converting, and compressing samples.

Setting AF_TAG_WIDTH does not set the AF_TAG_NUMBITS attribute (e.g. if you create a sample with AF_TAG_WIDTH set to 1, `GetAudioItemInfo()` AF_TAG_NUMBITS will return 16). Setting AF_TAG_NUMBITS does however correctly update the AF_TAG_WIDTH field.

See Also

`Attachment`, `Instrument`, `Template`, `StartInstrument()`, `sampler.dsp`, `delaymono.dsp`

ScreenGroup

A collection of one or more screens.

Description

A screen group is a collection of one or more screens.

Folio

graphics

Item Type

SCREENGROUPNODE

Create

CreateScreenGroup()

Delete

DeleteScreenGroup()

Use

AddScreenGroup(), RemoveScreenGroup()

Tags

CSG_TAG_DISPLAYHEIGHT
: (uint32) Create. This optional tag lets you specify the visible height of each screen within the screen group. This determines the number of lines of the bitmaps that make up the screen will be visible at any one time. When this

CSG_TAG_SCREENCOUNT
: (uint32) Create. This optional tag lets you specify the number of screens to create within the screen group. If the tag is not supplied, then two screens is created.

CSG_TAG_SCREENHEIGHT

(uint32) Create. This optional tag lets you specify the number of lines that make up each screen within the screen group. When this tag is not supplied, the current default height is used.

CSG_TAG_BITMAPCOUNT

(uint32) Create. This optional tag lets you specify the number of bitmaps per screen. This tag should currently not be used.

CSG_TAG_BITMAPWIDTH_ARRAY

(uint32 *) Create. This optional tag provides a pointer to an array of width values, indicating the width of each bitmap.

CSG_TAG_BITMAPHEIGHT_ARRAY

(uint32 *) Create. This optional tag provides a pointer to an array of height values, indicating the height of each bitmap.

CSG_TAG_BITMAPBUF_ARRAY

(void **) Create. This optional tag points to an array of pointers to memory buffers. This array must contain as many entries as the screen group has screens. The array points to the memory to use for bitmaps when creating the screen group. If this tag is not provided, the graphics folio will allocate memory for the bitmaps itself.

CSG_TAG_VDLTYPE

(uint32) Create. This optional tag specifies the type of VDL for each screen.

CSG_TAG_VDLPTR_ARRAY

(uint32 **) Create. This tag maps the bitmaps to the video display lists. Each element of the array is a pointer to a VDL.

CSG_TAG_VDLLENGTH_ARRAY

(uint32 *) Create. This optional tag specifies the length in words for each custom VDL referenced by the CSG_TAG_VDLPTR_ARRAY.

CSG_TAG_SPORTBITS

(uint32) Create. Specifies the memory allocation bits to use when allocating memory for the screen group, which lets you control from which bank of VRAM an allocation comes from. If this tag is not supplied, the graphics folio will allocate the memory from any bank of VRAM. The values you supply to this tag are either `(MEMTYPE_BANKSELECT | MEMTYPE_BANK1)` or `(MEMTYPE_BANKSELECT | MEMTYPE_BANK2).`

Semaphore

An item used to arbitrate access to shared resources.

Description

Semaphores are used to protect shared resources. Before accessing a shared resource, a task must first try to lock the associated semaphore. Only one task at a time can have the semaphore locked at any one moment. This prevents multiple tasks from accessing the same data at the same time.

Folio

kernel

Item Type

SEMA4NODE

Create

`CreateSemaphore()`, `CreateItem()`

Delete

`DeleteSemaphore()`, `DeleteItem()`

Query

`FindSemaphore()`

Use

`LockSemaphore()`, `UnlockSemaphore()`, `LockItem()`, `UnlockItem()`

Task

A process.

Description

A task item contains all the information needed by the kernel to support multitasking. It contains room to store CPU registers when the associated task context goes to sleep, it contains pointers to various resources used by the task, and it specifies the task's priority. There is one task item for every task or thread that exists.

Folio

kernel

Item Type

TASKNODE

Create Call

CreateThread(), ExecuteAsThread(), LoadProgram(), CreateItem()

Delete

DeleteThread(), DeleteItem()

Query

FindTask(), FindItem()

Modify

SetItemOwner(), SetItemPri()

Tags

CREATETASK_TAG_MAXQ-a value indicating the maximum quanta for the task.

CREATETASK_TAG_PC-initial PC.

CREATETASK_TAG_STACKSIZE-the size of a task's call stack.

CREATETASK_TAG_ARGC-the initial argc value passed to the task when it starts up.

CREATETASK_TAG_ARGP-the initial argp value passed to the task when it starts up.

CREATETASK_TAG_SP-initial stack pointer value (thread only).

CREATETASK_TAG_BASE-the initial r9 value.

TAG_ITEM_NAME
(char *) Create. Provide a pointer to the name of the task.

TAG_ITEM_PRI
(uint8) Create. Provide a priority for the taskin the range 10 to 199. If this tag is not given, the task will inherit the priority of the current context.

CREATETASK_TAG_PC
(void *) Create. Provide a pointer to the code to be executed.

CREATETASK_TAG_SP
(void *) Create. Provide a pointer to the memory buffer to use as stack for a thread. When this tag is present, a thread is created. If the tag is not present, then a task is created.

CREATETASK_TAG_STACKSIZE
(uint32) Create. Specifies the size in bytes of the memory buffer reserved for a thread's stack.

CREATETASK_TAG_ARGC
(uint32) Create. A 32-bit value that will be passed to the task or thread being launched as a first argument. If this is omitted, the first argument will be 0.

CREATETASK_TAG_ARGP
(void *) Create. A 32-bit value that will be passed to the task or thread being launched as a second argument. If this is omitted, the second argument will be 0.

CREATETASK_TAG_USERONLY
Create. Specifies that a thread being launched will not make any folio calls. This will be the case

for some threads which are used as pure computation engines. Specifying this tag causes the thread to use fewer system resources (no supervisor stack is allocated), but prevents the task from making any calls to a folio.

CREATETASK_TAG_MSGFROMCHILD

(Item) Create. Provides the item number of a message port. The kernel will send a status message to this port whenever the thread or task being created exits. The message is sent by the kernel after the task has been deleted. The `msg_Result` field of the message contains the exit status of the task. This is the value the task provided to `exit()`, or the value returned by the task's primary function. The `msg_DataPtr` field of the message contains the item number of the task that just terminated. Finally, the `msg_DataSize` field contains the item number of the thread or task that terminated the task. If the task exited on its own, this will be the item number of the task itself. It is the responsibility of the task that receives the status message to delete it when it is no longer needed by using `DeleteMsg()`.

CREATETASK_TAG_ALLOCDTHREADSP

Create. When this tag is supplied, it tells the kernel that if this thread dies by itself, by either returning to the kernel or by calling `exit()`, then the memory used for its stack must be freed automatically. When this tag is not provided, you are responsible for freeing the stack whenever the thread terminates.

ExecuteAsThread

Executes previously loaded code as a thread.

Synopsis

```
Item ExecuteAsThread(CodeHandle code,int32 argc, char **argv,char
*threadName, int32 priority);
```

Description

This function lets you execute a chunk of code that was previously loaded from disk using `LoadCode()`. The code will execute as a thread of the current task.

In order to function correctly, code being run as a thread should be linked with threadstartup.o instead of the usual cstartup.o.

The argc and argv parameters are passed directly to the `main()` entry point of the loaded code.

The values you supply for argc and argv are irrelevant to this function. They are simply passed through to the loaded code. Therefore, their meaning must be agreed upon by the caller of this function and by the loaded code.

threadName specifies the name of the thread.

priority specifies the priority the new thread should have. Providing a negative priority makes the thread inherit the priority of the current task or thread.

Arguments

code
: A code handle filled in by a previous call to LoadCode().

argc
: A value that is passed directly as the argc parameter to the loaded code's main() entry point. This function doesn't use the value of this argument, it is simply passed through to the loaded code.

argv
: A value that is passed directly as the argv parameter to the loaded code's main() entry point. This

function doesn't use the value of this argument; it is simply passed through to the loaded code.

threadName
: The name of the thread. Should be a descriptive string that identifies this thread.

priority
: The priority for the new thread. Supply a negative value to have the thread inherit the priority of the current task or thread. Priorities for user threads can be in the range of 10 to 199.

Return Value

Returns the item number of the new thread, or a negative error code if the thread could not be created.

Implementation

Folio call implemented in file folio V21.

Associated Files

filefunctions.h

See Also

`LoadCode()`, `UnloadCode()`, `ExecuteAsSubroutine()`

LoadCode

Loads a binary image into memory, and obtains a handle to it.

Synopsis

```
Err LoadCode(char *fileName, CodeHandle *code);
```

Description

This function loads an executable file from disk into memory. Once loaded, the code can be spawned as a thread, or executed as a subroutine.

In order to work correctly with this and associated functions, the executable file being loaded must have been linked with threadstartup.o or subroutinestartup.o instead of the usual cstartup.o.

Give this function the name of the executable file to load, as well as a pointer to a CodeHandle variable, where the handle for the loaded code will be stored. Note, "code" must point to a valid CodeHandle variable; that's where `LoadCode()` will put a pointer to the loaded code.

Once you finish using the loaded code, you can remove it from memory by passing the code handle to the `UnloadCode()` function.

To execute the loaded code, you must call either the `ExecuteAsThread()` function or the `ExecuteAsSubroutine()`function. Note that if the loaded code is reentrant, the same loaded code can be spawned multiple times simultaneously as a thread.

Arguments

fileName
: A NULL-terminated string indicating the executable file to load.

code
: A pointer to a CodeHandle variable, where a handle to the loaded code can be put.

Return Value

Returns > = 0 if successful, or a negative error code it the file could not be loaded.

Implementation

Folio call implemented in file folio V21.

Associated Files

filefunctions.h

See Also

LoadProgram(), UnloadCode(), ExecuteAsThread(), ExecuteAsSubroutine()

LoadProgram

Loads a binary image and spawns it as a task.

Synopsis

```
Item LoadProgram(char *cmdLine);
```

Description

This function loads an executable file from disk and launches a new task to execute the code.

You give this function a command line to interpret. The first component of the command line is taken as the name of the file to load. The entirety of the command line is passed to the new task as argc and argv in the `main()` function. The filename component of the command line specifies either a fully qualified pathname, or a pathname relative to the current directory.

The priority of the new task is identical to the priority of the current task. If you wish the task to have a different priority, you must use the `LoadProgramPrio()` function instead.

If a priority was given to the executable being launched using the modbin utility, then the priority that was specified to modbin will be used for the new task, and not the current priority.

Arguments

cmdLine
: A NULL-terminated string indicating the file to load as a task. The string may also contain arguments for the task being started.

Return Value

The Item number of the newly created task, or a negative error code if the task could not be created.

Implementation

Folio call implemented in file folio V20.

Associated Files

filefunctions.h

See Also

LoadProgramPrio(), LoadCode(), ExecuteAsThread()

LoadProgramPrio

Loads a binary image and spawns it as a task, with priority.

Synopsis

```
Item LoadProgramPrio(char *cmdLine, int32 priority);
```

Description

This function loads an executable file from disk and launches a new task to execute the code.

You give this function a command line to interpret. The first component of the command line is taken as the name of the file to load. The entirety of the command line is passed to the new task as argc and argv in the `main()` function. The filename component of the command line specifies either a fully qualified pathname, or a pathname relative to the current directory.

The priority argument specifies the task priority to use for the new task. If you simply want the new task to have the same priority as the current task, use the `LoadProgram()` function instead. Alternatively, passing a negative priority to this function will also give the new task the same priority as the current task.

If a priority was given to the executable being launched using the modbin utility, then the priority you give to this function is ignored, and the priority that was specified to modbin will be used instead.

Arguments

cmdLine
: A NULL-terminated string indicating the file to load as a task. The string may also contain arguments for the task being started.

priority
: The task priority for the new task. For user code, this can be in the range 10 to 199.

Return Value

The item number of the newly created task, or a negative error code if the task could not be created.

Implementation

Folio call implemented in file folio V20.

Associated Files

filefunctions.h

See Also

LoadProgram(), LoadCode(), ExecuteAsThread()

UnloadCode

Unloads a binary image previously loaded with LoadCode().

Synopsis

```
Err UnloadCode(CodeHandle code);
```

Description

This function frees any resources allocated by `LoadCode()`. Once `UnloadCode()` has been called, the code handle supplied becomes invalid and cannot be used again.

Arguments

code
: A code handle filled in by a previous call to LoadCode().

Return Value

Returns > = 0 if successful, or a negative error code of the CodeHandle supplied is somehow invalid.

Implementation

Folio call Implemented in file folio V21.

Associated Files

filefunctions.h

See Also

`LoadCode()`

ExecuteAsSubroutine

Executes previously loaded code as a subroutine.

Synopsis

```
int32 ExecuteAsSubroutine(CodeHandle code,int32 argc, char **argv);
```

Description

This function lets you execute a chunk of code that was previously loaded from disk using `LoadCode()`. The code will run as a subroutine of the current task or thread.

In order to function correctly, code being run as a subroutine should be linked with subroutinestartup.o instead of the usual cstartup.o.

The argc and argv parameters are passed directly to the `main()` entry point of the loaded code. The return value of this function is the value returned by `main()` of the code being run.

The values you supply for argc and argv are irrelevant to this function. They are simply passed through to the loaded code. Therefore, their meaning must be agreed upon by the caller of this function, and by the loaded code.

Arguments

code
: A code handle filled in by a previous call to LoadCode().

argc
: A value that is passed directly as the argc parameter to the loaded code's main() entry point. This function doesn't use the value of this argument, it is simply passed through to the loaded code.

argv
: A value that is passed directly as the argv parameter to the loaded code's main() entry point. This function doesn't use the value of this argument, it is simply passed through to the loaded code.

Return Value

Returns the value that the loaded code's `main()` function returns.

Implementation

Folio call implemented in file folio V21.

Caveats

If code being executed as a subroutine calls `exit()`, the parent thread or task will exit, not just the subroutine code.

Associated Files

filefunctions.h

See Also

`LoadCode()`, `UnloadCode()`, `ExecuteAsThread()`

Tuning

An item that contains information for tuning an instrument.

Description

A tuning item contains information for converting MIDI pitch numbers to frequency for an instrument or template. The information includes an array of frequencies, and 32-bit integer values indicating the number of pitches, notes in an octave, and the lowest pitch for the tuning frequency. See the CreateTuning() call in the "Audio folio Calls" chapter in this manual for more information.

Folio

audio

Item Type

AUDIO_TUNING_NODE

Create

CreateItem()

CreateTuning()

Delete

DeleteItem()

DeleteTuning()

Query

None

Modify

SetAudioItemInfo()

Use

TuneInsTemplate()

TuneInstrument()

Tags

AF_TAG_ADDRESS
(const ufrac16 *) Create, Modify. Array of frequencies in 16.16 Hz to be used as a lookup table.

AF_TAG_BASENOTE
(uint32) Create, Modify. MIDI note number that should be given the first frequency in the frequency array in the range of 0..127.

AF_TAG_FRAMES
(uint32) Create, Modify. Number of frequencies in array pointed to by AF_TAG_ADDRESS. This value must be >= NotesPerOctave.

AF_TAG_NOTESPEROCTAVE
(uint32) Create, Modify. Number of notes per octave. This is used to determine where in the frequency array to look for notes that fall outside the range of BaseNote..BaseNote+Frames-1. This value must be <= Frames.

VDL

A display control object.

Description

A VDL is a collection of hardware control values which are used to manipulate the display characteristics in strange and bizarre ways. VDLs tell the display hardware where the frame buffer is, its geometry, what colors to use, and other characteristics.

Because of their dependency on specific underlying hardware, whose quirks may change from manufacturer to manufacturer, the use of custom VDLs is discouraged at this time.

Folio

graphics

Item Type

VDLNODE

Create

`CreateVDL()`

`CreateVDLVA()`

`CreateItem()`

`SubmitVDL()`

Delete

`DeleteItem()`

`DeleteVDL()`

`DeleteItem()`

Query

None

Modify

`ModifyVDL()`

`ModifyVDLVA()`

Use

`SetVDL()`

Tags

The following tags are only valid when creating a VDL item:

CREATEVDL_TAG_VDLTYPE(int32) The type of VDL you wish to create. The currently supported types are:

- VDLTYPE_FULL: Contains complete description of all display characteristics.
- CREATEVDL_TAG_DISPLAYTYPE(int32) The type of display for which this VDL is intended. This is one of the DI_TYPE_ values in graphics.h.
- CREATEVDL_TAG_LENGTH(int32) The length of the submitted VDL, in 32-bit words.
- CREATEVDL_TAG_HEIGHT(int32) The number of display lines this VDL will affect.

CREATEVDL_TAG_DATAPTR(void *) Pointer to array of VDL instructions you wish to be validated and converted to a VDL item.

The following tags may be used to create or modify a VDL item:

CREATEVDL_TAG_SLIPSTREAM(Boolean)
: Enables/disables slipstream (non-zero == enable). When enabled, all pixels in the display buffer with a value of 000 are made transparent.

CREATEVDL_TAG_HAVG(Boolean)
Enables/disables horizontal interpolation (non-zero == enable).

CREATEVDL_TAG_VAVG(Boolean)
Enables/disables vertical interpolation (non-zero == enable).

CREATEVDL_TAG_HSUB(int32)
Sets the source for horizontal subpositioning bits as follows: 0:Force to zero. 1:Force to one. 2:Take from display buffer.

CREATEVDL_TAG_VSUB(int32)
Sets the source for vertical subpositioning bits as follows: 0:Force to zero. 1:Force to one. 2:Take from display buffer.

CREATEVDL_TAG_SWAPHV(Boolean)
Enables/disables swapping of the horizontal and vertical subpositioning bits: 0:Bit 0 == HSub; bit 15 == VSub (default) non-0:Bit 0 == VSub; bit 15 == HSub

CREATEVDL_TAG_CLUTBYPASS(Boolean)
Enables/disables the availability of CLUT bypass mode for pixels where bit 15 is set (non-zero == enable).

CREATEVDL_TAG_WINHAVG(Boolean)
Enables/disables horizontal interpolation for pixels making use of CLUT bypass mode (non-zero == enable).

CREATEVDL_TAG_WINVAVG(Boolean)
Enables/disables vertical interpolation for pixels making use of CLUT bypass mode (non-zero == enable).

CREATEVDL_TAG_WINHSUB(int32)
Sets the source for horizontal subpositioning bits for pixels making use of CLUT bypass mode as follows: 0:Force to zero. 1:Force to one. 2:Take from display buffer.

CREATEVDL_TAG_WINVSUB(int32)
Sets the source for vertical subpositioning bits for pixels making use of CLUT bypass mode as follows: 0:Force to zero. 1:Force to one. 2:Take from display buffer.

CREATEVDL_TAG_WINSWAPHV(Boolean)
Enables/disables swapping of the horizontal and vertical subpositioning bits for pixels making use of CLUT bypass mode: 0:Bit 0 == HSub; bit 15 == VSub (default) non-0:Bit 0 == VSub; bit 15 == HSub

Using an Item

Once an item is created or opened, you can manage it with the kernel item calls, which are described here.

Finding an Item

To find an item by specifying its item type and a set of tag arguments or a name, use these calls.

```
Item FindItem( int32 cType, const TagArg *tp )
Item FindItemVA( int32 cType, uint32 tags, ...)
Item FindNamedItem( int32 cType, const char *name )
Item FindVersionedItem( int32 cType, const char *name, uint8
vers, uint8 rev )
```

`FindItem()` accepts an item type value, which is created by `MkNodeID()` as it is for the `CreateItem()` call. This value specifies the type of the item for which the call should search. The second argument, `tp`, is a pointer to an array of tag arguments the call should try to match. When the call is executed, it looks through the list of items and, if it finds an item of the specified item type whose tag arguments match those set in the tag argument array, it returns the item number of that item.

`FindNamedItem()` calls `FindItem()` after creating a `TagArg` for the name. When it searches through the item list, it checks names of all items of the specified type. If it finds a match, it returns the item number of that item.

`FindVersionedItem()` accepts `cType` as its first argument, which is the same type of value as `itemType` in the other find calls. It also accepts a name, a version number, and a revision number. It searches to find the item that matches these specifications, then returns the item number of that item.

All of the item finding calls return a negative number if an error occurred in the search.

Getting a Pointer to an Item

To find the absolute address of an item, use this call:

```
void *LookupItem( Item i )
```

`LookupItem()` takes the item number of the item to locate as its only argument. If it finds the item, it

returns a pointer to the item's ItemNode structure; if it doesn't find the item, it returns NULL. Once you have a pointer to the item, you can use it to read the values in the fields of the data structure. The ItemNode structure is protected by the system, so a user task can't write to it.

Checking If an Item Exists

To see if an item exists, use this call:

```
void *CheckItem( Item i, uint8 ftype, uint8 ntype )
```

`CheckItem()` accepts the item number of the item you want to check, followed by the folio type and the node type of the item. (`ftype` and `ntype` are the same as the arguments accepted by `MkNodeID()`, and are supplied with constants defined in appropriate header files.) When executed, the call returns a NULL if it can't find an item with the specified item number, or if it found an item but it didn't have the specified folio and node types. If the call finds an item that matches in all respects, it returns a pointer to that item.

Changing Item Ownership

To change the ownership of an item from one task to another, use this call:

```
Err SetItemOwner( Item i, Item newOwner )
```

The first argument of `SetItemOwner()` is the item number of the item for which to change ownership; its second argument is the item number of the task that is to be the new owner. The task that makes this call must be the owner of the item. If ownership transfer fails, the call returns a negative number (an error code).

When ownership of an item changes, the item is removed from the former owner's resource table and placed in the new owner's resource table. This allows an item to remain in the system after the original owner dies; all items owned by a task are removed from the system when the task dies.

Changing an Item's Priority

To change the priority of an item, a task should call:

```
int32 SetItemPri( Item i, uint8 newpri )
```

Item priority determines how frequently system resources are allocated to an item. The higher its priority,

the more often an item has access to the resources. The task must be the owner of the item to change its priority.

`SetItemPri()` accepts the item number of the item whose priority you want to change, followed by an unsigned 8-bit integer which specifies the new priority for the item. Priority values range from 0-255. The call returns the former priority value of the item if successful, otherwise it returns a negative number.

Deleting an Item

When a task finishes with an item it owns, it should delete the item from the system to free resources for other uses. The task can use this call:

```
Err DeleteItem( Item i )
```

`DeleteItem()` accepts the item number of the item to delete. It returns 0 if successful, or an error code (a negative value) if an error occurs.

Only use `DeleteItem()` to delete items created with `CreateItem()`. If you've used a convenience call to create an item, you must use the corresponding deletion routine. A task must own an item to delete it.

Once an item is deleted, the kernel does not reuse the item number when it creates new items, to maintain the integrity of items. When a task tries to use an item number for a deleted item, the kernel informs the task that the item no longer exists.

Opening an Item

When a task uses a system- supplied item such as a device or a folio, it opens the item instead of creating it. To open an item, use these calls:

```
Item OpenItem( Item FoundItem, void *args )
Item FindAndOpenItem( int32 cType, const TagArg *tags)
Item FindAndOpenItemVA( int32 cType, uint32 tags, ...)
```

`OpenItem()` accepts the item number of the resource to be opened (which is found with `FindItem()`) and a pointer. Currently, NULL should always be passed in for this argument.

`FindAndOpenItem()` finds an item and opens it with one call. It works just like `FindItem()`, except that it automatically opens the item before returning it to you.

Closing an Item

When a task finishes using an open item, it should close it so the kernel knows the item is no longer required by that task. When no tasks require an opened item, the kernel can remove the item from memory, freeing system resources.

To close an item, use this call:

```
Err CloseItem( Item i )
```

`CloseItem()` accepts the item number of the item to close. It returns 0 if successful or an error code (a negative value) if an error occurs.

Function Calls

The following kernel function calls control items. See Kernel Folio Calls, in the *3DO System Programmer's Reference* for a complete description of these calls.

Creating Items

The following calls create items:

- `CreateBufferedMsg()` Creates a buffered message.
- `CreateIOReq()` Creates an I/O request.
- `CreateItem()` Creates an item.
- `CreateMsg()` Creates a standard message.
- `CreateMsgPort()` Creates a message port.
- `CreateSemaphore()` Creates a semaphore.
- `CreateSmallMsg)` Creates a small message.
- `CreateThread()` Creates a thread.
- `CreateUniqueMsgPort()` Creates a message port with a guaranteed unique name.
- `CreateUniqueSemaphore()` Creates a semaphore with a guaranteed unique name.

Opening Items

The following calls open items:

- `FindAndOpenDevice()` Finds and opens a device item.
- `FindAndOpenFolio()` Finds and opens a folio item.
- `FindAndOpenItem()` Finds and opens an item.
- `FindAndOpenNamedItem()` Finds an item by name and opens it.
- `OpenItem()` Opens a system item.
- `OpenNamedDevice()` Opens the named device.

Managing Items

The following calls manage items:

- `CheckItem()` Checks to see if an item exists.
- `FindDevice()` Finds a device item by name.
- `FindFolio()` Finds a folio item by name.
- `FindItem()` Finds an item by type and tag arguments.
- `FindMsgPort()` Finds a message port item by name.
- `FindNamedItem()`Finds an item by name.
- `FindSemaphore()` Finds a semaphore item by name.
- `FindTask()`Finds a task item by name.
- `FindVersionedItem()`Finds an item by name and version number.
- `LookupItem()`Gets a pointer to an item.
- `SetItemOwner()`Changes the owner of an item.
- `SetItemPri()`Changes the priority of an item.

Closing and Deleting Items

The following calls close and delete items:

- `CloseItem()` Closes a system item.
- `CloseNamedDevice()` Closes a device item.
- `DeleteIOReq()` Deletes an IOReq item.
- `DeleteItem()` Deletes an item.
- `DeleteMsg()`Deletes a message item.
- `DeleteMsgPort()`Deletes a message port item.
- `DeleteSemaphore()` Deletes a semaphore item.
- `DeleteThread()`Deletes a thread.

Controlling the State of a Task

A task can be in one of three states at any moment:

- **Running**, in which the task is the currently executing task.
- **Ready to run**, in which the task is in the ready queue, awaiting execution.
- **Waiting**, in which the task is in the wait queue, awaiting an external event to occur, so it becomes the running task, or if the priority of the current running task is greater than its own, it moves to the ready queue where it awaits execution.

Each non-privileged task has a priority that ranges from the lowest priority of 10 to the highest priority of 199. Priority determines the order of task execution for tasks that are in the ready queue. The kernel only executes the tasks in the ready queue with the highest-priority values.

The state of a task is determined partly by its own priority, and partly by the other tasks in the system and the interactions among them. A task can wait for other tasks to complete certain processing, at which point it resumes its own processing. Intertask communication is essential because it enables one task to notify another task on the waiting queue that an external event is complete.

If there are no external events, preemptive multitasking only takes place when tasks of equal priority are the highest priority tasks in the ready queue. If only one task has the highest-priority, only that task runs, and all other tasks wait until the task finishes or is replaced by another task with higher priority.

Yielding the CPU to Another Task

A task that executes a `Yield()` call cedes its CPU time immediately to another task with the same priority. The `Yield()` call, which neither takes arguments nor returns anything, is:

```
void Yield( void )
```

The next task in the ready queue with equal priority takes the place of the task that executes the `Yield()`. If there is no other task in the ready queue with equal priority, then the task that executes the `Yield()` call immediately resumes processing.

Going to and From the Waiting Queue

A task can place itself in the waiting queue with a wait function call, where it uses no CPU time waiting for an event to occur. A wait call defines the event or condition for which the task waits. When the condition occurs, the task moves to the ready queue and resumes running based on its priority. The basic wait call is:

```
int32 WaitSignal( uint32 sigMask )
```

Many other functions in Portfolio can make your task wait. Internally, all of these other functions eventually end up calling `WaitSignal()` to put your task to sleep.

`WaitSignal()` is the basic wait call that a task uses to wait for a signal. The wait calls `WaitIO()` and `WaitPort()` are built on `WaitSignal()`. Each of these calls puts a task on the waiting queue to await a signal, an I/O request return, or a message. Once the signal, I/O request, or message is received, the task returns to the ready queue.

Changing a Task's Parent

Normally, a task created by another task ends when the parent task ends. However, by passing ownership of the child task to another parent task, the child task can live on after the parent task ends. To change ownership, use this call:

```
Err SetItemOwner( Item i, Item newOwner )
```

The `SetItemOwner()` call takes two arguments, the item number of the task whose ownership you wish to change (`Item`), and the item number of the task that is to become the new parent (`newOwner`). This routine returns an error code if an error occurs.

A task must be the parent of a child task to change its ownership. If this call is successful, the child task's TCB is removed from the current task's resource table and placed in the parent task's resource table.

Changing Task Priorities

When a task first runs, its priority is set based on a field defined in its TCB data structure. You can change the priority (in the range of 10 through 199) of a task after it has been created by using this call:

```
int32 SetItemPri( Item i, uint8 newpri )
```

The call changes the priority of an existing task. The first argument, `i`, is the item number of the task whose priority you want to change. The second argument, `newpri`, is the new priority value that you want the task to have. The call returns the old or former priority of the task or an error code (a negative

number) if an error occurred.

A task can change its own priority by calling:

```
SetItemPri(CURRENTTASK->t.n_Item, newPriority);
```

Keep in mind that changing a task's priority affects its status in the ready queue. If you drop the priority below other tasks in the queue, the task can stop executing. If you raise the priority above other tasks in the queue, the task can run alone while the other tasks wait.

Starting and Ending Threads

Threads, as you recall, differ from child tasks in that they are more tightly bound to their parent task. They share the parent task's memory and can't transfer their ownership. However, they still need to be started and ended just as tasks do. This section describes the calls used to start and end threads.

Creating a Thread

The `CreateThread()` function creates a thread:

```
Item CreateThread ( cconst char *name, uint8 pri, void (*code)
(), int32 stacksize )
```

The first argument, `name`, is the name of the thread to create. The second argument, `pri`, is the priority that the thread will have. The third argument, `code`, is a pointer to the code that the thread will execute. The last argument, `stacksize`, specifies the size in bytes of the thread's stack. If successful, the call returns the item number of the thread. If unsuccessful, it returns an error code.

Before calling `CreateThread(),` you must determine what stack size to use for the thread. There is no default size for the `stacksize` argument; however, it's important to make the size large enough so that stack overflow errors do not occur. Stack overflow errors are characterized by random crashes that defy logical analysis, so it's a good approach to start with a large stack size and reduce it until a crash occurs, then double the stack size. In the sample code later in this chapter, the stack size is set to 10000. A good size to start with is 2048.

Example 1 shows the process of creating a thread:

Example 1: *Starting a task.*

```
#define STACKSIZE (10000)
...
int main(int argc, char *argv[])
{
uint8  Priority;
Item  T1;
...
Priority = CURRENTTASK->t.n_Priority;
T1 = CreateThread("Thread1", Priority, Thread1Proc, STACKSIZE);
...
```

```
}
int Thread1Proc()
{
/* This is the code for Thread1Proc */
}
```

Communication

Because threads share memory with the parent task, global variables can pass information among threads and tasks. In the sample code later in this chapter a number of global variables are declared at the start of the program, prior to the code for the threads and the parent task code. As global variables, they are accessible to the main routine and to the threads-an example of threads sharing the same memory as the parent task.

The code example given at the end of this chapter is a good illustration of using `CreateThread()` to start two threads. It shows the use of global variables that are shared by all threads, which signal among threads and the parent task. Signals for both threads are first allocated with `AllocSignal().` The parent task then uses `WaitSignal()` to wait for an event from either of the threads.

Opening Folios

Whenever you make folio calls from a thread, be sure to open the appropriate folios first. It's tempting to think that because a folio has been opened by the parent task, the thread can make calls from the same folio without opening it. Not true-and potentially fatal to the thread. Each thread must open all folios it intends to use.

Ending a Thread

Use `DeleteThread()` to end a thread when a parent task finishes with it:

```
Err DeleteThread( Item thread )
```

This function takes the item number of the thread to delete as its only argument, and returns a negative error code if an error occurs. Although all threads terminate automatically when the parent task ceases, it's good programming practice to kill a thread as soon as the parent task finishes using it. Note that when you terminate a thread, the thread's memory (which is shared with the parent task) is *not* freed, and remains the property of the parent task. A thread can also terminate itself by calling `exit()` or just returning.

Advanced Thread Usage

The `CreateThread()` and `DeleteThread()` functions are convenience routines, which make it easy to create and delete threads for the most common uses. It is sometimes necessary to have better control over thread creation. This requires allocating resources yourself and creating the thread item using `CreateItem()` or `CreateItemVA()`.

Creating a thread involves allocating memory for the stack that the thread will use, and constructing a tag argument list that describes how the thread should be created. The tags you supply are:

- **TAG_ITEM_NAME**, Provides a pointer to the name of the thread.
- **TAG_ITEM_PRI**, Provides a priority for the thread in the range 10 to 199. If this tag is not given, the thread inherits the priority of the current context.
- **CREATETASK_TAG_PC**, Provides a pointer to the code to be executed as a thread.
- **CREATETASK_TAG_SP**, Provides a pointer to the memory buffer to use as stack for the thread.
- **CREATETASK_TAG_STACKSIZE**, Specifies the size, in bytes, of the memory buffer reserved for the thread's stack.
- **CREATETASK_TAG_ARGC**, A 32-bit value that is passed as a first argument to the thread being created. If this value is omitted, the first argument is 0.
- **CREATETASK_TAG_ARGP**, A 32-bit value that is passed as a second argument to the thread being created. If this is omitted, the second argument is 0.
- **CREATETASK_TAG_MSGFROMCHILD**, Provides the item number of a message port. The kernel sends a status message to this port whenever the thread or task being created exits. The message is sent by the kernel after the task has been deleted. The `msg_Result` field of the message contains the exit status of the task. This is the value the task provided to `exit()`, or the value returned by the task's primary function. The `msg_DataPtr` field of the message contains the item number of the task that just terminated. If a task exited on its own, this is the item number of the task itself. It is the responsibility of the task that receives the status message to delete it when the message is no longer needed by using `DeleteMsg()`.
- **CREATETASK_TAG_ALLOCDTHREADSP**, Tells the kernel that if this thread dies by itself, by either returning to the kernel or by calling `exit()`; the memory used for its stack must be freed automatically. When this tag is not provided, you are responsible for freeing the stack whenever the thread terminates.

Loading and Executing Separate Code Modules

Using the `LoadProgram()` and `LoadProgramPrio()` calls mentioned previously, you can load an executable file from external storage, and launch it as a totally independent task. This gives the code its own memory pages and resource tracking. It also means that even if the program being loaded is only 4 KB big, it will still consume an entire page of RAM, which is normally 32 KB or 16 KB. Loading such programs can waste a lot of memory if you launch many of them.

Instead of launching separate tasks, it is generally better for an application to launch separate threads. As described above, you can use `CreateThread()` to do this with functions that are within your program.

In many cases, it is desirable to split up a program into multiple pieces, only one of which needs to be in memory at any given time. For example, the code to control the high score table of your game doesn't need to reside in memory when playing the game. It only needs to be loaded when the time comes to display the high score table.

Portfolio supports the partitioning of a game into multiple executable files with the `LoadCode()` function. Using `LoadCode()`, you can load the data for an executable file in memory, and then launch the code as a thread using `ExecuteAsThread()` or `ExecuteAsSubroutine()`. Once you finish with the code module, you can unload it from memory using the `UnloadCode()` function.

`LoadCode()` is defined as:

```
Err LoadCode( char *fileName, CodeHandle *code )
```

The `fileName` argument specifies the name of the executable file to load. This is a regular executable file as generated by the ARM linker, except that it must be linked with different startup code. If you plan on executing this file as a thread, then you must link the file with *threadstartup.o* instead of *cstartup.o*. If you plan on executing the file as a subroutine, it must be linked with *subroutinestartup.o*. The code argument is a pointer to a variable where a handle to the loaded code is stored.

`LoadCode()` reads the file and allocates memory to hold its contents. The allocated memory is within the address space of the current task. If the current task exits, the memory is automatically freed.

Once code has been loaded, it can be executed using either `ExecuteAsThread()` or `ExecuteAsSubroutine()`.

```
Err ExecuteAsThread( CodeHandle code, uint32 argc, char **argv,
char *threadName, int32 priority )
Err ExecuteAsSubroutine( CodeHandle code, uint32 argc, char
**argv )
```

The `code` argument specifies the loaded code to run. This is the value stored by `LoadCode()` in the `CodeHandle` variable passed to it. `argc` and `argv` are two 32-bit values Portfolio does not use, and are just passed through to the code being executed. These are the values that the `main()` routine in the loaded code receives for its `argc` and `argv` parameters. Finally, when executing a thread, you must specify a thread name, and priority, just like when using `CreateThread()`.

Once the code completes execution, call `UnloadCode()`:

```
Err UnloadCode( CodeHandle code )
```

This removes the loaded code and frees its memory.

When you load external code and execute it using the above function, it is important to understand that the code is actually a separate executable file. This means that global variables in one code module cannot be accessed by a different code module.

When you execute loaded code as a thread, any memory this thread allocates does not get freed when the thread exits or when the code is unloaded. The thread must clean up after itself, just like a thread created using `CreateThread()`.

If loaded code is being executed as a subroutine and calls `exit()`, this call causes the entire program, not just the subroutine, to exit.

Example

Example 1 contains sample code for using threads and signals.

The `main()` routine launches two threads. These threads simply sit in a loop and count. After a given number of iterations through their loop, they send a signal to the parent task.

When the parent task gets a signal, it wakes up and prints the current counters of the threads to show how much they were able to count.

Example 1: *Using Threads (signals.c).*

```
#include "types.h"
#include "task.h"
#include "kernel.h"
#include "stdio.h"
#include "operror.h"


/*****************************************************************************/


/* Global variables shared by all threads. */
static int32  thread1Sig;
static int32  thread2Sig;
static Item   parentItem;
static uint32 thread1Cnt;
static uint32 thread2Cnt;


/*****************************************************************************/


/* This routine shared by both threads */
static void DoThread(int32 signal, uint32 amount, uint32 *counter)
{
uint32 i;

    while (TRUE)
    {
        for (i = 0; i < amount; i++)
        {
            (*counter)++;
            SendSignal(parentItem,signal);
        }
    }
}
```

```
/*****************************************************************************/

static void Thread1Func(void)
{
    DoThread(thread1Sig, 100000, &thread1Cnt);
}

/*****************************************************************************/

static void Thread2Func(void)
{
    DoThread(thread2Sig, 200000,&thread2Cnt);
}

/*****************************************************************************/

int main(int32 argc, char **argv)
{
uint8  parentPri;
Item   thread1Item;
Item   thread2Item;
uint32 count;
int32  sigs;

    /* get the priority of the parent task */
    parentPri  = CURRENTTASK->t.n_Priority;

    /* get the item number of the parent task */
    parentItem = CURRENTTASK->t.n_Item;

    /* allocate one signal bits for each thread */
    thread1Sig = AllocSignal(0);
    thread2Sig = AllocSignal(0);

    /* spawn two threads that will run in parallel */
    thread1Item = CreateThread("Thread1", parentPri, Thread1Func, 2048);
    thread2Item = CreateThread("Thread2", parentPri, Thread2Func, 2048);

    /* enter a loop until we receive 10 signals */
    count = 0;
    while (count < 10)
    {
        sigs = WaitSignal(thread1Sig | thread2Sig);
```

```
        printf("Thread 1 at %d, thread 2 at

          %d\n",thread1Cnt,thread2Cnt);

        if (sigs & thread1Sig)
            printf("Signal from thread 1\n");

        if (sigs & thread2Sig)
            printf("Signal from thread 2\n");

        count++;
    }

    /* nuke both threads */
    DeleteThread(thread1Item);
    DeleteThread(thread2Item);
}
```

Function Calls

The following calls control tasks and threads. See Kernel Folio Calls, for more information on these calls.

Starting Tasks

The following calls start tasks:

- `CreateItem()` Creates an item.
- `LoadProgram()` Launches a program.
- `LoadProgramPrio()` Launches a program and gives it a priority.

Ending Tasks

The following calls delete or exit a task:

- `DeleteItem()` Deletes an item.
- `exit()` Exits from a task or thread.

Loading and Unloading Code

The following calls load or unload code:

- `LoadCode()` Loads a binary image into memory, and obtains a handle to it.
- `UnloadCode()` Unloads a binary image previously loaded with `LoadCode()`.

Starting Threads

The following calls create a thread:

- `CreateThread()` Creates a thread.
- `ExecuteAsThread()` Executes previously loaded code as a thread.

Ending Threads

The following calls delete and exit from a thread:

- `DeleteThread()` Deletes a thread.
- `exit()` Exits from a task or thread.

Controlling Tasks

The following calls control tasks and threads.

- `SendSignal()` Sends one or more signals to another task.
- `SetItemOwner()` Changes the owner of an item.
- `SetItemPri()` Changes the priority of an item.
- `WaitSignal()` Waits until a signal is sent.
- `Yield()` Gives up the CPU to a task of equal priority.

Using Tags and TagArgs

This chapter provides information on tags and TagArgs. It contains the following sections:

About Tags and TagArgs

A TagArg is a data structure that is used throughout Portfolio. The structure provides potentially long and variable lists of parameters to system function calls. TagArgs, also referred to simply as tags, provide a very flexible and expandable mechanism that allows Portfolio to evolve with minimal impact on application code.

Tags and TagArgs

A TagArg is a simple data structure that contains a command or attribute, and an argument for that command or attribute. The TagArg structure looks like:

```
typedef struct TagArg
{
    uint32  ta_Tag;
    TagData ta_Arg;
} TagArg;
```

TagArg structures are generally used in array form. Many Portfolio function calls take an array of TagArg structures as a parameter. Each function call that uses TagArg arrays defines which commands or attributes can be used with the call.

Arrays of TagArg structures are scanned from start to finish by the function calls that use them. Each element in the array specifies an individual command or attribute pertinent to that function. If a given command appears more than once in a TagArg array, the last occurrence within the array takes precedence. If an unknown command appears in a TagArg, the whole function call fails and returns an appropriate error.

Special Tag Commands

Portfolio provides three special tag commands that have universal meanings in all system calls: TAG_END, TAG_NOP, and TAG_JUMP. By setting the ta_Tag field to these commands, you can control how TagArg arrays are processed by the system.

TAG_END

The TAG_END command tells the system that this structure marks the end of an array of TagArg structures. The system stops scanning the array and goes no further.

TAG_NOP

The TAG_NOP command tells the system to ignore the particular TagArg structure. The system just skips ahead and continues processing with the next TagArg structure.

TAG_JUMP

The TAG_JUMP command links arrays of TagArg structures together. The ta_Arg field for this command must point to another array of TagArg structures. When the system encounters such a TagArg structure, it stops scanning the current array, and resumes scanning at the address specified by ta_Arg.

Tags and VarArgs

Although Portfolio TagArg calls traditionally use static arrays of TagArg structures to define their parameters, a much superior approach is possible by using the C VarArgs capability.

VarArgs lets C routines like `printf()` accept a variable number of parameters. You can use VarArgs to construct arrays of TagArg structures on the fly within a function call you are making. This is generally easier to write, understand, and maintain than static arrays.

Most Portfolio functions that accept TagArg arrays as parameters also have VarArgs counterparts. These counterparts have the same names as the regular functions, with the addition of a VA suffix to identify them as VarArgs.

For example, the `CreateItem()` kernel function accepts an array of TagArg structures as a parameter. The `CreateItemVA()` function is identical in purpose and function, except that it uses a VarArgs list of tags instead of a pointer to an array.

To build up a tag list on the stack, you enumerate all commands and their arguments in sequence, separated by commas, and terminate them with a `TAG_END` tag command. Here are examples of using `CreateItem()` and `CreateItemVA()` to create a message port.

Example 1: *Creating a message port*

```
static TagArgs tags[] =
{
    {CREATEPORT_TAG_SIGNAL,   0},
    {CREATEPORT_TAG_USERDATA, 0},
    {TAG_END,              0}
};

{
Item mp;

    tags[0].ta_Arg = (void *)sigMask;
    tags[1].ta_Arg = (void *)userData;
    mp = CreateItem(MKNODEID(KERNELNODE,MSGPORTNODE),tags);
}

{
Item mp;
```

```
    mp = CreateItemVA(MKNODEID(KERNELNODE,MSGPORTNODE),
                      CREATEPORT_TAG_SIGNAL,   sigMask,
                      CREATEPORT_TAG_USERDATA, userData,
                      TAG_END);
}
```

As you can see, the version using `CreateItemVA()` is easier to understand. All the definitions pertaining to the tags can be kept within the function call itself, instead of requiring a separate array.

Parsing Tags

If you write utility routines to be shared by many programmers, it is often useful to implement functions that take TagArg arrays in a manner similar to the system functions.

The `NextTagArg()` function lets you easily go through all the TagArg structures in an array. The function automatically handles all system tag commands like `TAG_NOP` and `TAG_JUMP`, and only returns TagArg structures that do not contain system tag commands. You give `NextTagArg()` a pointer to a variable that points to the tag array to process. It returns a pointer to the first TagArg structure within the array. You then call the function repeatedly until it returns NULL. Every time you call it, it returns the next TagArg structure in the array.

The `FindTagArg()` function takes a pointer to an array of TagArg structures and to a particular tag command. The function scans the supplied array looking for a TagArg structure that has the requested command. It returns a pointer to the TagArg structure, or NULL if no structure with that command is found.

The `GetTagArg()` function works much as `FindTagArg()` does, except that instead of returning a pointer to a TagArg structure, it returns the value stored in the `ta_Arg` field of the TagArg structure. You also give the function a default data value. If the desired TagArg can't be found, the function returns the default data value to you.

Finally, the `DumpTagList()` function displays all the tag commands and arguments in a TagArg array to the debugging terminal. This is very useful when you need to see every tag value being passed to a function call.

TagArg Function Calls

These functions help you parse arrays of TagArg structures:

- `DumpTagList` Displays all of the TagArg structures within a TagArg array to the debugging terminal.
- `FindTagArg` Finds a TagArg that contains a given command within a TagArg array.
- `GetTagArg` Returns the argument for a TagArg that contains a given command within a TagArg array.
- `NextTagArg` Iterates through all the structures in a TagArg array.

Managing Linked Lists

This chapter explains how to manage linked lists. For details about the functions described here, see Kernel Folio Calls, in the *3DO System Programmer's Reference*.

This chapter contains the following topics:

About Linked Lists

Linked lists are used throughout Portfolio and in applications you create with it. To make using lists easier (and to meet the needs of system software and its lists), the kernel defines a special type of linked list, known as a Portfolio list. The kernel also provides a variety of functions for creating and managing Portfolio lists.

Like all linked lists, Portfolio lists are dynamic; they can expand and contract as needed. Their contents, known as nodes, are ordered (there is a first node, a second node, and so on), and you can add a new node at any position in a list. The following sections explain how Portfolio lists are different from other linked lists.

Characteristics of Portfolio Lists

Unlike ordinary linked lists, which contain only nodes, Portfolio lists also contain a special component known as an anchor, which marks both ends of the list. The anchor (which is implemented as a C union) serves as both the beginning-of-list marker (known as the head anchor) and the end-of-list marker (known as the tail anchor). Figure 1 illustrates an anchored list.

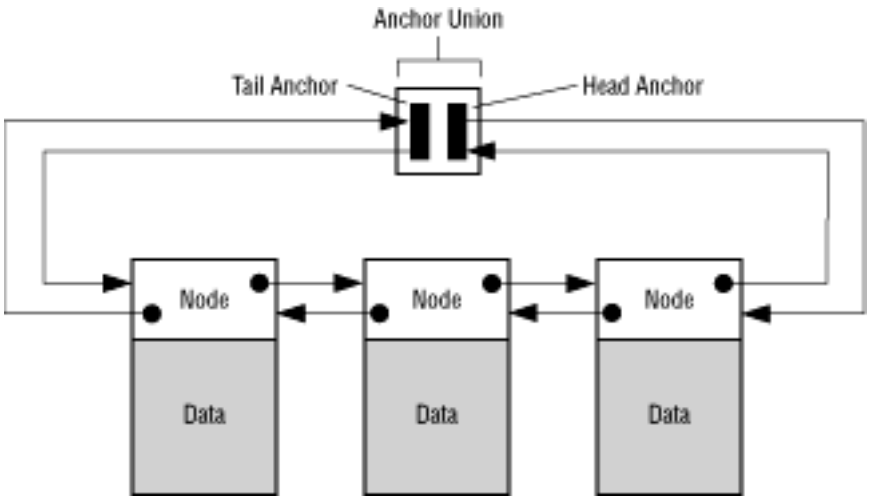

Figure 1: *Anchored list.*

When a task traverses a Portfolio list, it determines whether it's at the beginning or end of the list by testing to see if the subsequent node is an anchor.

As the previous illustration shows, Portfolio lists are doubly linked: Each node contains two pointers, one to point to the following node or anchor and one to the previous node or anchor. As a result, back-to-front list traversals are as efficient as front-to-back traversals.

Characteristics of Nodes

In Portfolio lists, there are special data structures that contain only information needed for list management. This information includes the necessary forward and backward pointers, a priority value (described later in this section), and other fields that are used primarily by the operating system. A node can also have a name. Use the `FindNamedNode()` call to find a node by name.

To create a list component, you define a data structure whose first field is a node. An example is the NoteTracker structure defined in the music library:

Example 1: *The Note Tracker structure in the music library.*

```
typedef struct NoteTracker
{
        Node    nttr_Node;
        int8    nttr_Note;
        int8    nttr_MixerChannel;
        uint8   nttr_Flags;
        int8    nttr_Channel;  /* MIDI */
        Item    nttr_Instrument;
} NoteTracker;
```

To pass such a component to one of the many list-manipulation functions that takes a Node structure as an argument, you simply cast the argument to type Node. Here's an example:

```
    AddTail( &DSPPData.dspp_ExternalList, (Node *) dext );
```

Every node in a list has a priority (a value from 0-255 that is stored in the `n_Priority` field of the Node structure). When you use a list, you have the option of keeping its nodes sorted by priority (done automatically by the kernel if you use `InsertNodeFromHead()` or `InsertNodeFromTail()`), or you can specify other ways to arrange the contents (by using the `UniversalInsertNode()` function). You can also change the priority of a node in a list with the `SetNodePri()` function, whereupon the kernel automatically repositions the node in the list to reflect its new priority value.

A node can be in only one list at a time.

Creating and Initializing a List

To create an empty linked list, you first create a variable of type `List`. You then initialize the list, which gives the list a name and initializes its head and tail anchors, by calling the `InitList()` function:

```
void InitList( List *l, const char *name )
```

The `l` argument is a pointer to the list to be initialized. The `name` argument is the name of the list.

Another way to initialize a list is by using the `INITLIST()` macro. This macro lets you define a fully initialized `List` as a variable. By using the macro, you avoid the need to call the `InitList()` function. Here is an example of using the `INITLIST` macro to create a variable called `listOfStuff`:

```
static List listOfStuff = INITLIST(listOfStuff,"List Name");
```

Adding a Node to a List

You can add a node to a list in any of the following ways:

- The beginning or end of a list.
- A position in the list that corresponds to the node's priority.
- A position in the list determined by comparing one or more values contained in the node to values of nodes currently in the list, or relative to another node already in the list.

The following sections describe each of these cases.

Adding a Node to the Head of a List

To add a node to the head of a list, use the `AddHead()` function:

```
void AddHead( List *l, Node *n )
```

The `l` argument is a pointer to the list to which to add the node, while the `n` argument is a pointer to the node to add.

Adding a Node to the Tail of a List

To add a node to the tail of a list, use the `AddTail()` function:

```
void AddTail( List *l, Node *n )
```

The `l` argument is a pointer to the list to which to add the node, while the `n` argument is a pointer to the node to add.

Adding a Node After Another Node in a List

To add a node to a list and position it after another node already in the list, use the `InsertNodeAfter()` function:

```
void InsertNodeAfter( Node *oldNode, Node *newNode )
```

The `oldNode` argument is a pointer to a node already in the list, while the `newNode` argument is a pointer to the new node to insert.

Adding a Node Before Another Node in a List

To add a node to a list and position it before another node already in the list, use the `InsertNodeBefore()` function:

```
void InsertNodeBefore( Node *oldNode, Node *newNode )
```

The `oldNode` argument is a pointer to a node already in the list, while the `newNode` argument is a pointer to the new node to insert.

Adding a Node According to Its Priority

The nodes in a list are often arranged by priority. To insert a new node immediately before any other nodes of the same priority, use the `InsertNodeFromHead()` function:

```
void InsertNodeFromHead( List *l, Node *n )
```

As in the other functions for adding nodes, the `l` argument is a pointer to the list to which to add the node, while the `n` argument is a pointer to the node to add.

The name `InsertNodeFromHead()` refers to the way the kernel traverses the list to find the correct position for a new node: it compares the priority of the new node to the priorities of nodes already in the list, beginning at the head of the list, and inserts the new node immediately after nodes with higher priorities. If the priorities of all the nodes in the list are higher than the priority of the new node, the node is added to the end of the list.

To insert a new node immediately after all other nodes of the same priority, you use the `InsertNodeFromTail()` function:

```
void InsertNodeFromTail( List *l, Node *n )
```

Again, the `l` argument is a pointer to the list to which to add the node, while the `n` argument is a pointer to the node to add. As with `InsertNodeFromHead(),` the name `InsertNodeFromTail()` refers to the way the kernel traverses the list to find the correct position for the new node: it compares the priority of the new node to the priorities of nodes already in the list, beginning at the tail of the list, and inserts the new node immediately before nodes with lower priorities. If the priorities of all the nodes in

the list are lower, the node is added to the head of the list.

Adding a Node According to Other Node Values

Arranging list nodes by priority is only one way to order a list. You can arrange the nodes in a list by node values other than priority by using the `UniversalInsertNode()` function to insert new nodes:

```
void UniversalInsertNode( List *l, Node *n, bool (*f)(Node *n,
Node *m) )
```

The `l` argument is a pointer to the list to which to add the node, while the `n` argument is a pointer to the node to add. Like `InsertNodeFromHead()`, `UniversalInsertNode()` compares the node to be inserted with nodes already in the list, beginning with the first node. The difference is that it uses the comparison function `f` provided by your task to compare the new node to existing nodes. If the comparison function returns TRUE, the new node is inserted immediately before the node to which it was compared. If the comparison function always returns FALSE, the new node becomes the last node in the list. The comparison function, whose arguments are pointers to two nodes, can use any data in the nodes for the comparison.

Changing the Priority of a Node

List nodes are often ordered by priority. You can change the priority of a list node (and thereby change its position in the list) by using the `SetNodePri()` function:

```
uint8 SetNodePri( Node *n, uint8 newpri )
```

The `n` argument is a pointer to the list node whose priority you want to change. The `newpri` argument specifies the new priority for the node (a value from 0 to 255). The function returns the previous priority of the node. When you change the priority of a node, the kernel automatically moves the node immediately.

Removing a Node From a List

You can remove the first node, the last node, or a specific node from a list. The following sections explain how to do it.

Removing the First Node

To remove the first node from a list, use the `RemHead()` function:

```
Node *RemHead( List *l )
```

The `l` argument is a pointer to the list from which you want to remove the node. The function returns a pointer to the node that was removed from the list or NULL if the list was empty.

Removing the Last Node

To remove the last node from a list, use the `RemTail()` function:

```
Node *RemTail( List *l )
```

The `l` argument is a pointer to the list from which you want to remove the node. The function returns a pointer to the node that was removed from the list or NULL if the list was empty.

Removing a Specific Node

To remove a specific node from a list, use the `RemNode()` function:

```
void RemNode( Node *n )
```

The `n` argument is a pointer to the node you want to remove. Because a node can be only in one list, the node is automatically removed from the correct list.

Finding out If a List Is Empty

To find out if a list is empty, use `IsListEmpty()`:

```
bool IsListEmpty( const List *l )
```

The `l` argument is a pointer to the list to check. The macro returns TRUE if the list is empty or FALSE if it isn't.

Traversing a List

You can traverse a list equally quickly from front to back or from back to front. The following sections explain how.

Traversing a List From Front to Back

To traverse a list from front to back, use `ScanList()`. It iterates through all of the elements in the list.

Example 1: *Traversing a list front to back.*

```
ScanList(list,n,DataType)
{
    /* here you can dereference "n" */
}
```

`ScanList()` is a macro made up of the simpler macros `FirstNode()`, `IsNode()`, and `NextNode()`.

Use `FirstNode()` to get the first node in a list:

```
Node *FirstNode( const List *l )
```

The `l` argument is a pointer to the list the node is in. The macro returns a pointer to the first node in the list or, if the list is empty, a pointer to the tail anchor.

To check to see if a node is an actual node rather than the tail anchor, use `IsNode()`:

```
bool IsNode( const List *l, const Node *n )
```

The `l` argument is a pointer to the list containing the node; the `n` argument is a pointer to the node to check. The macro returns FALSE if it is the tail anchor or TRUE for any other node. (`IsNode()`returns TRUE for any node that is not the tail anchor, no matter if the node is in the specified list.)

To go from one node to its successor in the same list, use `NextNode()`:

```
Node *NextNode( const Node *n )
```

The n argument is a pointer to the current node. (This node must be in a list.) The macro returns a pointer to the next node in the list or, if the current node is the last node in the list, to the tail anchor.

Traversing a List From Back to Front

To traverse a list from front to back, use the ScanListB() macro. It iterates through all the elements in the list.

Example 2: *Traversing a list back to front.*

```
ScanListB(list,n,DataType)
{
    /* here you can dereference "n" */
}
```

ScanListB() is a macro made up of the simpler macros LastNode(), IsNodeB(), and PrevNode().

Use LastNode() to get the last node in a list:

```
Node *LastNode( const List *l )
```

The l argument is a pointer to the list the node is in. The macro returns a pointer to the last node in the list or, if the list is empty, a pointer to the head anchor.

To check to see if a node is an actual node rather than the head anchor, use IsNodeB():

```
bool IsNodeB( const List *l, const Node *n )
```

The l argument is a pointer to the list containing the node to check; the n argument is a pointer to the node to check. The macro returns FALSE if it is the head anchor or TRUE for any other node. (The macro returns TRUE for any node that is not the head anchor, whether or not the node is in the specified list.)

To go from one node to its predeccessor in the list, use the PrevNode() macro:

```
Node *PrevNode( const Node *n )
```

The n argument is a pointer to the current node. (This node must be in a list.) The macro returns a pointer to the previous node in the list or, if the current node is the first node in the list, to the head anchor.

Finding a Node by Name

Because list nodes can have names, you can search for a node with a particular name with the `FindNamedNode()` function:

```
Node *FindNamedNode( const List *l, const char *name )
```

The `l` argument is a pointer to the list to search; the `name` argument is the name of the node for which to search. The function returns a pointer to the Node structure or NULL if the named node is not found. The search is not case-sensitive; that is, the kernel does not distinguish uppercase and lowercase letters in the node names.

Finding a Node by Its Ordinal Position

You can find a node that appears in a given position from the beginning or end of a list. To find a node from the beginning of a list, use the `FindNodeFromHead()` function:

```
Node *FindNodeFromHead( const List *l, uint32 position )
```

The `l` argument is a pointer to the list to search; the `position` argument is the position of the node sought within the list, counting from the head of the list. The first node in the list has position 0. The function returns a pointer to the node, or NULL if there are not enough nodes in the list for the requested position.

To find a node from the end of a list, use the `FindNodeFromTail()` function:

```
Node *FindNodeFromTail( const List *l, uint32 position )
```

The `l` argument is a pointer to the list to search; the `position` argument is the position of the node sought within the list, counting from the tail of the list. The last node in the list has position 0. The function returns a pointer to the node, or NULL if there are not enough nodes in the list for the requested position.

Determining the Ordinal Position of a Node

Given a list and a node, you can determine the position of the node within the list, counting from the beginning or the end of the list. To determine the position of a node relative to the beginning of a list, use the `GetNodePosFromHead()` function:

```
int32 GetNodePosFromHead( const List *l, const Node *n );
```

The `l` argument is a pointer to the list in which the node is located; the `n` argument is the node to find in the list. The function scans the list looking for the node, and returns the position of the node within the list. The first node in the list has position 0. If the node cannot be located in the list, the function returns -1.

To determine the position of a node relative to the end of a list, use the `GetNodePosFromTail()` function:

```
int32 GetNodePosFromTail( const List *l, const Node *n );
```

The `l` argument is a pointer to the list in which the node is located; the `n` argument is the node to find in the list. The function scans the list looking for the node, and returns the position of the node relative to the end of the list. The last node in the list has position 0. If the node cannot be located in the list, the function returns -1.

Counting the Number of Nodes in a List

To count the number of nodes in a list, use the `GetNodeCount()` function:

```
uint32 GetNodeCount( const List *l );
```

The `l` argument is a pointer to the list of which you want count the nodes. The function returns the number of nodes currently in the list.

Example

Example 4 illustrates how to traverse a list.

Example 1: *List traversal.*

```
void DumpList ( const List *theList )
{
    Node *n;

    ScanList(theList,n,Node)
    {
        printf("Node = 0x%lx\n", n);
    }
}
```

Primary Data Structures

The following sections describe the most important data structures that use linked lists. With the exception of the n_Name field in a Node structure (which you use to name a node), tasks perform all normal operations involving lists by using the function calls described in this chapter.

The Node Structure

The Node data structure is the standard structure for any named node in a linked list. The n_Name field lets you easily locate any node in a linked list.

Example 1: *The Node data structure.*

```
typedef struct Node
{
    struct Node          *n_Next;        /* pointer to next node in list */
    struct Node          *n_Prev;        /* pointer to previous node in list */
    uint8                 n_SubsysType;  /* what folio manages this node */
    uint8                 n_Type;        /* what type of node for the folio */
    uint8                 n_Priority;    /* queueing priority */
    uint8                 n_Flags;       /* flags used by the system */
    int32                 n_Size;        /* total size of node including hdr */
    char                 *n_Name;        /* ptr to null terminated string or
NULL */
} Node, *NodeP;
/* n_Flag bits */
/* bits 4-7 are reserved for the system */
/* bits 0-3 are available for node specific use by the system */
#define NODE_RSRV1       0x40
#define NODE_SIZELOCKED  0x20          /* The size of this item has been

                                       /* locked down */
#define NODE_ITEMVALID  0x10           /* This is an ItemNode */
#define NODE_NAMEVALID  0x80           /* This node's namefield is valid */
```

MinNode and NamelessNode Structures

In addition to the regular Node structure, Portfolio also defines the MinNode and NamelessNode structures. These structures can be used in place of the full Node structure when defining your own lists. These structures have much less overhead than the Node structure, so your lists use less memory.

The NamelessNode structure is identical to the Node structure, except that it doesn't have the n_Name field. You can use the NamelessNode structure in place of a Node structure for all the functions and macros explained in this chapter, except for the FindNamedNode() and DumpNode() functions. Since these functions use the n_Name

field, they cannot handle the NamelessNode.

Example 2: *The NamelessNode structure.*

```
/* Node structure used when the Name is not needed */
typedef struct NamelessNode
{
    struct NamelessNode *n_Next;
    struct NamelessNode *n_Prev;
    uint8 n_SubsysType;
    uint8 n_Type;
    uint8 n_Priority;
    uint8 n_Flags;
    int32 n_Size;
} NamelessNode, *NamelessNodeP;
```

The MinNode structure is very small, and provides just enough information to link within lists. It can be used with most functions and macros explained in this chapter, except for `SetNodePri()`, `InsertNodeFromTail()`, `InsertNodeFromHead()`, `FindNamedNode()`, and `DumpNode()`. These functions use the extra fields found in the Node structure, and cannot work with the simple MinNode structure.

Example 3: *The MinNode structure.*

```
/* Node structure used for linking only */
typedef struct MinNode
{
    struct MinNode *n_Next;
    struct MinNode *n_Prev;
} MinNode;
```

The List Data Structure

The List data structure is the means by which nodes are linked together.

Example 4: *The List data structure.*

```
typedef struct List
{
    Node l;                  /* A list is a node itself */
    ListAnchor ListAnchor;   /* Anchor point for list of nodes */
} List, *ListP;
```

The ListAnchor Union

The ListAnchor union contains the forward and backward pointers for the first and last node of any linked list.

Example 5: *The ListAnchor union.*

```
typedef union ListAnchor
{
    struct              /* ptr to first node */
    {                  /* anchor for lastnode */
    Link links;
    Link *filler;
    } head;
    struct
    {
    Link *filler;
    Link links;        /* ptr to lastnode */
    } tail;            /* anchore for firstnode */
} ListAnchor;
```

The Link Data Structure

The Link data structure contains the forward and backward pointers for a linked list.

Example 6: *The Link data structure.*

```
typedef struct Link
{
    struct Link *flink;    /* forward (next) link  */
    struct Link *blink;    /* backward (prev) link */
} Link;
```

Function Calls

The following list contains the function calls that handle linked lists. See Kernel Folio Calls, in the *3DO System Programmer's Reference* for more information on these calls.

Initializing a List

The following calls initialize a list:

- `InitList()` Initializes a list.
- `INITLIST()` Statically initializes a list.

Adding Nodes to a List

The following calls handle nodes:

- `AddHead()` Adds a node to the head of a list.
- `AddTail()` Adds a node to the tail of a list.
- `InsertNodeAfter()` Inserts a node after another node already in a list.
- `InsertNodeBefore()` Inserts a node before another node already in a list.
- `InsertNodeFromHead()` Inserts a node into a list.
- `InsertNodeFromTail()` Inserts a node into a list.
- `UniversalInsertNode()` Inserts a node into a list.

Changing the Priority of a List Node

The following call changes the priority of a node:

- `SetNodePri()` Changes the priority of a list node.

Removing Nodes From a List

The following calls remove nodes from a list:

- `RemHead()` Removes the first node from a list.

- RemNode() Removes a specified node from a list.
- RemTail() Removes the last node from a list.

Checking to See If a List Is Empty

The following call checks if a list is empty:

- IsListEmpty() Checks whether a list is empty.

Testing Nodes and Lists

The following calls are used to test nodes and lists:

- IsNode() Checks whether a node is an actual node or the tail (end-of-list) anchor.
- IsNodeB() Tests whether the node is an actual node or the head (beginning-of-list) anchor.

Traversing a Linked List

The following calls are used to traverse a list:

- FirstNode() Gets the first node in a list.
- LastNode() Gets the last node in a list.
- NextNode() Gets the next node in a list.
- PrevNode() Gets the previous node in a list.
- ScanList()Walks through all the nodes in a list.
- ScanListB() Walks through all the nodes in a list backwards.

Finding a Node by Name

The following call finds a node by name.

- FindNamedNode() Finds a node by specifying its name.

Ordinal Node Position Functions

The following functions deal with nodes using their ordinal position in a list.

- `FindNodeFromHead()` Finds a node in a list by counting from the head of the list.
- `FindNodeFromTail()` Finds a node in a list by counting from the tail of the list.
- `GetNodePosFromHead()` Determines the position of a node within a list, counting from the head of the list.
- `GetNodePosFromTail()` Determines the position of a node within a list, counting from the tail of the list.

Counting the Nodes in a List

- `GetNodeCount()` Counts the number of nodes in a list.
- `DumpNode()` Prints contents of a node to the debugging terminal.

Managing Memory

All tasks need memory. This chapter explains how to allocate memory, how to share it with other tasks, how to get information about it, and how to free it. For details of the functions described here, see Kernel Folio Calls, in the *3DO System Programmer's Reference*.

This chapter contains the following topics:

About Memory

This section explains the fundamentals of memory and memory usage for 3DO software.

Types of Memory

Current 3DO systems include three types of memory: dynamic random access memory (DRAM), video random access memory (VRAM), and nonvolatile random access memory (NVRAM). Future systems may include additional types of memory.

DRAM

Dynamic random access memory (DRAM) is generic memory for operations that don't involve special interactions with 3DO hardware, such as the SPORT bus transfers. Current 3DO systems include 2 MB of DRAM, which can be expanded to either 15 MB (in systems with 1 MB of VRAM) or 14 MB (in systems with 2 MB of VRAM). Future systems may have larger memory capacities.

VRAM

Not all memory is created equal. VRAM is one kind of special purpose memory, that is, memory primarily or exclusively used with particular parts of the 3DO hardware (in this case, the SPORT bus). Video random access memory (VRAM) is RAM for video and graphics operations that involve the SPORT bus. (For information about the SPORT bus, see the Understanding the Cel Engine and SPORT in the *3DO Graphics Programmer's Guide*.) Because VRAM is connected to the standard memory bus as well as the SPORT bus, you can also use it for generic operations.

Current systems include 1 MB of VRAM, which can be expanded to 2 MB. Future systems may have larger memory capacities.

In systems with 2 MB of VRAM, VRAM is divided into two 1 MB banks. The SPORT bus can only transfer data between locations in the same bank, never between banks. When blocks of VRAM are allocated, each block must come from the same bank.

Note: Future 3DO systems may have other kinds of special purpose memory, including memory for direct memory access (DMA), for the cel engine, for audio, and for the digital signal processor (DSP). See Allocating Memory for information about when to specify these future memory types in current software.

NVRAM

Every 3DO system includes at least 32 KB of nonvolatile random access memory (NVRAM) that can preserve information when the system is turned off or rebooted. Applications can use NVRAM to store small amounts of information, such as application configuration information, user preferences, and high scores for games.

Unlike other types of memory, access to NVRAM is through the Portfolio file system, where it is treated as a separate volume. For more information about NVRAM, see The Filesystem and the File Folio.

Organization of Memory

The following sections explain how memory is arranged (in fixed-size units known as *pages*), how it is divided up when allocated (into contiguous arrays of bytes known as *blocks*), how the kernel keeps track of memory available for allocation (in special lists known as *free memory lists*), and where allocated memory comes from (from lists of free memory lists known as free memory pools).

Memory Pages

In 3DO system the hardware divides memory into pages. Currently, the number of pages in the system, also called system pages, is fixed (64 pages of DRAM and 64 pages of VRAM); the size of each page depends on the amount of each type of memory in the system. For example, in current systems with 2 MB of DRAM and 1 MB of VRAM, DRAM pages are 32 KB (2 MB divided by 64) and VRAM pages are 16K (1 MB divided by 64). In a loaded system with 14 MB of DRAM, 2 MB of VRAM, and cruise control, DRAM pages are 256 KB and VRAM pages are 32 KB.

Each page of memory has an owner. Although any task can read any memory location in RAM, only the task that owns a page (or a task that the owner designates) can write to the page. How tasks become owners of memory is explained in How Memory Allocation Works. How tasks can transfer ownership of memory to other tasks is explained in the section Transferring Memory to Other Tasks.

When a task needs memory, it allocates a block of memory from the memory pages it owns. It specifies the size of the block in a call to a memory allocation function. The bytes in a memory block are always contiguous, even when the block crosses page boundaries.

VRAM Memory Pages

As described in the previous section, VRAM is divided by the hardware into 64 pages. These pages are part of the memory protection scheme. VRAM has a secondary page size imposed by the SPORT hardware, and is used for graphics operations. This secondary page size is currently fixed at 2K. SPORT operation can only be performed in increments of 2K with blocks aligned on a 2K boundary. You can find more information about the SPORT device and its restrictions in the *3DO Portfolio Graphics Programmer's Guide*.

For the remainder of this chapter, unless otherwise indicated, the term "pages" refers to the memory

protection pages, and not to the SPORT hardware pages.

Free Memory Lists

The kernel uses linked lists to keep track of memory that is available for allocation. These lists, known as free memory lists, contain entries for all currently free blocks.

Most tasks let the kernel take care of memory allocation and free memory lists. However, tasks that need additional control over memory allocation can create their own memory lists and allocate memory to themselves from those lists.

Free Memory Pools

The kernel keeps a list of free memory lists for each owner of memory. For example, it creates a list containing two free memory lists, one for DRAM and one for VRAM, for each new task that is created. This list of memory lists-which contains all the unused memory that a task owns -is the task's free memory pool. When a task allocates memory, it must allocate it from its own free memory pool.

Note: Threads share the free memory lists with their parent and sibling threads. Access to these lists is controlled by a semaphore.

In addition to the free memory pools for tasks, there is a system-wide free memory pool that contains the unused memory that the kernel owns. The next section explains how memory is transferred from the system-wide free memory pool to the free memory pools for specific tasks.

How Memory Allocation Works

The following sections take you step-by-step through the process of memory allocation and illustrates what happens to memory pools, lists, and blocks at each step.

The Beginning: The System-Wide Free Memory Pool

In the beginning, after the system is started and before any user tasks are launched, there is a system-wide free memory pool with DRAM and VRAM. Figure 1 shows this memory pool.

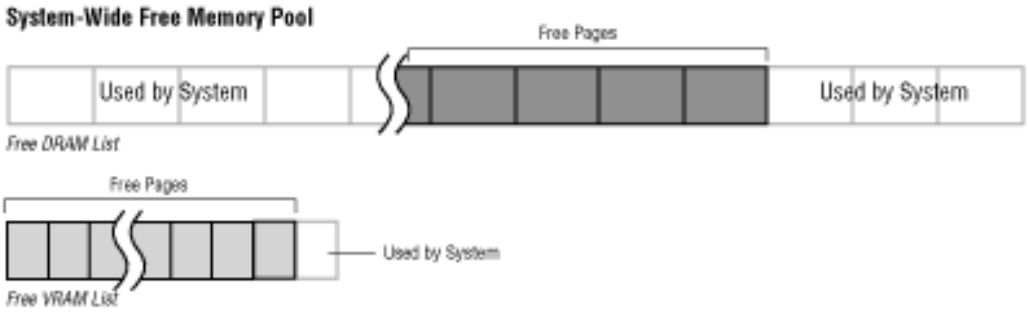

Figure 1: *System-wide free memory pool on bootup.*

Figure 1 illustrates that the system software allocates a number of pages for itself (currently 19 pages of DRAM and 1 page of VRAM for a system with 2 MB DRAM and 1 MB VRAM). On current 3DO systems with 2 MB of DRAM and 1 MB of VRAM, this leaves at least 1440 KB of DRAM and 1008 KB of VRAM for applications. In certain cases, the operating system can free additional memory for applications.

A Task Gets a Free Memory Pool

Next, the first user task is launched. The kernel creates a free memory pool for each new task. The pool consists of two free memory lists: one for DRAM and one for VRAM. The kernel then gives the task the memory it needs to get started (the amount needed for the task's code, global variables, and stack) by transferring the necessary pages of DRAM and VRAM from the system-wide free memory pool to the task's pool, thereby making the task the owner of the memory. Figure 2 shows the result.

Note: A task's VRAM pool is usually empty when the task first starts up.

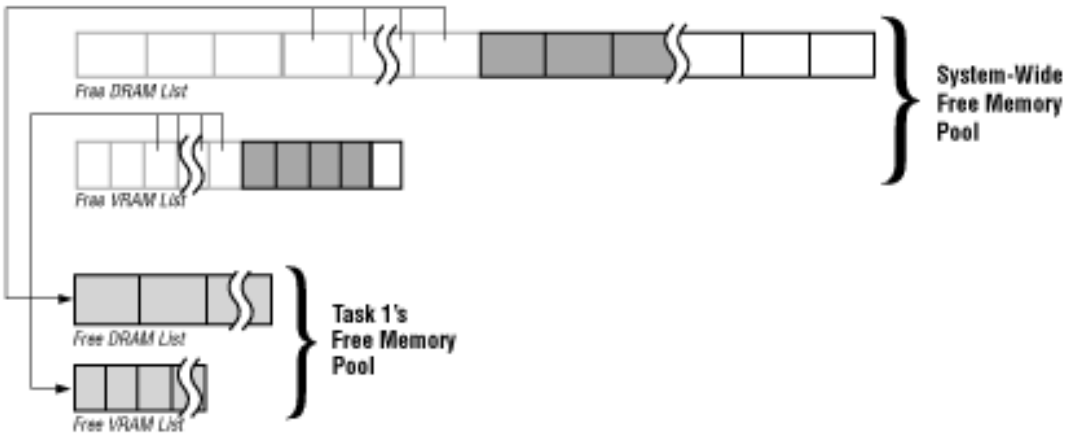

Figure 2: *A task receives necessary memory for its own free memory pool.*

When memory moves from one free memory pool to another, only full pages of memory are moved. All the memory in a page belongs to the same owner, so a page is the smallest amount of memory that can be transferred from one owner to another.

The Task Allocates Its First Memory Block

The new task allocates its first memory block, a block of VRAM for its frame buffer. Figure 3 shows this VRAM allocation.

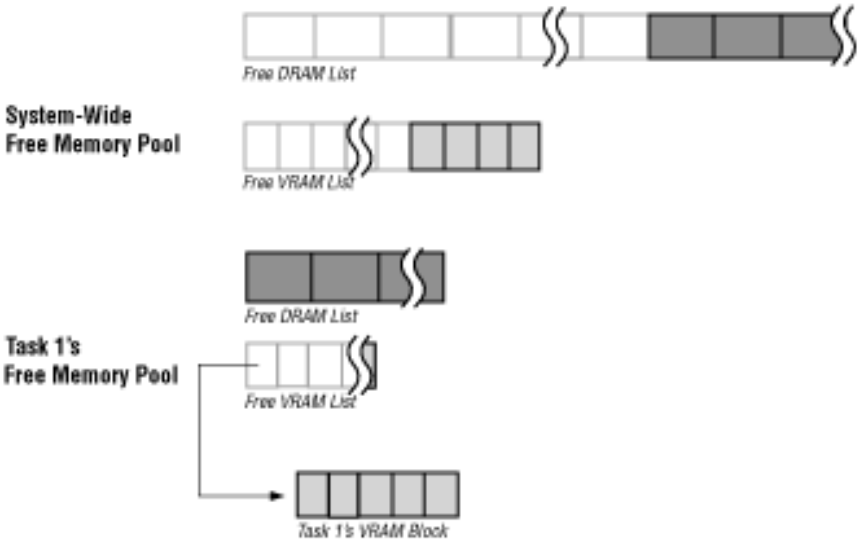

Figure 3: *A task allocates a block of VRAM for its frame buffer.*

The Task Allocates More Blocks

The task allocates several blocks of DRAM, as shown in Figure 4.

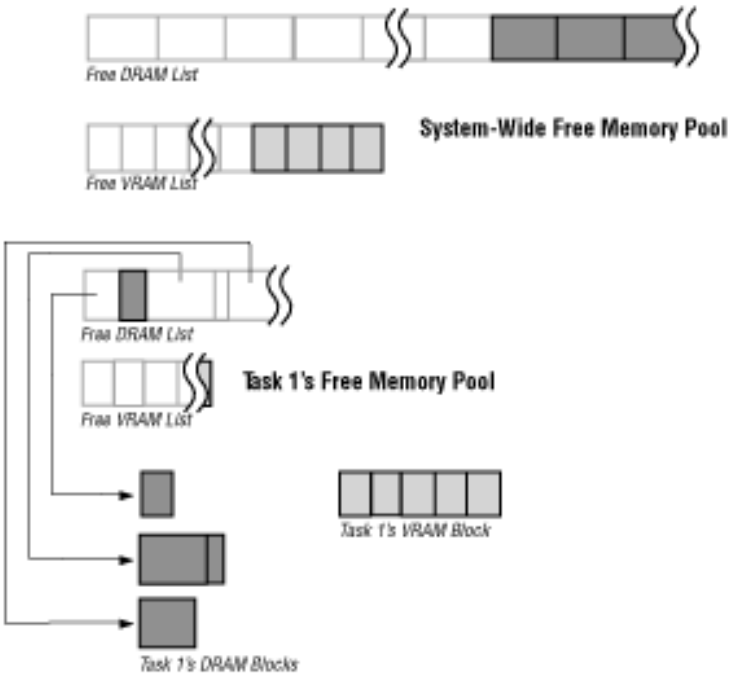

Figure 4: *A task allocates blocks of DRAM.*

The Task Frees Memory

When the task finishes with a particular block of memory, it returns the block to its free memory pool, as shown in Figure 5. The kernel automatically coalesces any free blocks in an adjacent free memory list. To make it easy to find adjacent blocks, the kernel sorts memory lists by memory address; adjacent blocks are thus next to each other in a list.

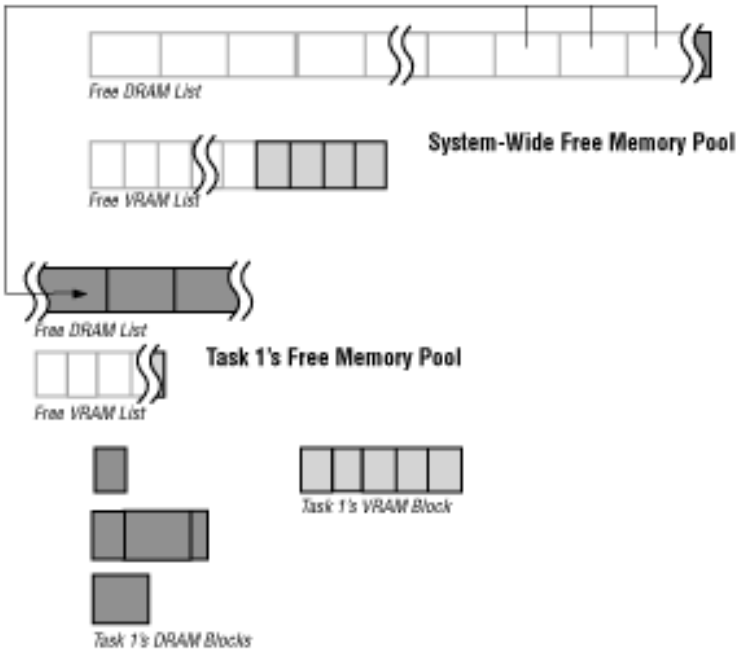

Figure 5: *A task frees a block of memory.*

The Task Needs More Memory

When the amount of memory a task requests in an allocation is more than the amount of contiguous DRAM remaining in the task's free memory pool, the kernel transfers pages from the system free pool to the task, thereby replenishing the task's pool.

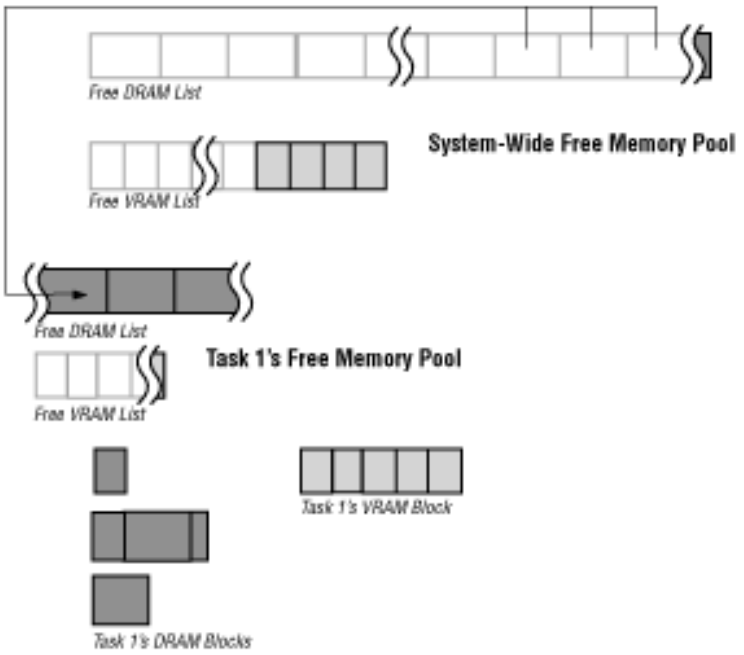

Figure 6: *The kernel transfers pages of DRAM to satisfy a memory request.*

Another Task Gets a Free Memory Pool

A second task is now launched. Like all new tasks, it gets its own free memory pool as shown in Figure 7.

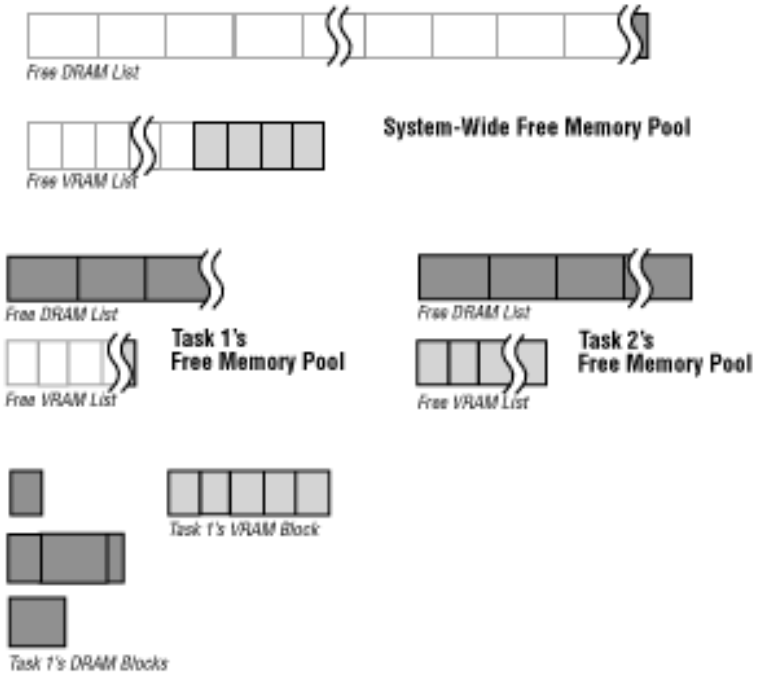

Figure 7: *A second task receives its own free memory pool.*

The System-Wide Free Memory Pool Is Drained

The two tasks keep allocating memory blocks from their free memory pools, and the kernel continues to replenish the task pools from the system-wide free memory pool when necessary. Finally, the system-wide free memory pool gets low on memory, as shown in Figure 8.

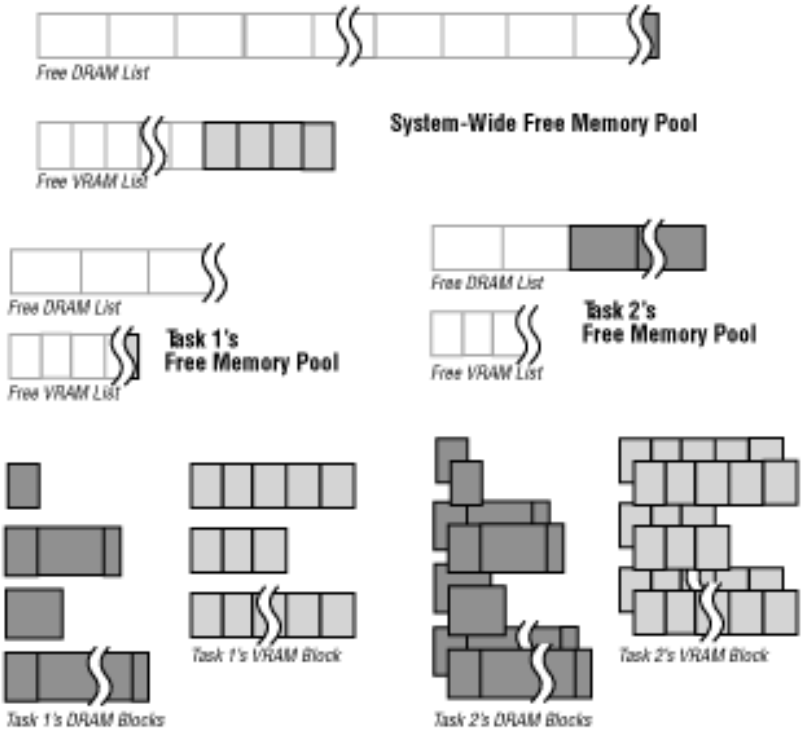

Figure 8: *The system wide free memory pool runs low on memory.*

Tasks Reclaim Memory and Return It to the Kernel

When the tasks try to allocate more memory, the system calls `ScavengeMem()` on the tasks behalf, as shown Figure 9.

`ScavengeMem()`returns any completely unused pages of memory in a task's free memory pool to the system-wide free memory pool, thereby making the pages available to other tasks.

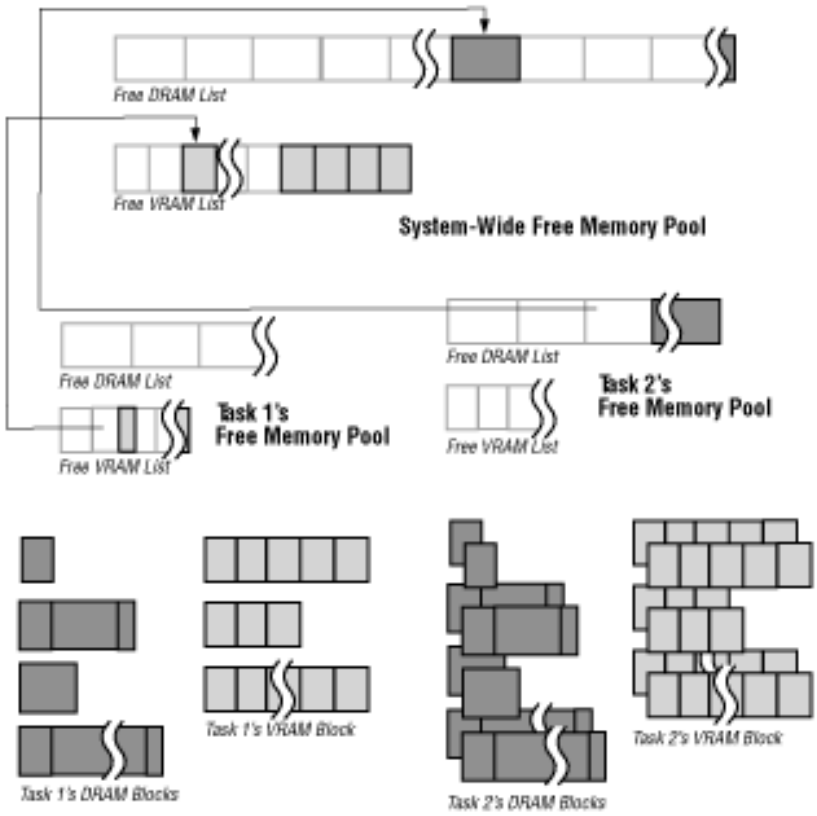

Figure 9: *Tasks scavenge memory in response to the kernel's signal.*

Guidelines for Using Memory

Remember the following guidelines when using memory:

- For each memory allocation function, there is a corresponding function to free the memory that was allocated and return it to the free memory list it came from. You must use this corresponding function to free memory. Table 1 lists the corresponding functions:

Table 1: *Corresponding memory allocation and memory freeing calls.*

Allocate Function	Corresponding Free Function
AllocMem()	FreeMem()
malloc()	free()
AllocMemList()	FreeMemList()
AllocMemFromMemList()	FreeMemToMemList()
AllocMemFromMemLists()	FreeMemToMemLists()

- Use 32-bit addressing for all memory. Future 3DO systems may have enough memory to need it.
- Don't tamper with the system structures (such as free memory lists) that control memory allocation, ownership, and write access. The ways these work are subject to change in future versions of Portfolio.
- Although tasks can write to each other's memory, tasks should normally use intertask communication to share information. Intertask communications are described in Communicating Among Tasks.
- Certain regions of the address space must not be accessed. Memory references must be restricted to addresses that have valid RAM. In addition, the lower page of RAM should never be accessed by applications.
- Use `memdebug` to identify problems with your memory usage. See Debugging Memory Usage for more information.

Allocating Memory

The following sections explain the easy way to allocate memory: by asking the kernel to do it. To learn how tasks can allocate their own memory from private memory lists, see Using Private Memory Lists.

Allocating a Memory Block

The normal way to allocate memory is with the `AllocMem()` macro:

```
void *AllocMem( int32 s, uint32 t )
```

The `s` argument specifies the size of the memory block to allocate, in bytes. The `t` argument contains memory allocation flags that specify the type of memory to allocate. You can include optional flags in the `t` argument that specify other characteristics of the block. The macro returns a pointer to the allocated block or NULL if the block couldn't be allocated.

Memory Block Flags

In the `t` argument, you must include one of the following flags to specify the type of memory to allocate:

- **MEMTYPE_ANY**. Allocates any memory that is available.
- **MEMTYPE_VRAM**. Allocates only video random-access memory (VRAM).
- **MEMTYPE_DRAM**. Allocates only dynamic random-access memory (DRAM).

If a block of VRAM must come from a specific VRAM bank, you must include the following flag to specify that bank:

- **MEMTYPE_BANKSELECT**, Allocates VRAM from a specific VRAM bank.

You must also include one of the following two VRAM bank selection flags:

- **MEMTYPE_BANK1.** Allocates only memory from VRAM bank 1.
- **MEMTYPE_BANK2**. Allocates only memory from VRAM bank 2.

The following flags are for compatibility with future hardware. You can set them in addition to the preceding flags.

- **MEMTYPE_DMA**. Allocates only the memory that is accessible via direct memory access

(DMA). Currently, all memory is accessible via DMA, but this may not be true in future hardware. Include this flag if you know the memory must be accessible via DMA.

- **MEMTYPE_CEL**. Allocates memory that is accessible only to the cel engine. Currently, all memory is accessible to the cel engine, but this may not be true in future hardware. Include this flag if you know the memory will be used for graphics.

- **MEMTYPE_AUDIO**. Allocates only memory that is used only for audio data (such as digitized sound). Currently, all memory can be used for audio, but this may not be true in future hardware. Include this flag if you know the memory will be used for audio data.

- **MEMTYPE_DSP**. Allocates only memory that is accessible to the digital signal processor (DSP). Currently, all memory is accessible to the DSP, but this may not be true in future hardware. Include this flag if you know the memory must be accessible to the DSP.

You can use the following optional flags to specify alignment (where the block is in relation to page boundaries), fill (the initial value of all memory locations in the block), and other allocation characteristics:

- **MEMTYPE_FILL**. Sets every byte in the memory block to the value of the lower-8 bits of the flags. If this flag is not set, the previous contents of the memory block are not changed. Using this flag is slower than not using it, so don't set this flag if you plan to initialize memory with your own data.

- **MEMTYPE_INPAGE**. Allocates a memory block that does not cross page boundaries.

- **MEMTYPE_SYSTEMPAGESIZE**. Allocates a memory block that starts on a memory protection page boundary as opposed to a SPORT page boundary.

- **MEMTYPE_STARTPAGE**. Allocates a memory block that starts on a page boundary. When allocating VRAM, this flag tells the system to allocate memory at the start of a SPORT page (2 KB boundaries) as opposed to a memory protection page. To always use the memory protection page size, you must also set the `MEMTYPE_SYSTEMPAGESIZE` flag.

- **MEMTYPE_MYPOOL**. Allocates a memory block from memory that is already in your task's free memory pool. This means that if there is not sufficient memory in the task's pool, the kernel will not allocate additional memory from the system-wide free memory pool.

- **MEMTYPE_FROMTOP**. Causes the allocation to come from the top of the task's free pool instead of from the bottom.

If there is insufficient memory in a task's free memory pool to allocate the requested memory, the kernel

automatically transfers the necessary pages of additional memory from the system-wide free memory pool to the task's free memory pool. The only exceptions are when there is not enough memory in both pools together to satisfy the request, or when the MEMTYPE_MYPOOL memory flag-which specifies that the memory block must be allocated only from the task's current free memory pool-is set.

Freeing a Memory Block

To free a memory block allocated with AllocMem(), use FreeMem():

```
void FreeMem( void *p, int32 size )
```

The p argument points to the memory block to free. The size argument specifies the size of the block to free, in bytes. The size you specify must always be the same as the size specified when you allocated the block.

Allocating Memory in Programs Ported From Other Platforms

If you're porting a program from another platform that uses the malloc() function from the standard C library to allocate memory, you can continue to use malloc():

```
void *malloc( int32 size )
```

The size argument specifies the size of block to allocate, in bytes. It returns a pointer to the block that was allocated, or NULL if the memory couldn't be allocated.

Because malloc() does not accept memory allocation flags, you cannot use it to specify a particular kind of memory or any other memory characteristics. If you are writing programs specifically for 3DO systems, you should use AllocMem() in place of malloc().

You must only use free() to free memory that you've allocated with malloc(). Each memory allocation function has a corresponding deallocation function; if you use a different memory allocation function, you must use its corresponding deallocation function. (See .)

Freeing Memory in Programs Ported From Other Platforms

To free a memory block you have allocated with malloc(), use free():

```
void free( void *p )
```

The `p` argument points to the memory block to free. The entire block is freed automatically; unlike `FreeMem(),` you do not need to specify the amount of memory to free.

Using Private Memory Lists

If you have dynamic components running and allocating memory, it may be necessary to limit the amount of memory certain threads can use. You can create a private memory for those threads, memory list. Those threads would only allocate memory from this list and since you can control the total amount of memory in the private memory list, you can control the maximum amount of memory that certain threads can allocate.

Creating a Private Memory List

You can create a private memory list by calling the `AllocMemList()` function:

```
MemList *AllocMemList( const void *p, const char *name )
```

The `p` argument is a pointer to a memory address of the type of memory (either DRAM or VRAM) you want to store in the list. A single memory list can store either DRAM or VRAM, but not both. You use the `name` argument to name the memory list. The return value is a pointer to the resulting MemList structure, or NULL if an error occurred.

Adding Memory to a Private Memory List

Initially, the memory list you create with `AllocMemList()` is empty. To add memory to the list, use the `FreeMemToMemList()` function:

```
void FreeMemToMemList( MemList *ml, void *p, int32 size )
```

The `ml` argument is a pointer to the memory list from which to free the memory. The `p` argument is a pointer to the memory to free. The memory block must be properly aligned in position and size for the type of memory being manipulated. The alignment granularity can be determined with `GetMemAllocAlignment()`. The number returned by this function indicates the alignment needed for the pointer; the size of the memory block being freed must also be a multiple of this value. Your task must already own this memory. The type of the memory to free must be the same as the type specified in the p argument of `AllocMemList()`. The `size` argument specifies the amount of memory to move the free list, in bytes. You can move a block of any size to the free list.

Allocating Memory From a Private Memory List

Once you've created a private memory list and put memory into it, you can allocate memory from the list with the `AllocMemFromMemList()` function:

```
void *AllocMemFromMemList( MemList *ml, int32 size, uint32
memFlags )
```

The `ml` argument is a pointer to the memory list from which to allocate the memory. The `size` argument specifies the size of the block to allocate, in bytes. The `memtype` argument contains flags that specify the type of memory to allocate, the alignment of the block with respect to the page, and so on; these flags can include `MEMTYPE_ANY`, `MEMTYPE_VRAM`, `MEMTYPE_DRAM`, `MEMTYPE_BANKSELECT`, `MEMTYPE_BANK1`, `MEMTYPE_BANK2`, `MEMTYPE_DMA`, `MEMTYPE_CEL`, `MEMTYPE_AUDIO`, `MEMTYPE_DSP`, `MEMTYPE_FILL`, `MEMTYPE_INPAGE`, `MEMTYPE_STARTPAGE`, `MEMTYPE_SYSTEMPAGESIZE`, and `MEMTYPE_MYPOOL`. Note that the flags that specify the actual memory type (such as `MEMTYPE_DRAM`) must match the type of the memory contained in the list. For complete descriptions of these flags, see Allocating a Memory Block. The function returns a pointer to the block that was allocated or NULL if there was not enough memory in the list to satisfy the request.

To free memory that you've allocated with `AllocMemFromMemList()`, you must use only `FreeMemToMemList()` (described in the next section). Each memory allocation function has a corresponding deallocation function; if you use another memory allocation function, you must use its corresponding deallocation function.

Freeing Memory to a Private Memory List

To free a block of memory that you've allocated from a private memory list and return it to the list, you use the `FreeMemToMemList()` function:

```
void FreeMemToMemList( MemList *ml, void *p, int32 size )
```

The `ml` argument is a pointer to the memory list from which the block was allocated. The `p` argument is a pointer to the block to free. The `size` argument specifies the size of the block to free, in bytes.

Deallocating a Private Memory List

To deallocate a private memory list, you first empty it (using `ControlMem()` to return the memory pages to the kernel, as described in Transferring Memory to Other Tasks) and then use the `FreeMemList()` function:

```
void FreeMemList( MemList *ml )
```

The `ml` argument is a pointer to the memory list to deallocate. If the list is not empty, any memory it contains is lost.

Sharing Private Memory Lists Among Threads

To use a single memory list from multiple independent threads, there must be extra protection so the threads don't corrupt the shared data structure. To share a memory list, you must attach a semaphore item to the MemList structure. This is done by creating a semaphore item and putting its Item number in the `meml_Sema4` field of the MemList structure.

Transferring Memory to Other Tasks

The previous section explained how to use `ControlMem()` to grant and revoke write access to memory pages. You can also use `ControlMem()` to transfer ownership of memory pages to other tasks:

```
Err ControlMem( void *p, int32 size, int32 cmd, Item task )
```

If the value of the `cmd` argument is `MEMC_GIVE`, the call gives the memory pages to the task specified by the task argument. In this case, the `p` argument, a pointer to a memory location, and the `size` argument, the amount of the contiguous memory, in bytes, beginning at the memory location specified by the `p` argument, together specify the memory to give away. If these two arguments specify any part of a page, the entire page is given away.

You can also give memory pages back to the kernel-and thereby return them to the system-wide free memory pool-by calling `ControlMem()` as described here, but with 0 as the value of the task argument. Memory pages can be returned to the kernel automatically when a task calls the `ScavengeMem()` function described in Reclaiming Memory for Other Tasks.

Reclaiming Memory for Other Tasks

When you free a large chunk of memory, it is good practice to return the pages to the system-wide free memory pool with `ScavengeMem()`, so other tasks can use them:

```
int32 ScavengeMem( void )
```

This function finds pages of memory from which no memory is allocated in the task's free memory pool and gives those pages back to the system-wide free memory pool. The function returns the amount of memory that was returned to the system-wide memory pool, in bytes, or it returns 0 if no memory was returned.

The Filesystem and the File Folio

This chapter describes the 3DO file system. It includes sample code that illustrates how to use File folio calls. This chapter contains the following topics:

3DO File System Interface

The 3DO file system is, in essence, a high-level device driver that interposes itself between an application and the "raw" disk device. The application can issue a call to open a specific disk file and will receive in return an item for a device node. This device item (referred to internally as an "open file") can be treated like most other device items-you can create one or more IOReq items for the device item, and send these IOReqs to the device to read data from the file.

The file system architecture is implemented as a combination of the following elements:

- The **File folio** that provides high-level functions to interface to the lower-level components.
- The **driver** that manages the file system and open file devices.
- The **daemon** task that manages I/O scheduling.

The 3DO filesystem is an intrinsic part of Portfolio and starts automatically during system start up.

Design Philosophy

The 3DO filesystem structures were designed with a number of factors:

- Most 3DO file systems reside on CD-ROM devices.
- CD-ROM drives are slow, with seek times measured in hundreds of milliseconds. Minimizing the amount of head-seeking required during game play is critical to the responsiveness of the system.
- 3DO CD-ROMs tend to be handled and stored under less-than-ideal conditions. They can be subject to accidental damage during or between uses.

For these reasons, the 3DO file system supports data replication, in a way that is transparent to the application and to the user (except for a possible delay, if one copy of the data cannot be read and the CD-ROM mechanism must seek another copy). The file driver prioritizes, sorts, and optimizes I/O requests to the file system device to minimize the total seek-and-read time for individual requests and for groups of requests.

Files

Files are collections of blocks, each block containing a fixed number of bytes. The block size of a file is determined generally by the block or sector size of the device on which it resides. For example, on a CD-

ROM file system, most files have a block size of 2,048 bytes.

The file system deals with files exclusively in terms of blocks. However, it does maintain a logical size in bytes for each file. This logical size indicates the number of bytes actually used within the blocks allocated for the file.

The File folio provides a number of FileStream functions that offer a stream-oriented interface to the file system. Using these functions, you can read and seek in files in a manner that is similar to the C stdio routines. These functions are described in Byte-Stream File Access.

Directories

The 3DO-native file system implements directories as ordinary disk files that are flagged as a specific type. The directories can be opened as normal files using the `OpenDiskFile()` function. To read the contents of a directory, you must use the `OpenDirectory()` and `ReadDirectory()` functions.

Avatars

The 3DO CD-ROM file system supports the unique concept of avatars. Each file consists of one or more avatars, placed in various locations across the disk. Each avatar is an exact, coequal image of the data in the file.

How Avatars Work

The file driver keeps track of the current position of the device's read head. When asked to read data, it automatically chooses the "closest avatar in good condition" of each block of data it is asked to read. If it must perform more than one read, the driver sorts the read requests to minimize the amount of head-seeking. It honors the I/O request priorities during this process-completing all requests of one priority before scheduling any requests of a lower priority.

If the device has trouble reading a data block contained within one avatar of a file, the file driver marks that avatar as flawed and reissues the read request-thus searching out another avatar on the disk, if one exists.

It is not necessary, nor is it desirable, for all files in a 3DO file system to have the same number of avatars. Critical directories, or files accessed frequently throughout the execution of the application, can have as many as a dozen avatars scattered across the disk. Large files, or those containing noncritical data, may have only one avatar.

Placement of a File's Avatars

The placement of a file's avatars can have a great effect on the performance of a Portfolio game application. If your application opens and reads several files in sequence, and the avatars lie some distance away from one another on the CD-ROM, the CD-ROM drive must *seek* from one file to another; your program must wait while it does so. However, if the avatars are located close together, ideally, right next to one another, the CD-ROM drive needs to seek less, and perhaps not at all. Every seek on a CD-ROM drive takes tens to hundreds of milliseconds; it's important to minimize the number of seeks for your application to perform well.

The software tool that lays out a Portfolio CD-ROM can optimize the avatar locations on your files in an effective, semiautomatic fashion. It can create two or more avatars of frequently used files, to reduce the distance needed to seek to the file. It can even weave portions of different files together into a single accelerated-access file called *Catapult*, eliminating most of the seeks that occur during the system startup process.

The layout optimization and multiple-avatar speedups, and the Catapult acceleration, are completely transparent to your application. You do not need to change any application code to take advantage of them. They do not change the functionality of the filesystem, they simply improve its performance.

Instructions on doing an optimized CD-ROM layout with Catapult acceleration can be found in the *3DO CD-ROM Mastering Guide*. You should do an optimized and Catapult-accelerated layout of your application before you submit it to The 3DO Company for encryption. If you submit a nonoptimized CD-ROM image for encryption, the company may ask you to optimize and resubmit it.

Byte-Stream File Access

A set of function calls in the File folio eases the job of accessing the contents of a file in a byte-by-byte fashion, rather than in the block-by-block mode supported by the file driver interface. These functions include:

- `OpenDiskStream()`
- `SeekDiskStream()`
- `ReadDiskStream()`
- `CloseDiskStream()`

`With these functions, a program can read an arbitrary number of bytes from anywhere within a file, without worrying about block boundaries.`

Opening a File

Before you can read a file's data using the streaming routines, you must open the file:

```
Stream *OpenDiskStream( char *theName, int32 bSize )
```

`OpenDiskStream()` opens the file identified by an absolute or relative pathname, allocates the specified amount of buffer space, and initiates an asynchronous read to fill the buffer with the first portion of the file's data. It returns NULL if any of these functions cannot be performed for any reason.

Reading the Data

To read data from a stream file, you must call `ReadDiskStream()`

```
int32 ReadDiskStream( Stream *theStream, char *buffer, int32
nBytes )
```

`ReadDiskStream()` reads data from the given stream file starting at the current position within the file. The data is put into the supplied buffer. The number of bytes actually read from the file is returned. The position within the file is advanced by the number of bytes read, so that the next time `ReadDiskStream()` is called, the data that follows is read.

Setting Up the Read Position

To change the current position within a stream file, you must call:

```
SeekDiskStream( Stream *thream, int32 offset, enum SeekOrigin
whence )
```

After you call `SeekDiskStream()`, data transfers start with the first byte at the new file position. The `whence` argument specifies whether the operation is relative to either the beginning or end of the file, or to the current position in the file. The `offset` argument specifies the number of bytes offset relative to the `whence` position. It returns the actual absolute file position that results from the seek position or a negative error code if unsuccessful.

The offset can be specified in any of three ways: absolute (positive) byte offset from the beginning of file (`SEEK_SET`), relative byte offset from the current position in the file (`SEEK_CUR`), or absolute (negative) byte offset from the end of the file (`SEEK_END`).

Cleaning Up

`CloseDiskStream()` closes the stream's file, deallocates the buffer memory, and releases the stream structure.

```
void CloseDiskStream(Stream *theStream);
```

Pathnames and Filenames

Gaining access to files is done by describing the files using pathnames. Pathnames provide a description of the location of files within a file system hiearchy. The 3DO file system uses pathnames akin to those used in UNIX. An absolute pathname is of the form:

```
/file system/dir1/dir2/dir3/(more)/dirN/filename
```

Each task in Portfolio has a current directory associated with it. Relative pathnames describe a location relative to a known directory. An absolute pathname always starts with a backslash (/), while a relative pathname never does.

A relative pathname can be in one of three forms:

```
filename dir/filename dir1/dir2/(more)/dirN/filename
```

Relative pathnames can be used in either of two ways:

- In the `OpenDiskFile()` call, pathnames specify a path relative to the current task's current directory.
- In the `OpenDiskFileInDir()` call, pathnames specify a path relative to a directory whose item is specified in the call.

In addition, three special conventions are used to further describe the 3DO directory structure. A path component of a full stop (.) indicates the current directory. A path component of two full stops (..) indicates the parent directory. Finally, a caret (^) indicates the root of the file system.

Each pathname component should not be longer than 31 characters. In addition, certain characters are not allowed in component names. These illegal characters are backslash (/), dollar sign ($), left bracket ({), right bracket (}), and a pipe (|).

Accessing Directories

Four functions in the File folio provide access to directories and their contents. These functions can enumerate the contents of directories or obtain information about all the currently mounted file systems.

- `OpenDirectoryPath()`
- `OpenDirectoryItem()`
- `ReadDirectory()`
- `CloseDirectory()`

Opening the Directory

You can open a directory to read by specifying the directory's pathname or the directory's item number. `OpenDirectoryPathName()` opens a directory by specifying its pathname.

```
Directory *OpenDirectoryPathName( char *thePath )
```

This function opens a directory, allocates a new Directory structure, and prepares for a traversal of the contents of the directory. It returns a pointer to a Directory structure that or NULL if an error occurs.

`OpenDirectoryItem()` opens a directory by referencing its item number.

```
Directory *OpenDirectoryItem( Item openFileItem )
```

It allocates a new Directory structure, opens the directory, and prepares for a traversal of the contents of the directory. `OpenDirectoryItem()` returns a pointer to the Directory structure, or NULL if an error occurs. Typically, you obtain the item number of a directory by calling `OpenDiskFile()` on the directory, or by calling `GetDirectory()`.

Reading the Directory

`ReadDirectory()` reads the next entry from the specified directory.

```
int32 ReadDirectory( Directory *dir, DirectoryEntry *de )
```

This function gets information about the next directory entry, and deposits this information in the supplied DirectoryEntry structure. The contents of this structure can then be examined to get details about the entry. The function returns an error code if all entries in the directory have been processed.

Cleaning Up

`CloseDirectory()` closes a directory that was previously opened using `OpenDirectoryItem()` or `OpenDirectoryPath()`.

```
void CloseDirectory( Directory *dir )
```

All resources get released.

Finding Mounted File Systems

You can obtain a list of the mounted file systems by scanning the "/" directory. If you open this directory and call `ReadDirectory()` on it, you iterate through all of the currently mounted file system. The DirectoryEntry structure will then hold information about the file system.

The Current Directory

Each task in the Portfolio environment has a current directory associated with it. The current directory is the starting location for relative pathnames that are used with various File folio calls which take a pathname as an argument.

When an application is loaded from a file, it automatically gets its current directory set to where the program file is located. This makes it easier for the new task to find any files it needs.

The File folio provides two calls, `GetDirectory()` and `ChangeDirectory(),` to determine and change the location of the current directory of the current task.

Finding the Current Directory

`GetDirectory()` returns the item number of the current directory of the calling task.

```
Item GetDirectory( char *pathBuf, int32 pathBufLen )
```

If `pathBuf` is non-NULL, it points to a buffer of writable memory whose length is given in `pathBufLen`; the absolute pathname of the current working directory is stored into this buffer.

Changing the Current Directory

`ChangeDirectory()` changes the current directory of the current task to the absolute or relative location specified by the path.

```
Item ChangeDirectory( char *path )
```

The function returns the item number of the new directory, or a negative error code if an error occurs.

Loading and Executing an Application

The File folio provides six functions to load and run executable code:

- `LoadProgram()`
- `LoadProgramPrio()`
- `LoadCode()`
- `UnloadCode()`
- `ExecuteAsSubroutine()`
- `ExecuteAsThread()`

Loading and Launching a Task

`LoadProgram()` loads an executable file from disk and launches a new task which executes the code in the file.

```
Item LoadProgram(char *cmdLine);
```

The only argument of this function is a command line to interpret. The first component of the command line is taken as the name of the file to load. The entire command line is passed to the new task as `argc` and `argv` in the `main()` function. The file name component of the command line specifies either a fully qualified pathname, or a pathname relative to the current directory.

The priority of the new task is the same as the priority of the current task. If the task should have a different priority, use the `LoadProgramPrio()` function.

`LoadProgramPrio()` is identical to `LoadProgram()` except that it specifies a task priority for the new task.

```
Item LoadProgramPrio(char *cmdLine, int32 priority);
```

You give this function a command line to interpret. The first component of the command line is taken as the name of the file to load. Like `LoadProgram()`, the entire command line is passed to the new task as `argc` and `argv` in the `main()` function. The file name component of the command line specifies either a fully qualified pathname, or a pathname relative to the current directory.

The `priority` argument specifies the task priority for the new task. If you simply want the new task to have the same priority as the current task, use the `LoadProgram()` function instead. Alternatively, passing a negative priority to this function will also give the new task the same priority as the current

task. `LoadProgram()` and `LoadProgramPrio()` will set the current directory of the newly created task to the directory from which the executable was loaded.

Loading a Code Module

`LoadCode()` loads an executable file from disk into memory. Once loaded, the code can be spawned as a thread, or executed as a subroutine.

```
Err LoadCode(char *fileName, CodeHandle *code);
```

In order to work correctly with this and associated functions, the executable file being loaded must have been linked with *threadstartup.o* or *subroutinestartup.o* instead of *cstartup.o*

This function requires the name of the executable file to load, as well as a pointer to a `CodeHandle` variable, where the handle for the loaded code will be stored.

`Note:` `code` must point to a valid `CodeHandle` variable, because LoadCode() uses this location to put a pointer to the loaded code.

Executing a Loaded Code Module

To execute the loaded code, you must call either the `ExecuteAsThread()` function or the `ExecuteAsSubroutine()` function. If the loaded code is reentrant, the same loaded code can be spawned multiple times simultaneously as a thread.

`ExecuteAsSubroutine()` executes a chunk of code that was previously loaded from disk using `LoadCode()`.

```
int32 ExecuteAsSubroutine(CodeHandle code, int32 argc, char
**argv);
```

This function runs the code as a subroutine of the current task or thread.To work correctly, code that is run as a subroutine should be linked with *subroutinestartup.o* instead of the usual *cstartup.o*

The parameters in `argc` and `argv` are passed directly to the `main()` entry point of the loaded code. The return value of this function is the value returned by `main()` of the code being run.

The values you supply for `argc` and `argv` are irrelevant to this function. They are simply passed through to the loaded code. Therefore, their meaning must be agreed upon by the caller of this function, and by the loaded code.

`ExecuteAsThread()` executes a chunk of code that was previously loaded from disk using `LoadCode()`. This function executes code that will execute as a thread of the current task.

```
Item ExecuteAsThread(CodeHandle code, int32 argc, char **argv,
char *threadName, int32 priority);
```

To work correctly, code being run as a thread should be linked with *threadstartup.o* instead of the usual *cstartup.o*

The `argc` and `argv` parameters are passed directly to the `main()` entry point of the loaded code.The values you supply for `argc` and `argv` are irrelevant to this function. They are simply passed through to the loaded code. Therefore, their meaning must be agreed upon by the caller of this function, and by the loaded code. `threadName` specifies the name of the thread. `priority` specifies the priority the new thread should have. Providing a negative priority makes the thread inherit the priority of the current task or thread.

Unloading a Code Module

Once you are done using the loaded code, you can remove it from memory by passing the code handle to the `UnloadCode()` function. `UnloadCode()` frees any resources allocated by `LoadCode()`. Once `UnloadCode()` has been called, the code handle supplied becomes invalid and cannot be used again.

Working With NVRAM

The Portfolio operating system treats NVRAM-nonvolatile random access memory-as a file system volume. This maintains consistency for I/O operations; you use the same steps for accessing NVRAM as you do for all other files. Portfolio also provides an NVRAM maintenance utility called lmadm. See the *3DO Debugger Programmer's Guide*, "Terminal Window Commands," for full details.

A minimally configured 3DO system has at least 32 KB of NVRAM. Application developers can use this facility for persistent storage of small pieces of data. Practical uses include providing a game saving feature, to save user preference settings, or to store application configuration information. There is no way to query the operating system for the amount of available NVRAM memory prior to creating an NVRAM file, because NVRAM is not confined to one contiguous area. The correct procedure is to try to create and allocate the NVRAM file using the `OpenDiskFile()` call. If the call is successful, then sufficient memory was available. If the call fails for lack of memory, then it returns an error code indicating insufficient space.

The following sections explain how to access NVRAM. They include descriptions of high-level NVRAM access routines.

How Not to Manage NVRAM

When NVRAM is full, some titles cannot manage VRAM appropriately. Some titles fail silently, discarding information the user wants to save. Other titles overwrite information saved by other titles. This can be annoying if a user reached a relatively high level in another title.

Caution: Developers should avoid managing NVRAM in the ways mentioned above.

How to Manage NVRAM

A title should do the following to manage NVRAM properly:

- Remove obsolete files. For example, files for a previous version of the same title can probably be removed.
- If possible, save files as compressed files.
- Never delete information about a different title without prompting the user.

- Prompt the user to delete information currently in NVRAM if a title attempts to write to NVRAM but fails because it is full. This can be done using Access, discussed in Access: A Simple User Interface. Some guidelines are discussed in the next section.

How a Title Should Save Its Status

Every title should go through the same set of steps to save its status. This section lists the steps in sequence. Commented sample code for parts of this sequence is provided in the 3DO Toolkit 1.3.1.

1. Bring up the Save Title display. This display differs from title to title. Users can usually choose to overwrite files or add to a list of existing files.

2. If the user cancels, exit with no error.

3. Otherwise, attempt to write file.

4. If writing file works, exit with no error.

5. Otherwise, delete the file that was just saved. (This is important in order to remove any parts of the file that might have made it to NVRAM.) Then inform the user that there is not enough room to save the game, and bring up the Delete File display.

6. If the user chooses a file to delete, remove the file and go back to step 3 above.

   Otherwise, if the user cancels-return to Save Title display and go back to step 1.

   Note: A sample program that brings up a Delete File display is included in this release.

Working With NVRAM Files

Portfolio supports NVRAM access through file system function calls. Your title can use I/O requests to work with an open NVRAM file as a device, as it would any other file. You can use the functions in the section Using NVRAM to create (and allocate) blocks, and to read from, write to, get the block size of a file, and set the end of a file.

This section first provides information on how to choose appropriate filenames, then provides some information about how to look at NVRAM using the 3DO Debugger.

Choosing Appropriate Filenames

The NVRAM volume is a flat file system; it cannot contain subdirectories. Choose filenames wisely to avoid conflicts with other title developers. To avoid name conflicts while still providing readable filenames to end users, 3DO recommends the following file-naming standard. Filenames in NVRAM have a limit of 31 characters. Filenames that follow this standard consist of two parts:

<title name> [<user/level name>]

- The first part is the name of the title itself. If space allows you should use the full name of your title, with no abbreviations.

- The second part of the filename uniquely identifies the file to the user. This part is flexible depending on the purpose of the file. For example, if the file is a game and users are prompted for their name as part of the game, the users' name or the name of the game level could be part of the filename.

Note: When a title displays a Load Game or a Save Game display, it should only show its own files within the list. That is, when loading a saved game from Total Eclipse, for example, a player shouldn't be able to see games from Monster Manor in the file list. In addition, the entries displayed in that list should not include the title's name.

Following this standard reduces the chance of filename conflicts in NVRAM. However, it still does not completely eliminate the possibility of confusion between titles that create files with similar or identical names. Therefore, it is recommended that you store a unique identifier in the first two bytes of every file that your title creates in NVRAM. To guarantee uniqueness, this identifier should be the licensee number assigned to you by 3DO.

Working With NVRAM via the 3DO Debugger

As with any other volume, NVRAM must be formatted properly before the file system can access it. If you're working from the debugger, type the following command, followed by the Enter key, in the debugger Terminal window:

```
format RAM 3 0 NVRAM
```

After issuing this command, reset the target. You should see the */NVRAM* volume mount during file system start-up. Thereafter, you can refer to this volume as */NVRAM*. You will not need to perform this format ritual again, since the contents of NVRAM are preserved across restarts and shutdowns. On a production player NVRAM is preformatted.

If you're working from a CD-ROM image, the volume is formatted automatically.

Working With NVRAM Files From the Debugger

You can work with files in NVRAM the same way you'd work with other files. Copy them using the Copy command. To view them, type into the Terminal window:

```
$c/ls /nvram
```

A list of all files appears in NVRAM. To see a more detailed list, type:

```
$c/lmdump RAM 3 0
```

NVRAM Chunk Definition

This section describes a format for files in NVRAM. The format lets you embed information about a file inside the file itself. As a result, the Storage Manager or other software can present users with information about the files in NVRAM.

This section first lists file type and chunk information, then presents a new chunk, the comment chunk, and an example.

File type and chunk information

The files must have a special type, 3DOF, which stands for 3DO file format. This must be set for applications to know that the file is in the correct format.

A file which is in the 3DO file format consists of one or more chunks. Each chunk contains a chunk ID, chunk size, and chunk data. For more information about the 3DO file format see the *3DO CD-ROM Mastering Guide*.

The comment chunk

The comment chunk contains information about the file in NVRAM.

Table 1: *Components of comment chunk.*

Name	Descriptions
ID	NVRT
size	Length of the language data + 4

```
                    |bytes for the language code
-------------------------------------------------------
chunk data          |Language code and the comment
                    |with a null at the end
-------------------------------------------------------
language code       |0 for English
-------------------------------------------------------
```

For example, a comment chunk might look like this:

```
`NVRT',        20,        0,        "The 3DO Company"
```

Access: A Simple User Interface

This chapter shows how a task uses Access to ask for and retrieve input from a 3DO user. This chapter contains the following topics:

How Access Works

Many times a task needs to present information to or ask for input from the user. For example, a task may need the file name of a file to load, save, or delete, or ask the user which of two choices to make.

Access is a Portfolio system task that provides a mechanism for displaying this type of information to the user and accepting user input. Access can create transaction boxes, display specified text, read controller input, and perform all the other user input activities.

To use Access, a task communicates its user interface needs by sending messages to the Access message port. The calling task receives results by receiving reply messages from Access. For example, a task sends a message to Access requesting it to ask the user to load a file. Access presents the appropriate transaction box, accepts the user's input (via the control pad), and sends the name of the file back to the task.

Access provides two benefits to calling tasks:

- First, the task doesn't have to create a user interface from scratch. Access creates the transaction boxes for user input and monitors controllers to get the response to the transaction boxes, all according to simple requests from the calling task.

- Second, Access provides a consistent user interface for common activities found in many tasks. For example, once a user learns how to load a file in one program that uses Access, he or she doesn't have to learn a different process to load a file in another program that uses Access.

How to Use Access

Using Access is simple. The calling task must follow these steps, which are explained in detail in the sections that follow:

1. Create a message to send to Access and a reply port to receive a reply to the message.
2. Find the Access message port.
3. Create a list of tag arguments that request a specific user-interface transaction and describe the properties of the transaction box that carries out the request.
4. Send a message to Access that points to the tag argument list.
5. Enter the wait state and wait for a reply from access.
6. Read the returned user input from Access's reply message.

The trick is in setting up the tag arguments to specify the interface transaction with the desired transaction box layout.

Setting Up Message Communication With Access

Before a task can send tag argument lists to Access, it must first create a message item and a message port as described in Communicating Among Tasks. Then, it must then find Access's message port, "access," using the `FindMsgPort()` call. The following code segment returns the item number of Access's message port:

```
FindMsgPort("access");
```

Creating a Tag Argument List

Once a task is set up to communicate with Access, it must create a tag argument list to carry tag arguments to Access. The tag arguments that Access reads are the same tag argument data types used to specify item attributes with the `CreateItem()` call. For a list of the tag argument data types, see Managing Items. Each tag argument has two fields: the tag and the argument value. You need only provide tag arguments for the attributes you wish to specify; all nonspecified attributes are set to default values. Two tag arguments are mandatory: `ACCTAG_REQUEST` and `ACCTAG_SCREEN`. The final tag

argument in the list must be set to TAG_END to terminate the list.

All Access tags and the constants used to set their argument values are found in the include file *access.h*.

Specifying a Request

The first order of business is to request the type of user-input transaction the task wants Access to perform. To specify, the task uses the tag ACCTAG_REQUEST, which can accept any one of the following arguments:

- ACCREQ_LOAD displays a transaction box that asks the user to specify a file to be loaded. The transaction returns the pathname of a file to be loaded.
- ACCREQ_SAVE displays a transaction box that asks the user to specify a file to save. The transaction returns the pathname of a file to save.
- ACCREQ_DELETE displays a transaction box that asks the user to specify a file to delete. The transaction returns the pathname of a file to delete.
- ACCREQ_OK displays a transaction box with text supplied by the calling task to inform the user of a fact. The transaction box contains a single OK button, which the user can click to continue. The transaction returns a 0.
- ACCREQ_OKCANCEL displays a transaction box with text supplied by the calling task to inform the user of a proposed action. The transaction box contains two buttons, OK and Cancel, which the user can click on either to confirm the action or cancel it. The transaction returns a 0 for the OK button or a 1 for the Cancel button.
- ACCREQ_ONEBUTTON displays a transaction box with text supplied by the calling task to inform the user. The transaction box has a single button with text also supplied by the calling task. The user can click on it to continue. The transaction returns a 0.
- ACCREQ_TWOBUTTON displays a transaction box with text supplied by the calling task to inform the user. The transaction box has two buttons, each with text supplied by the calling task. The user can click on either button to choose between the two choices. The transaction returns a 0 for the left button or a 1 for the right button.

When the calling task requests a specific user interface transaction, it must supply the necessary information for carrying out the transaction. For example, if it specifies ACCREQ_TWOBUTTON, it must supply text for both of the buttons in the transaction box and background text to explain the context of the buttons.

Specifying a Screen

When a task requests a user interface transaction through Access, it must specify a screen where the transaction box appears. To do so, the task provides the tag argument `ACCTAG_SCREEN`, which takes the item number of the screen as its argument.

Note: Before you call Access, you must call `ControlMem()` so that Access can write to the screen's memory.

Specifying Transaction Box Text

When Access puts a transaction box up on a screen, it can give the box a title that appears in its upper-left corner, fill the center of the box with explanatory text, and specify text for custom buttons as shown in Figure 1.

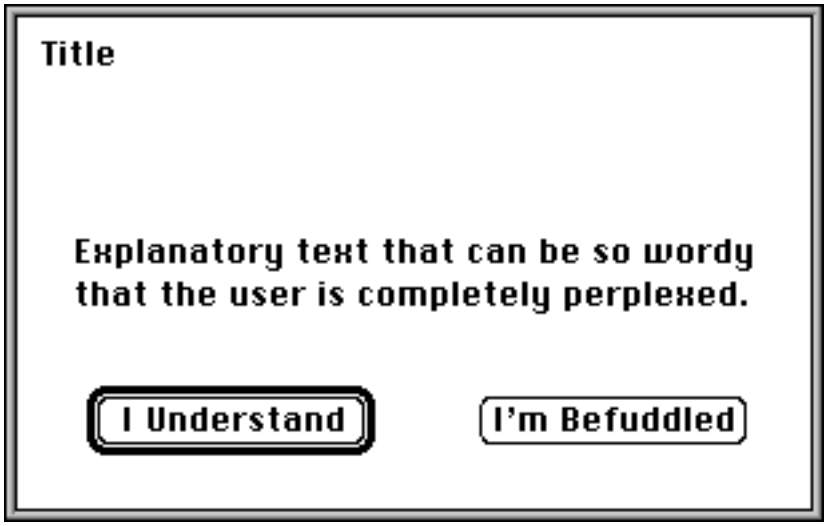

Figure 1: *An Access transaction box.*

To specify a title for the transaction box, a task uses the tag argument `ACCTAG_TITLE`, which takes as its argument a pointer to a NULL-terminated text string to be used as the title.

To specify text for the center of the transaction box, a task uses the tag argument `ACCTAG_TEXT`, which takes as its argument a pointer to a NULL-terminated text string to be used as explanatory text. Access automatically wraps text within the transaction box.

To specify text for the first button (the left button), a task uses the tag argument `ACCTAG_BUTTON_ONE`; to specify text for the second button (the right button), it uses the tag `ACCTAG_BUTTON_TWO`. Both tag arguments accept a pointer to a NULL-terminated text strings containing the text to be inserted in the button.

Keep in mind that if you supply too much text for a button or the title, you may overflow the display.

Preserving the Screen Background

When Access presents a transaction box in a screen, it can overwrite the pixels beneath the transaction box without restoring them (which leaves a big blank rectangle when the transaction box closes), or it can save the pixels and restore them after closing the transaction box. To choose screen background preservation, a task sends the tag `ACCTAG_SAVE_BACK` with the argument 0.

Preserving the screen background requires a fair amount of memory. To be memory efficient, you can choose not to preserve the screen background by not sending the `ACCTAG_SAVE_BACK` tag argument at all.

Creating a String Buffer for User Input Return

When an Access transaction box returns user input in the form of a text string, Access returns the string by storing it in a buffer and then pointing to the buffer in its reply message. The task requesting the transaction must supply the string buffer, allocating memory if necessary and then pointing to it and giving its size. To point to a buffer, a task uses the tag argument `ACCTAG_STRINGBUF`, which takes a pointer to the beginning address of the string buffer as its argument. It must also supply the tag argument `ACCTAG_STRINGBUF_SIZE`, which takes the size in bytes of the string buffer as its argument.

Note: Before you call Access, you must call `ControlMem()` so that Access can write to the buffers.

Setting Transaction Box Colors

When Access presents a transaction box, it uses four colors:

- The **background color** fills the interior of the box and the interior of any unselected buttons.
- The **foreground color** forms the characters in all text within the transaction box.
- The **highlight color** fills the interior of any selected button.
- The **shadow color** forms the outline of buttons in the transaction box.

To set those colors, a task uses these four tag arguments:

- `ACCTAG_FG_PEN` sets the foreground color.
- `ACCTAG_BG_PEN` sets the background color.
- `ACCTAG_HILITE_PEN` sets the highlight color.
- `ACCTAG_SHADOW_PEN` sets the shadow color.

Each of these tags takes as an argument a pen color value, which is a 15-bit 3DO RGB value stored in the low 15-bits of a 32-bit unsigned integer. (The upper 17 bits are set to 0.)

Sending a Message to Access and Waiting for a Reply

Once the tag argument list for a transaction is created, the task requesting the transaction sends a message to Access. The message points to the first tag argument in the list and gives the size of the buffer used to store the tag arguments in bytes. Access receives the message, reads the tag arguments to define a transaction, and puts up the appropriate transaction box to query the user.

In the meantime, the task that requested the transaction should entering the wait state to wait for a reply message from Access. It can do so using the `WaitPort()` call. When Access is finished with the transaction, it sends a reply to the task's reply port, which wakes up the task. The task can then read the contents of Access's reply message.

Reading a Reply Message

Access's reply message to a task requesting a transaction contains the field `msg_Result`, which contains the results of the transaction. If an error occurs, it contains a negative value (an error code defined in *access.h*). If the transaction was successful, `msg_Result` contains a 0 or a 1. These values can be interpreted in different ways depending on the button content of the transaction box. If the box contains a single button, `msg_Result` contains 0 if the button was clicked. If the box contains a Cancel and an OK button, it contains 0 for OK and 1 for Cancel. If the box contains two task-labeled buttons, it contains 0 for the left button and 1 for the right button.

Communicating Among Tasks

This chapter gives you the background and necessary programming details to perform intertask communication. Kernel Folio Calls in the *3DO System Programmer's Reference* contains more specific information about the calls used in this chapter.

This chapter contains the following topics:

About Intertask Communication

Two mechanisms can be used for intertask communication: signals and messages. Signals are used as an intertask flag with which one task can notify another that something has occurred. Messages not only allow one task to notify another, but also carry data from the sending task to the receiving task.

Signals and a task's ability to wait for a signal are the basis for all intertask communication. The message mechanism, which passes more detailed information, is built on top of the signal mechanism. Sending and receiving a message is based on a signal sent to the receiving task, informing it that a message is waiting.

Using Signals

The procedure for using a signal involves several steps:

- Task A uses `AllocSignal()` to allocate a signal bit (1 bit within a 32-bit word) for a signal.
- Task A communicates the allocated signal bit to Task B so that it knows which signal to send (using the allocated signal bit) back to Task A.
- Task A uses `WaitSignal()` to enter a wait state until it receives a signal on the bit it allocated.
- Task B uses `SendSignal()` to send a signal to Task A. Task B must provide `SendSignal()` with the item of that task to signal, and the signals to send the task.
- Task A receives the signal and returns from the wait state.
- Task A uses `FreeSignal()` to free the signal bit when it's no longer needed.

Allocating a Signal Bit

To allocate a signal bit for a task, use this call:

```
int32 AllocSignal( uint32 sigMask )
```

`AllocSignal()` accepts as its sole argument a 32-bit word (`sigMask`) which specifies which signal bits to allocate. There are 31 signal bits for each task. The lower-8 bits (bits 0-7) are reserved for system use and cannot be allocated. Bits 8-30 can be allocated for task use. To specify a particular signal bit to allocate, just set the corresponding bit in the `sigMask` parameter. For example, to allocate bit 21, you set `sigMask` to 0x00200000. If you don't care which signal bit you get, you can use `AllocSignal(0)`, in which case the kernel chooses a free bit and allocates it for you. It is most common to simply use `AllocSignal(0)` instead of requesting specific signal bits.

`AllocSignal()` returns a signal mask with bits set to 1 where signals were successfully allocated. If no bits were available for allocation, the call returns 0. If there was an error in allocation, the call returns an error code (a negative number).

The code fragment in example 8-1 demonstrates using `AllocSignal()` to allocate a signal bit. In this program, a parent task creates two threads, and it uses the signal mechanisms for the threads to communicate with the parent task. Here, the parent task allocates two of its signal bits, one for each thread. First, it declares two global variables, `threadSig1` and `threadSig2`, to use as signal mask variables for each of the threads:

Example 1: *Allocating a signal bit.*

```
/* Global variables shared by all threads. */
```

```
static int32  thread1Sig;
static int32  thread2Sig;
...
...

/* allocate one signal bit for each thread */
thread1Sig = AllocSignal(0);
thread2Sig = AllocSignal(0);

if ((thread1Sig == 0) || (thread2Sig == 0))
{
    /* could not allocate the needed signal bits */
}
```

The two calls to `AllocSignal()` allocate 2 available signal bits. If successful, the variables `threadSig1` and `threadSig2` each contain a signal mask with a specially allocated bit for each thread to use.

Receiving a Signal

To receive a signal, a task enters wait state using this call:

```
int32 WaitSignal( uint32 sigMask )
```

`WaitSignal()` accepts a signal mask that specifies which of the task's allocated signal bits to wait for. It can specify more than one bit to wait for, but it can't specify any bits that haven't already been allocated. If unallocated bits are specified, the call returns a negative value (an error code) when it executes. Before returning, `WaitSignal()` clears the signal bits.

When the allocated bits are specified and the call executes, the task enters the wait state until a signal is sent to one of the signal bits it specified in `WaitSignal()`. The task then exits the wait state and returns to the ready queue. The call returns a signal mask with 1 set in each bit where a signal was received.

A task can receive signals before it enters a wait state, in which case they're kept as pending signals in a pending signals mask maintained in the task's control block (TCB). There is no signal queue-that is, no more than one signal can be kept pending for each allocated signal bit. When a task calls `WaitSignal()`, any pending signals that match the signal mask given to `WaitSignal()` cause the call to return immediately, and the task never enters the wait state. The signal bits that the task was waiting for are cleared before returning.

Sampling and Changing the Current Signal Bits

It is sometimes necessary for a task to look at the current state of the signal bits to determine whether a given signal was received. To do this, use `GetCurrentSignals`():

```
int32 GetCurrentSignals(void);
```

The macro returns a signal mask with bits on for every signal bit that has been received. This provides a sample or the

current signal bits without entering the wait state.

You can forcibly set the signal bits of your task by using `SendSignal()` with 0 for the task item. This is sometimes useful to set the initial state of the task's signals.

You can also forcibly clear signal bits of a task by using the `ClearCurrentSignals()` macro.

```
Err ClearCurrentSignals(int32 signalMask);
```

This macro takes a signal mask that describes the signal bits to clear. The macro returns a negative value if reserved signal bits (bits 0-7) are specified.

Sending a Signal

A task sends one or more signals to another task with this call:

```
Err SendSignal( Item task, uint32 sigMask )
```

The call accepts the item number of the task to which the signal or signals should be sent, and a signal mask with a 1 in each bit to which a signal is sent. When it executes, the kernel checks that each specified signal bit in `SendSignal()` is allocated by the receiving task. If it's not, then the call returns a negative number (an error code).

If all specified bits are allocated by the receiving task, then the kernel performs a logical OR between the `SendSignal()` signal mask and the pending signals mask maintained in the receiving task's TCB. This is how the pending signals mask maintains a record of pending signals. When the receiving task uses `WaitSignal(),` the kernel checks the pending signals mask, and if any bits are set to 1, the task immediately returns to execution.

Freeing Signal Bits

When a task no longer needs an allocated signal bit, it uses this call:

```
Err FreeSignal( uint32 sigMask )
```

The call accepts a signal mask with a 1 set in each bit that is to be freed, and a 0 set in each bit where allocation status should be unchanged. For example, if bits 8 and 9 are currently allocated and you wish to free only bit 8, then you need to create a signal mask with the binary value 1<<8 (which equals a 256 decimal value).

`FreeSignal()`, frees the specified bits when executed. It returns 0 if it was successful, and a negative number (an error code) if unsuccessful.

Sample Code for Signals

Example 2 is a full code sample on which the previous code sample is based. The variables declared at the beginning of the example are global to the main routine (the parent task) and the two threads. These global variables pass signal

masks between the threads and the task. The main routine, which appears at the end of the example, allocates signals for the two threads it will create.

The `main()` routine uses `WaitSignal()` to wait for signals from the two threads. The threads use `SendSignal()` to send their signals to the parent task.

Example 2: *Complete code sample for signals.*

```
#include "types.h"
#include "task.h"
#include "kernel.h"
#include "stdio.h"
#include "operror.h"


/*****************************************************************************/


/* Global variables shared by all threads. */
static int32  thread1Sig;
static int32  thread2Sig;
static Item   parentItem;
static uint32 thread1Cnt;
static uint32 thread2Cnt;


/*****************************************************************************/


/* This routine shared by both threads */
static void DoThread(int32 signal, uint32 amount, uint32 *counter)
{
uint32 i;

    while (TRUE)
    {
        for (i = 0; i < amount; i++)
        {
            (*counter)++;
            SendSignal(parentItem,signal);
        }
    }
}


/*****************************************************************************/


static void Thread1Func(void)
{
```

```
    DoThread(thread1Sig, 100000, &thread1Cnt);
}


/*****************************************************************************/


static void Thread2Func(void)
{
    DoThread(thread2Sig, 200000,&thread2Cnt);
}


/*****************************************************************************/


int main(int32 argc, char **argv)
{
uint8  parentPri;
Item   thread1Item;
Item   thread2Item;
uint32 count;
int32  sigs;

    /* get the priority of the parent task */
    parentPri  = CURRENTTASK->t.n_Priority;

    /* get the item number of the parent task */
    parentItem = CURRENTTASK->t.n_Item;

    /* allocate one signal bits for each thread */
    thread1Sig = AllocSignal(0);
    thread2Sig = AllocSignal(0);

    /* spawn two threads that will run in parallel */
    thread1Item = CreateThread("Thread1", parentPri, Thread1Func, 2048);
    thread2Item = CreateThread("Thread2", parentPri, Thread2Func, 2048);

    /* enter a loop until we receive 10 signals */
    count = 0;
    while (count < 10)
    {
        sigs = WaitSignal(thread1Sig | thread2Sig);

        printf("Thread 1 at %d, thread 2 at %d\n",thread1Cnt,thread2Cnt);

        if (sigs & thread1Sig)
            printf("Signal from thread 1\n");

        if (sigs & thread2Sig)
            printf("Signal from thread 2\n");
```

```
        count++;
    }

    /* nuke both threads */
    DeleteThread(thread1Item);
    DeleteThread(thread2Item);
}
```

Passing Messages

The message system allows tasks to send data to one another, providing information flow from task to task. Tasks must create the following elements for the message system to work:

- The sending task must create a message port.
- The receiving task must create a message port.
- The sending task must create a message.

Once the elements are in place, the sending task follows these procedures:

- It specifies a message to send and a destination message port to which to send the message.
- It specifies the data to go with the message.

The receiving task does one of the following procedures:

- It enters the wait state to wait for a message on a specified message port.
- It checks one of its message ports to see if a message has arrived there.

Once a message arrives at a task's message port, the task follows these procedures:

- It removes the message from its message port.
- It gets a pointer to the message, and reads data directly from the message, or finds the data block pointer and size and reads from the data block.
- If the message needs a reply, the task writes new data directly into the message or sets a pointer and size in the message for a new data block.
- It sends the reply back to the message.

The message system is very flexible, and allows a single task to create multiple message ports and multiple messages. A single message port can receive many messages; they're queued up and retrieved one at a time by the receiving task. A message port allows a single task to receive, in serial order, messages from many other tasks.

Messages, like all other items, are assigned priority numbers. These priority numbers determine the order of messages within a message queue: higher- priority messages go to the front of the queue. Messages with the same priority are arranged in arrival order: earlier messages come first.

Creating a Message Port

Before a task can send or receive a message, it must have at least created one message port. To create a message port, use this call:

```
Item CreateMsgPort( const char *name, uint8 pri, uint32 signal )
```

The call accepts three arguments: a name string, `name`, which it uses to name the message port; a 1-byte priority value, `pri`, which assigns a priority to the message port; and a 32-bit signal mask, `signal`, which assigns an allocated signal bit to the message port.

The message port's name and priority don't change the way the message port operates, but are useful when another task tries to find the message port.

The signal mask specifies a signal to associate with the message port. The signal mask must contain only allocated signal bits. If you specify a signal mask of 0, then `CreateMsgPort()` allocates a signal bit automatically. When the port is later deleted using `DeleteMsgPort()`, the signal bit is automatically freed.

`CreateMsgPort()` returns the item number of the newly created message port if successful, or returns a negative number (an error code) if unsuccessful. The message port now exists as an item in system memory, owned by the creating task.

The message port uses its assigned signal bit to signal its owner task whenever a message arrives at the port. The signal bit should *not* be freed until the message port is freed.

The item number returned by `CreateMsgPort()` is a handle to the new port. The owner task should give it out to any task that may want to send a message to it. Sending a message requires specifying the item number of the message port to which the message is sent.

Creating a Message

To create a message to send to a specific message port, a task must first create its own message port to use as a reply port. Then, it must specify which of three message types to create:

- A **standard message**, which has a pointer to a block of data and the size of that data.
- A **small message**, which has two 32-bit words of data.
- A **buffered message**, which has a built-in data block.

If the task sends 8 or fewer bytes of data within the message itself, it should create a small message.

If the task sends more than 8 bytes of data and only points to the data in a data block that isn't carried with the message, it should create a standard message.

And if the task sends more than 8 bytes of data within the message, it should create a buffered message.

Standard Messages

To create a standard message, use this call:

```
Item CreateMsg( const char *name, uint8 pri, Item mp )
```

The first argument, `name`, is the name for the message, which is useful for finding the message later. The second argument, `pri`, sets the message's priority, which determines how it is sorted in a receiving task's message queue. It can be a value from 0 to 255. The third argument, `mp`, is the item number of a message port that belongs to the creating task.

This is the message's reply port, where it returns if the receiving task sends it back with a reply call.

A standard message carries with it a pointer to a data block, and the size of the data block. The task that receives the message can obtain the pointer to the data block and access the data. If the receiving task has write permission to the pages of memory where the data block is located, it can even write to the data block directly. Because of the fact you pass a pointer to a data block, standard messages are often referred to as "pass-by-reference" messages.

The reply to a standard message can contain 4 bytes of data.

When executed, `CreateMsg()` creates a standard message and returns the message's item number. If unsuccessful, it returns a negative number (an error code). Example 1 uses `CreateMsg()`:

Example 1: *Sample using CreateMsg( ).*

```
static Item myMsg;
static Item mpItem;

/* Create a message */
    myMsg = CreateMsg("Message1", 0, mpItem);
     if (myMsg < 0)
    {
        printf("error creating Message\n");
        return (-2);
    }
```

Small Messages

Both standard and buffered messages have message structure fields with pointers to a data block and the size of the data block. A small message uses those same fields to store up to 8 bytes of data instead of storing a pointer and a block size value.

To create a small message, use `CreateSmallMsg()`:

```
Item CreateSmallMsg( const char *name, uint8 pri, Item mp )
```

The arguments are identical to those for `CreateMsg()`. You'll see the difference when you call `SendSmallMsg()` (described later).

Buffered Messages

Some applications require a message that can carry data within the message itself. This lets the sender copy data into the message, send the message, and not have to keep around its original data buffer. The buffer is no longer needed because the data is now in the message itself. In addition, when the destination task returns a buffered message, it can also write data to the message's buffer, allowing a great deal of information to be returned.

Because buffered messages require data to be copied into them before being sent, they are often referred to as "pass-by-value" messages.

With buffered messages, a sending task can send large amounts of read/write data to another task. The receiving task can modify this data and return it to the caller. This avoids granting the receiving task write permission to memory buffers in the sending task's address space.

To create a buffered message, use this call:

```
Item CreateBufferedMsg( const char *name, uint8 pri, Item mp, uint32 datasize
)
```

Like `CreateMsg()`, this call accepts arguments that point to a name, supply a priority, and give the item number of a reply port for the message. The additional argument, `datasize`, specifies the size, in bytes, of the data buffer to create.

When the call executes, it creates a buffered message with its accompanying buffer. If successful, the call returns the item number of the message. If unsuccessful, it returns a negative number (an error code).

Sending a Message

A task can send all three types of messages, but each requires different handling.

Small Messages

To send a small message, use this call:

```
Err SendSmallMsg( Item mp, Item msg, uint32 val1, uint32 val2 )
```

The first argument, `mp`, is the item number of the message port to which the message is sent. The second argument, `msg`, is the item number of the small message to send. The third argument, `val1`, is the first 4 bytes of data to send in the message; the fourth argument, `val2`, is the second 4 bytes of data to send in the message. (If you don't know the item number of the message port, but you know its name, you can use `FindMsgPort()` to get the item number.)

When `SendSmallMsg()` executes, it writes the first value into the `msg_DataPtr` field of the message's data structure and the second value into the `msg_DataSize` field of the structure. It then sends the message to the specified message port. It returns a 0 if the message is successfully sent, or a negative number (an error code) if unsuccessful.

You should *not* send a standard or buffered message with this call, because the values you supply can be read as an erroneous pointer and data size.

Standard and Buffered Messages

To send a standard or buffered message, use this call:

```
Err SendMsg( Item mp, Item msg, const void *dataptr, int32 datasize )
```

Like `SendSmallMsg()`, this call accepts the item number of a message port to which to send the message, `mp`, and the item number of the message to send, `msg`. Instead of accepting values to store with the message, `SendMsg()` accepts a pointer to a data block in the `dataptr` argument and the size, in bytes, of the data block in the `datasize` argument.

When `SendMsg()` executes, it sends the message, returning a 0 if successful, or a negative number (an error code) if unsuccessful. Its effect on the message depends on whether the message is standard or buffered.

A standard message stores the data block pointer and the data size value in the message. The data block remains where it is, and the receiving task reads it there, using the pointer and size value to find it and know how far it extends.

With a buffered message, `SendMsg()` checks the size of the data block to see if it will fit in the message's buffer. If it won't, the call returns an error. If it does fit, `SendMsg()` copies the contents of the data block into the message's buffer. The receiving task can then read the data directly out of the message's buffer.

The advantage of a buffered message is that it carries the full data block within its buffer, so the sending task can immediately use the original data block's memory for something else. The task doesn't need to maintain the data block until it's sure that the receiving task has read the block. The data block is handled by the system, and is carried within the message in protected system memory.

Receiving a Message

Once a message is sent to a message port, it is held in the message port's message queue until the receiving task retrieves the message, or the sending task decides to pull it back. The messages within the queue are put in order first by priority, and then by order received.

To receive a message, a task can wait until it receives notification of a message at a message port, or it can check a message port directly to see if a message is there.

Waiting for a Message

To enter the wait state until a message arrives on a specific message port, a task uses this call:

```
Item WaitPort( Item mp, Item msg )
```

`WaitPort()` accepts the required argument, `mp`, which contains the item number of the message port where the task receives messages. The call also accepts an optional argument, `msg`, which contains the item number of a specific message the task expects to receive at the message port. (To wait for any incoming message, use 0 as the value of `msg`.)

Note: If the message arrived before `WaitPort()` is called, the task never enters the wait state.

When `WaitPort()` executes, the task enters the wait state until it detects a message on the message port's message queue. At that point, if the call doesn't specify a specific message, the task exits the wait state, and the call returns the item number of the first message in the port's queue. The message is removed from the message queue, and the task can use its item number to find and read the message.

If a specific message was given to `WaitPort()` in addition to a message port, the task waits until that message arrives at the message port. When it arrives, the task exits the wait state and the message is removed from the message queue.

Checking for a Message

To check for a message at a message port without entering the wait state, a task uses this call:

```
Item GetMsg( Item mp )
```

`GetMsg()` accepts a single argument: the item number of the message port where it wants to check for messages. When it executes, it checks the message port to see if there are any messages in the port's queue. If there are, it returns the item number of the first message in the queue and removes it from the queue. If there is no message at the message port, the call returns 0. If there is an error, the call returns an error code (a negative number).

Working With a Message

Once a task receives a message, it interprets the contents of the message to determine what the message means. To do this, the task must obtain a pointer to the Message structure associated with the message's item number:

```
Message *msg;
Item     msgItem;

   msg = (Message *)LookupItem(msgItem);
```

Once the task has the pointer, it can determine the type of message it has received by inspecting the contents of the `msg.n_Flags` field of the message structure. If the `MESSAGE_SMALL` bit is set in this field, it means it is a small message. If the `MESSAGE_PASS_BY_VALUE` bit is set, it means it is a buffered message. If neither of these bits is set, it means that it is a standard message.

When you set up communication channels among your tasks or threads using message ports, you should know what type of messages will be sent around to these ports, so that you don't always need to check the types.

Small Messages

When you receive a small message, the only information you can get from the message is contained in the `msg_DataPtr` and `msg_DataSize` fields. The information corresponds directly to the `val1` and `val2` arguments the sending task specified when it called `SendSmallMsg()`. When you return such a message, you supply a 4-byte result code, and two new 32-bit values to put into `msg_DataPtr` and `msg_DataSize`.

Standard Messages

When you receive a standard message, the `msg_DataPtr` field of the message contains the address of a data area which you can access. The size of the data area is contained in `msg_DataSize`. Typically, the data area points to a data structure belonging to the task that sent the message.

If the standard message comes from another task, as opposed to a sibling thread, the data area likely will only be readable by the receiving task, unless the sending task takes the extra steps to grant write permission to that data area. To determine whether your task can write to a data area pointed to in a standard message, use the `IsMemWritable()` function.

If you send messages between threads and tasks of the same application, it is often not necessary to go through the extra trouble of checking whether the data area is writable before writing to it, since this can be part of the protocol setup within your application.

When a task replies to a standard message, it provides a 4-byte result code, and new values for `msg_DataPtr` and

msg_DataSize. When a task calls ReplyMsg(), it can leave the values to what they were by simply using msg_DataPtr and msg_DataSize as parameters to ReplyMsg().

Buffered Messages

When a task receives a buffered message, the msg_DataPtr field of the message contains the address of a read-only data area, and msg_DataSize specifies the number of useful bytes within this buffer. The task can read this data at will.

When a task replies a buffered message, it supplies a 4-byte result code, and can optionally supply new data to be copied into the message's buffer. The task can determine the size available in the message's buffer by looking at the msg_DataPtrSize field of the message structure.

Pulling Back a Message

When a sending task sends a message, the task can pull the message out of a message queue before the message has been received. To do so, it uses the GetThisMsg() call:

```
Item GetThisMsg( Item msg )
```

The single argument, msg, specifies the message to rescind. If the message is still on the message port it was sent to, it is removed and its item number returned. If the message was already removed from the port by the receiving task, then the function returns 0.

Replying to a Message

Once a message is received, the receiving task can reply, which returns a message to its reply port. The message contains 4 bytes that are set aside for a reply code; the replying task assigns an arbitrary value for storage there, often a value that reports the success or failure of message handling.

Once again, the three different message types require different handling when replying to a message.

Small Messages

To send back a small message in reply to the sending task, use this call:

```
Err ReplySmallMsg( Item msg, int32 result, uint32 val1, uint32 val2 )
```

ReplySmallMsg() accepts four arguments. The first argument, msg, is the item number of the message being returned in reply. The second argument, result, is a 32-bit value written into the reply field of the message data structure. The third and fourth arguments, val1 and val2, are data values written into the pointer and size fields of the message data structure (just as they are in SendSmallMsg()). Note that no destination message port is specified because the message returns to its reply port, which is specified within the message data structure.

When ReplySmallMsg() executes, it sends a message back in reply and returns a 0 if successful, or an error code (a negative number) if unsuccessful.

Standard and Buffered Messages

To send back a standard or buffered message in reply, use this call:

```
Err ReplyMsg( Item msg, int32 result, const void *dataptr, int32 datasize)
```

`ReplyMsg()` accepts four arguments. The first two, `msg` and `result`, are the same as those accepted by `ReplySmallMsg()`. The third, `dataptr`, is a pointer to a data block in memory. The fourth, `datasize`, is the size in bytes of that data block.

When `ReplyMsg()` executes, it sends the message in reply and returns a 0 if successful, or an error code (a negative number) if unsuccessful. The effect the reply has on the message depends on whether the message is standard or buffered, just as it does in `SendMsg()`.

If the message is standard, the data block pointer and the data size value are stored in the message, and are read as such by the task getting the reply. If the message is buffered, `ReplyMsg()` checks the size of the data block to see if it will fit in the message's buffer. If it doesn't fit, the call returns an error. If it does fit, `ReplyMsg()` copies the contents of the data block into the message's buffer and sends the message.

Finding a Message Port

When a task sends a message to message a port, it needs to know the item number of the port. To get the item number of the port, it is often necessary to use the `FindMsgPort()` call:

```
Item FindMsgPort( const char *name )
```

When you provide `FindMsgPort()` the name of the message port to find, it returns the item number of that port. If the port doesn't exist, it returns a negative error code. If multiple ports of the same name exist, it returns the item number of the port with the highest priority.

Example Code for Messages

The code in Example 2 demonstrates how to send messages among threads or tasks.

The `main()` routine of the program creates a message port where it can receive messages. It then spawns a thread. This thread creates its own message port and message. The thread then sends the message to the parent's message port. Once the parent receives the message, it sends it back to the thread.

Example 2: *Samples using the message passing routines (msgpassing.c).*

```
#include "types.h"
#include "item.h"
#include "kernel.h"
#include "task.h"
#include "msgport.h"
#include "operror.h"
#include "stdio.h"
```

```
/****************************************************************************/

/* A signal mask used to sync the thread with the parent */
int32 parentSig;

/****************************************************************************/


static void ThreadFunction(void)
{
Item  childPortItem;
Item  childMsgItem;
Item  parentPortItem;
Err   err;
Msg  *msg;

    printf("Child thread is running\n");

    childPortItem = CreateMsgPort("ChildPort",0,0);
    if (childPortItem >= 0)
    {
        childMsgItem = CreateSmallMsg("ChildMsg",0,childPortItem);
        if (childMsgItem >= 0)
        {
            parentPortItem = FindMsgPort("ParentPort");
            if (parentPortItem >= 0)
            {
                /* tell the paren't we're done initializing */
                SendSignal(CURRENTTASK->t_ThreadTask->t.n_Item,parentSig);

                err = SendSmallMsg(parentPortItem,childMsgItem,12,34);
                if (err >= 0)
                {
                    err = WaitPort(childPortItem,childMsgItem);
                    if (err >= 0)
                    {
                        msg = (Msg *)LookupItem(childMsgItem);
                        printf("Child received reply from parent: ");
                        printf("msg_Result %d, msg_DataPtr %d, msg_DataSize %d\n",
                               msg->msg_Result, msg->msg_DataPtr, msg-
                                  >msg_DataSize);
                    }
                    else
                    {
                        printf("WaitPort() failed: ");
                        PrintfSysErr(err);
                    }
                }
                else
                {
```

```
                printf("SendSmallMsg() failed: ");
                PrintfSysErr(err);
            }

            SendSignal(CURRENTTASK->t_ThreadTask->t.n_Item,parentSig);
        }
        else
        {
            printf("Could not find parent message port: ");
            PrintfSysErr(parentPortItem);
        }
        DeleteMsg(childMsgItem);
    }
    else
    {
        printf("CreateSmallMsg() failed: ");
        PrintfSysErr(childMsgItem);
    }
    DeleteMsgPort(childPortItem);
}
else
{
    printf("CreateMsgPort() failed: ");
    PrintfSysErr(childPortItem);
}
}

/*****************************************************************************/


int main(int32 argc, char **argv)
{
Item  portItem;
Item  threadItem;
Item  msgItem;
Msg  *msg;

    parentSig = AllocSignal(0);
    if (parentSig > 0)
    {
        portItem = CreateMsgPort("ParentPort",0,0);
        if (portItem >= 0)
        {
            threadItem = CreateThread("Child",10,ThreadFunction,2048);
            if (threadItem >= 0)
            {
                /* wait for the child to be ready */
                WaitSignal(parentSig);

                /* confirm that the child initialized correctly */
```

```
                if (FindMsgPort("ChildPort") >= 0)
                {
                    printf("Parent waiting for message from child\n");

                    msgItem = WaitPort(portItem,0);
                    if (msgItem >= 0)
                    {
                        msg = (Msg *)LookupItem(msgItem);
                        printf("Parent got child's message: ");
                        printf("msg_DataPtr %d, msg_DataSize %d\n",

                          msg->msg_DataPtr, msg->msg_DataSize);
                        ReplySmallMsg(msgItem,56,78,90);
                    }
                    else
                    {
                        printf("WaitPort() failed: ");
                        PrintfSysErr(msgItem);
                    }
                }

                /* wait for the thread to tell us it's done before we zap it */
                WaitSignal(parentSig);

                DeleteThread(threadItem);
            }
            else
            {
                printf("CreateThread() failed: ");
                PrintfSysErr(threadItem);
            }
            DeleteMsgPort(portItem);
        }
        else
        {
            printf("CreateMsgPort() failed: ");
            PrintfSysErr(portItem);
        }
        FreeSignal(parentSig);
    }
    else
    {
        printf("AllocSignal() failed");
    }

    return 0;
}
```

Function Calls

The following are the calls related to signals and messages.

Allocating Signals

The following calls work with signal bits.

- `AllocSignal()` Allocates one or more signals for intertask communication.
- `ClearCurrentSignals()` Clears specified signals bits of the current task.
- `FreeSignal()` Frees signal bits.
- `GetCurrentSignals()` Returns the state of the current task's signal bits.
- `SendSignal()` Sends one or more signals to another task or thread.
- `WaitSignal()` Waits for one or more signals to arrive.

Creating Messages and Message Ports

The following calls create messages and ports.

- `CreateBufferedMsg()` Creates a buffered message.
- `CreateMsg()` Creates a standard message.
- `CreateMsgPort()`Creates a message port.
- `CreateSmallMsg()`Creates a small message.
- `CreateUniqueMsgPort()`Creates a message port with a guaranteed unique name.

Deleting Messages and Message Ports

The following calls delete messages and ports.

- `DeleteMsg()` Deletes a message of any type.
- `DeleteMsgPort()`Deletes a message port.

Finding a Message Port

The following call finds a message port by name.

- `FindMsgPort()` Returns the item number of the message port having the specified name.

Sending Messages

The following calls send different types of messages:

- `SendMsg()` Sends a message.
- `SendSmallMsg()`Sends a small message.

Receiving Messages

The following calls retrieve messages:

- `GetMsg()` Gets a message from a message port.
- `GetThisMsg()` Gets a specific message.
- `WaitPort()`Waits for a message to arrive.

Replying to Messages

The following calls reply to messages:

- `ReplyMsg()` Sends a reply to a message.
- `ReplySmallMsg()` Sends a reply to a small message.

Example

Example 1 uses Access to display eight different transaction boxes, one for each type of transaction. Tag argument's supply explanatory text and two labeled choice buttons for transaction boxes that need them. The task enters the wait state after it requests each transaction and awaits the user choice in a reply message from Access. When the task receives the reply, it leaves the wait state.

Example 1: *Using Access to put up transaction boxes.*

```
#include "types.h"
#include "mem.h"
#include "msgport.h"
#include "graphics.h"
#include "access.h"
#include "stdio.h"
#include "string.h"



/****************************************************************************/

#define ERROR(x,y)  {printf(x); PrintfSysErr(y); result = y;}
#define NUM_SCREENS 1


/****************************************************************************/


static Item  accessPortItem;
static Item  accessItem;
static Item  replyPortItem;
static Item  msgItem;
static Msg  *msg;
static Item  screenGroupItem;
static Item  screenItems[NUM_SCREENS];


/****************************************************************************/


static void ClearScreenPages(Item *screenItems)
{
uint32  i;
Screen *screen;
Bitmap *bitmap;
uint32  numBytes;
Item    sport;
uint32  vramPageSize;

    /* If this call fails, SetVRAMPages() will also fail, and nothing bad
     * will happen. So we don't bother to check the error return.
```

```
     */
    sport = GetVRAMIOReq();

    for (i = 0; i < NUM_SCREENS; i++)
    {
        screen   = (Screen *)LookupItem(screenItems[i]);
        bitmap   = screen->scr_TempBitmap;
        numBytes = bitmap->bm_Width * bitmap->bm_Height * 2;

        vramPageSize = GetPageSize(MEMTYPE_VRAM);

        SetVRAMPages(sport,bitmap->bm_Buffer, 0,(numBytes + vramPageSize - 1) /

            vramPageSize,-1);
    }

    DeleteItem(sport);
}


/****************************************************************************/

static void TransferScreenPages(Item *screenItems, bool toAccess)
{
uint32  i;
Screen *screen;
Bitmap *bitmap;
uint32  numBytes;

    for (i = 0; i < NUM_SCREENS; i++)
    {
        screen   = (Screen *)LookupItem(screenItems[i]);
        bitmap   = screen->scr_TempBitmap;
    numBytes = bitmap->bm_Width * bitmap->bm_Height * 2;

        if (toAccess)
            ControlMem((void *)bitmap->bm_Buffer,numBytes,MEMC_OKWRITE,accessItem);
        else
            ControlMem((void *)bitmap->bm_Buffer,numBytes,MEMC_NOWRITE,accessItem);
    }
}

/****************************************************************************/

static void SendAccessMsg(uint32 callNumber, char *buffer,
                          uint32 tags, ...)
{
Err err;

    err = SendMsg(accessPortItem, msgItem, &tags, 0);

    if (err >= 0)
    {
```

```
        WaitPort(replyPortItem,0);
        printf("Call #%ld returned %ld, buffer = '%s'\n",callNumber,msg-
>msg_Result,buffer);
    }
    else
    {
        printf("SendAccessMsg() #%ld failed with %ld\n",callNumber,err);
    }

    ClearScreenPages(screenItems);
}


/*****************************************************************************/


static void CallThemAll(void)
{
char buf[256];

    buf[0] = 0;

    SendAccessMsg(1, buf, ACCTAG_REQUEST,          ACCREQ_ONEBUTTON,
                          ACCTAG_SCREEN,         screenItems[0],
                          ACCTAG_TITLE,          "Hi ho",
                          ACCTAG_TEXT,           "Isn't this a great example?",
                          ACCTAG_BUTTON_ONE,     "I agree",
                          ACCTAG_FG_PEN,         0x7000,
                          TAG_END);

    SendAccessMsg(2, buf, ACCTAG_REQUEST,          ACCREQ_TWOBUTTON,
                          ACCTAG_SCREEN,         screenItems[0],
                          ACCTAG_TITLE,          "Space Zappers",
                          ACCTAG_TEXT,           "Wanna play more Space Zappers?",
                          ACCTAG_BUTTON_ONE,     "Yep",
                          ACCTAG_BUTTON_TWO,     "Nope",
                          ACCTAG_FG_PEN,         0x0700,
                          TAG_END);

    ControlMem(buf,sizeof(buf),MEMC_OKWRITE,accessItem);

    strcpy(buf,"Gee");
    SendAccessMsg(3, buf, ACCTAG_REQUEST,          ACCREQ_LOAD,
                          ACCTAG_SCREEN,         screenItems[0],
                          ACCTAG_TITLE,          "Space Zappers",
                          ACCTAG_STRINGBUF,      buf,
                          ACCTAG_STRINGBUF_SIZE, sizeof(buf),
                          TAG_END);

    strcpy(buf,"Wow");
    SendAccessMsg(4, buf, ACCTAG_REQUEST,          ACCREQ_SAVE,
                          ACCTAG_SCREEN,         screenItems[0],
                          ACCTAG_TITLE,          "Space Zappers",
```

```
                        ACCTAG_STRINGBUF,      buf,
                        ACCTAG_STRINGBUF_SIZE, sizeof(buf),
                        TAG_END);

    strcpy(buf,"Zawee");
    SendAccessMsg(5, buf, ACCTAG_REQUEST,           ACCREQ_DELETE,
                        ACCTAG_SCREEN,         screenItems[0],
                        ACCTAG_TITLE,          "Space Zappers",
                        ACCTAG_STRINGBUF,      buf,
                        ACCTAG_STRINGBUF_SIZE, sizeof(buf),
                        TAG_END);

    ControlMem(buf,sizeof(buf),MEMC_NOWRITE,accessItem);
}


/*****************************************************************************/

int main(uint32 argc, char *argv[])
{
MsgPort *accessPort;
Err      result;
Err      err;

    err = OpenGraphicsFolio();
    if (err >= 0)
    {
        screenGroupItem = CreateScreenGroupVA(screenItems,
                    CSG_TAG_SCREENCOUNT, (void *)NUM_SCREENS,
                                          CSG_TAG_DONE);
        if (screenGroupItem >= 0)
        {
            ClearScreenPages(screenItems);
            AddScreenGroup(screenGroupItem, NULL);

            err = DisplayScreen(screenItems[0], 0);
            if (err >= 0)
            {
                accessPortItem = FindMsgPort("access");
                if (accessPortItem >= 0)
                {
                    accessPort = (MsgPort *)LookupItem(accessPortItem);
                    if (accessPort)
                    {
                        accessItem = accessPort->mp.n_Owner;

                        replyPortItem = CreateMsgPort("accessexample",0,0);
                        if (replyPortItem >= 0)
                        {
                            msgItem = CreateMsg("accessmsg",0,replyPortItem);
                            if (msgItem >= 0)
                            {
```

```
                            msg = (Msg *)LookupItem(msgItem);
                            if (msg)
                            {
                                TransferScreenPages(screenItems, TRUE);

                                CallThemAll();
                                result = 0;

                                TransferScreenPages(screenItems, FALSE);
                            }
                            else
                            {
                                ERROR("Could not lookup message\n",0);
                            }
                            DeleteItem(msgItem);
                        }
                        else
                        {
                            ERROR("Could not create message\n",msgItem);
                        }
                        DeleteMsgPort(replyPortItem);
                    }
                    else
                    {
                        ERROR("Could not create reply message

                          port\n",replyPortItem);
                    }
                }
                else
                {
                    ERROR("Could not lookup the Access message port\n",-1);
                }
            }
            else
            {
                ERROR("Could not find the Access message

                  port\n",accessPortItem);
            }
            RemoveScreenGroup(screenGroupItem);
        }
        else
        {
            ERROR("Could not display screen group\n",err);
        }
        DeleteScreenGroup(screenGroupItem);
    }
    else
    {
        ERROR("Couldn't create screen group\n",screenGroupItem);
    }
```

```
    CloseGraphicsFolio();
    }
    else
    {
        ERROR("Could not open the graphics folio\n",err);
    }

    return ((int)result);
}
```

Access Message Tag Arguments

These tags define an Access transaction:

- ACCTAG_REQUEST requests a type of transaction. It takes as an argument one of the constants listed below:
    - ACCREQ_LOAD
    - ACCREQ_SAVE
    - ACCREQ_DELETE
    - ACCREQ_OK
    - ACCREQ_OKCANCEL
    - ACCREQ_ONEBUTTON
    - ACCREQ_TWOBUTTON

- ACCTAG_SCREEN specifies a screen on which to display the transaction box. It takes the item number of an existing screen as an argument.

- ACCTAG_BUFFER is reserved for future use.

- ACCTAG_BUFFERSIZE is reserved for future use.

- ACCTAG_TITLE takes as its argument a pointer to a NULL-terminated text string to be used as the transaction box for the title.

- ACCTAG_TEXT takes as its argument a pointer to a NULL-terminated text string to be used as the explanatory text for the transaction box.

- ACCTAG_BUTTON_ONE takes as its argument a pointer to a NULL- terminated text string to be used as the text for the left button in the transaction box.

- ACCTAG_BUTTON_TWO takes as its argument a pointer to a NULL- terminated text string to be used as the text for the right button in the transaction box.

- ACCTAG_SAVE_BACK specifies that pixels beneath the transaction box be saved and restored when the transaction box is closed. It takes zero as its argument. If this tag argument isn't used, background pixels aren't saved and restored.

- ACCTAG_STRINGBUF points to a string buffer that receives user input text from Access. It takes

a pointer to a memory location as its argument.

- ACCTAG_STRINGBUF_SIZE gives the size of a string buffer that receives user input text from Access. It takes the size of the buffers in byte as its arguments.

- ACCTAG_FG_PEN provides the color used for text in the transaction box. It takes a pen color value as its argument.

- ACCTAG_BG_PEN provides the color used for the transaction box background. It takes a pen color value as its argument.

- ACCTAG_HILITE_PEN provides the color used for the background of a highlighted button in the transaction box. It takes a pen color value as its argument.

- ACCTAG_SHADOW_PEN provides the color used for the button outline in the transaction box. It takes a pen color value as its argument.

Examples

This section provides code listings of programs that use the File folio and the file system.

Using NVRAM

Example 1 is a program listing that demonstrates how to save data to a file in NVRAM or to any other writable device.

The `SaveDataFile()` function takes a name, a data pointer, and a size indicator, and creates a file with that name and size, containing the supplied data.

Writing data to a file requires a bit of finesse. Data can only be written in blocks, not bytes. So if the data being written out is not a multiple of the target device's block size, the last chunk of data must be copied to a buffer which is the size of a block, and the block can then be written out.

Example 1: *Saving data to a file in NVRAM (nvram.c).*

```
#include "types.h"
#include "filesystem.h"
#include "filefunctions.h"
#include "io.h"
#include "string.h"
#include "mem.h"
#include "stdio.h"
#include "operror.h"


/*****************************************************************************/


Err SaveDataFile(const char *name, void *data, uint32 dataSize)
{
Item       fileItem;
Item       ioReqItem;
IOInfo     ioInfo;
void      *temp;
Err        result;
uint32     numBlocks;
uint32     blockSize;
uint32     roundedSize;
FileStatus status;

    /* get rid of the file if it was already there */
    DeleteFile((char *)name);

    /* create the file again... */
    result = CreateFile((char *)name);
```

```
if (result >= 0)
{
    /* open the file for access */
    fileItem = OpenDiskFile((char *)name);
    if (fileItem >= 0)
    {
        /* create an IOReq to communicate with the file */
        ioReqItem = CreateIOReq(NULL, 0, fileItem, 0);
        if (ioReqItem >= 0)
        {
            /* get the block size of the file */
            memset(&ioInfo, 0, sizeof(IOInfo));
            ioInfo.ioi_Command         = CMD_STATUS;
            ioInfo.ioi_Recv.iob_Buffer = &status;
            ioInfo.ioi_Recv.iob_Len    = sizeof(FileStatus);
            result = DoIO(ioReqItem, &ioInfo);
            if (result >= 0)
            {
                blockSize = status.fs.ds_DeviceBlockSize;
                numBlocks = (dataSize + blockSize - 1) / blockSize;

                /* allocate the blocks we need for this file */
                ioInfo.ioi_Command         = FILECMD_ALLOCBLOCKS;
                ioInfo.ioi_Recv.iob_Buffer = NULL;
                ioInfo.ioi_Recv.iob_Len    = 0;
                ioInfo.ioi_Offset          = numBlocks;
                result = DoIO(ioReqItem, &ioInfo);
                if (result >= 0)
                {
                    /* tell the system how many bytes for this file */
                    memset(&ioInfo,0,sizeof(IOInfo));
                    ioInfo.ioi_Command         = FILECMD_SETEOF;
                    ioInfo.ioi_Offset          = dataSize;
                    result = DoIO(ioReqItem, &ioInfo);
                    if (result >= 0)
                    {
                        roundedSize = 0;
                        if (dataSize >= blockSize)
                        {
                            /* If we have more than one block's worth of
                             * data, write as much of it as possible.
                             */

                            roundedSize = (dataSize / blockSize) * blockSize;
                            ioInfo.ioi_Command         = CMD_WRITE;
                            ioInfo.ioi_Send.iob_Buffer = (void *)data;
                            ioInfo.ioi_Send.iob_Len    = roundedSize;
                            ioInfo.ioi_Offset          = 0;
                            result = DoIO(ioReqItem, &ioInfo);
```

```
                        data      = (void *)((uint32)data + roundedSize);
                        dataSize -= roundedSize;
                    }

                    if ((result >= 0) && dataSize)
                    {
                        /* If the amount of data left isn't as large
                         * as a whole block, we must allocate a memory
                         * buffer of the size of the block, copy the
                         * rest of the data into it, and write the
                         * buffer to disk.
                         */

                        temp = AllocMem(blockSize,MEMTYPE_DMA |

                                        MEMTYPE_FILL);
                        if (temp)
                        {
                            memcpy(temp,data,dataSize);
                            ioInfo.ioi_Command         = CMD_WRITE;
                            ioInfo.ioi_Send.iob_Buffer = temp;
                            ioInfo.ioi_Send.iob_Len    = blockSize;
                            ioInfo.ioi_Offset          = roundedSize;
                            result = DoIO(ioReqItem, &ioInfo);

                            FreeMem(temp,blockSize);
                        }
                        else
                        {
                            result = NOMEM;
                        }
                    }
                }
            }
        }
        DeleteIOReq(ioReqItem);
    }
    else
    {
        result = ioReqItem;
    }
    CloseDiskFile(fileItem);
}
else
{
    result = fileItem;
}

/* don't leave a potentially corrupt file around... */
if (result < 0)
```

```
            DeleteFile((char *)name);
    }

    return (result);
}


/*****************************************************************************/


static char testData[] = "This is some test data to save out";

int main(int32 argc, char **argv)
{
Err err;

    err = SaveDataFile("/NVRAM/test",testData,sizeof(testData));
    if (err < 0)
    {
        printf("Could not save data: ");
        PrintfSysErr(err);
    }
}
```

Displaying Contents of a File

Example 2 is a program listing that demonstrates how to read a file using the byte stream calls, and output its contents to the debugging terminal.

Example 2: *Displaying the contents of a 3DO file (type.c)*

```
#include "types.h"
#include "stdio.h"
#include "filestream.h"
#include "filestreamfunctions.h"


/*****************************************************************************/


#define MAXLINE 1024

static void Type(const char *path)
{
Stream *stream;
char    c;
char    line[MAXLINE];
int32   linelen;
int32   lines;
int32   endOfLine, endOfFile;
```

```
/* Open the file as a byte-stream. A buffer size of zero is
 * specified, which permits the File folio to choose an appropriate
 * amount of buffer space based on the file's actual block size.
 */

stream = OpenDiskStream((char *)path, 0);
if (!stream)
{
    printf("File '%s' could not be opened\n", path);
    return;
}

lines     = 0;
linelen   = 0;
endOfLine = FALSE;
endOfFile = FALSE;

/* Spin through a loop, grabbing one byte each time.
 *
 * A more efficient implementation would grab bytes
 * in larger batches (e.g. 128 bytes at a time) and parse them in a
 * more reasonable fashion.
 */

while (!endOfFile)
{
    if (ReadDiskStream(stream, &c, 1) < 1)
    {
        endOfFile = TRUE;
        endOfLine = TRUE;
        line[linelen] = '\0';
    }
    else if (c == '\r' || c == '\n')
    {
        endOfLine = TRUE;
        line[linelen] = '\0';
    }
    else if (linelen < MAXLINE-1)
    {
        line[linelen++] = c;
    }

    if (endOfLine)
    {
        printf("%s\n", line);
        linelen = 0;
        lines++;
        endOfLine = FALSE;
    }
}
```

```
    printf("\n%d lines processed\n\n", lines);

    /* close the file */
    CloseDiskStream(stream);
}


/*****************************************************************************/


int main(int32 argc, char **argv)
{
int32 i;

    if (argc < 2)
    {
        printf("Usage: type file1 [file2] [file3] [...]\n");
    }
    else
    {
        /* go through all the arguments */
        for (i = 1; i < argc; i++)
            Type(argv[i]);
    }

    return 0;
}
```

Scanning the File System

Example 3 is a program listing of an application that scans the file system.

Example 3: *Scanning the file system (Walker.c).*

```
#include "types.h"
#include "io.h"
#include "stdio.h"
#include "operror.h"
#include "filesystem.h"
#include "directory.h"
#include "directoryfunctions.h"
#include "filefunctions.h"


/*****************************************************************************/


static void TraverseDirectory(Item dirItem)
{
```

```
Directory      *dir;
DirectoryEntry  de;
Item            subDirItem;
int32           entry;

    dir = OpenDirectoryItem(dirItem);
    if (dir)
    {
        entry = 1;
        while (ReadDirectory(dir, &de) >= 0)
        {
            printf("Entry #%d:\n", entry);
            printf("  file '%s', type 0x%lx, ID 0x%lx",
                    de.de_FileName, de.de_Type,
                    de.de_UniqueIdentifier);
            printf(", flags 0x%lx\n", de.de_Flags);
            printf("  %d bytes, %d block(s) of %d byte(s) each\n",
                    de.de_ByteCount,
                    de.de_BlockCount, de.de_BlockSize);
            printf("  %d avatar(s)", de.de_AvatarCount);
            printf("\n\n");

            if (de.de_Flags & FILE_IS_DIRECTORY)
            {
                subDirItem = OpenDiskFileInDir(dirItem, de.de_FileName);
                if (subDirItem >= 0)
                {
                    printf("********** RECURSE **********\n\n");
                    TraverseDirectory(subDirItem);
                    printf("******* END RECURSION *******\n\n");
                }
                else
                {
                    printf("***  RECURSION FAILED  ***\n\n");
                }
            }
            entry++;
        }
        CloseDirectory(dir);
    }
    else
    {
        printf("OpenDirectoryItem() failed\n");
        CloseDiskFile(dirItem);
    }
}


/*****************************************************************************/
```

```
static Err Walk(const char *path)
{
Item startItem;

    startItem = OpenDiskFile((char *)path);
    if (startItem >= 0)
    {
        printf("\nRecursive directory scan from %s\n\n", path);
        TraverseDirectory(startItem);
        printf("End of %s has been reached\n\n", path);
        return 0;
    }
    else
    {
        printf("OpenDiskFile(\"%s\") failed: ",path);
        PrintfSysErr(startItem);
        return startItem;
    }
}


/*****************************************************************************/


int main(int32 argc, char **argv)
{
int i;

    if (argc <= 1)
    {
        /* if no directory name was given, scan the current directory */
        Walk(".");
    }
    else
    {
        for (i = 1; i < argc; i++)
            Walk(argv[i]);
    }

    return 0;
}
```

Listing the Contents of a Directory

Example 4 is a code listing that list the contents of a directory.

Example 4: *Listing a directory (ls.c)*

```
**/

#include "types.h"
#include "stdio.h"
#include "string.h"
#include "io.h"
#include "operror.h"
#include "filesystem.h"
#include "filefunctions.h"
#include "directory.h"
#include "directoryfunctions.h"


/****************************************************************************/


static void ListDirectory(const char *path)
{
Directory      *dir;
DirectoryEntry  de;
Item            ioReqItem;
int32           entry;
int32           err;
char            fullPath[FILESYSTEM_MAX_PATH_LEN];
IOInfo          ioInfo;
Item            dirItem;

    /* open the directory for access */
    dirItem = OpenDiskFile((char *)path);
    if (dirItem >= 0)
    {
        /* create an IOReq for the directory */
        ioReqItem = CreateIOReq(NULL, 0, dirItem, 0);
        if (ioReqItem >= 0)
        {
            /* Ask the directory its full name. This will expand any aliases
             * given on the command-line into fully qualified pathnames.
             */
            memset(&ioInfo, 0, sizeof(ioInfo));
            ioInfo.ioi_Command         = FILECMD_GETPATH;
            ioInfo.ioi_Recv.iob_Buffer = fullPath;
            ioInfo.ioi_Recv.iob_Len    = sizeof(fullPath);
            err = DoIO(ioReqItem, &ioInfo);
            if (err >= 0)
            {
                /* now open the directory for scanning */
                dir = OpenDirectoryPath((char *)path);
                if (dir)
                {
                    printf("\nContents of directory %s:\n\n", fullPath);
                    entry = 1;
```

```
                    while (ReadDirectory(dir, &de) >= 0)
                    {
                        printf("%5s", (char *) &de.de_Type);
                        printf(" %8d", de.de_ByteCount);
                        printf(" %4d*%4d", de.de_BlockCount, de.de_BlockSize);
                        printf(" %3d", de.de_AvatarCount);
                        printf(" %8x", de.de_Flags);
                        printf(" %s\n", de.de_FileName);
                        entry++;
                    }
                    CloseDirectory(dir);

                    printf("\nEnd of directory\n\n");
                }
                else
                {
                    printf("OpenDirectory(\"%s\") failed\n",fullPath);
                }
            }
            else
            {
                printf("Unable to get full path name for '%s': ",path);
                PrintfSysErr(err);
            }
            DeleteIOReq(ioReqItem);
        }
        else
        {
            printf("CreateIOReq() failed: ");
            PrintfSysErr(ioReqItem);
        }
        CloseDiskFile(dirItem);
    }
    else
    {
        printf("OpenDiskFile(\"%s\") failed: ",path);
        PrintfSysErr(dirItem);
    }
}


/****************************************************************************/


int main(int32 argc, char **argv)
{
int32 i;

    if (argc <= 1)
    {
        /* if no directory name was given, scan the current directory */
```

```
        ListDirectory(".");
    }
    else
    {
        /* go through all the arguments */
        for (i = 1; i < argc; i++)
            ListDirectory(argv[i]);
    }

    return 0;
}
```

File System Folio Function Calls

The following section lists the File folio calls. See Kernel Folio Calls in the *3DO System Programmer's Reference* for a complete description of the calls.

File Access Calls

- `CloseDiskFile()` Closes a file.
- `CreateAlias()` Creates an alias for a file.
- `CreateFile()` Creates a file.
- `DeleteFile()` Deletes a file.
- `OpenDiskFile()` Opens a disk file.
- `OpenDiskFileInDir()` Opens a disk file contained in a specific directory.

Program Calls

- `ExecuteAsSubroutine()` Executes previously loaded code as a subroutine.
- `ExecuteAsThread()` Executes previously loaded code as a thread.
- `LoadProgram()` Launches a program.
- `LoadProgramPrio()` Launches a program and gives it a priority.
- `LoadCode()` Loads a binary image into memory, and obtains a handle to it.
- `UnloadCode()` Unloads a binary image previously loaded with `LoadCode()`.

Directory Access Calls

- `ChangeDirectory()` Changes the current directory.
- `CloseDirectory()` Closes a directory.
- `GetDirectory()` Gets the item number and pathname for the current directory.
- `OpenDirectoryPath()` Opens a directory specified by a pathname.
- `OpenDirectoryItem()` Opens a directory.
- `ReadDirectory()` Reads the next entry from a directory.

Byte-Stream Calls

- `OpenDiskStream()` Opens a disk stream for byte stream-oriented I/O.

- `CloseDiskStream()` Closes a disk stream.
- `ReadDiskStream()` Reads from a disk stream.
- `SeekDiskStream()` Performs a seek operation on a disk stream.

CreateFile

Creates a file.

Synopsis

```
Item CreateFile( char *path )
```

Description

This function creates a new file.

Arguments

path
: The pathname for the file.

Return Value

The function returns a a value greater than or equal to 0 if the file is created successfully or a negative error code if an error occurs.

Implementation

SWI implemented in file folio V20.

Note: You do not need to delete the item returned by this function.

Associated Files

filefunctions.h

See Also

DeleteFile(), OpenDiskFile(), OpenDiskFileInDir()

DeleteFile

Deletes a file.

Synopsis

```
Err DeleteFile( char *path )
```

Description

This function deletes a file.

Arguments

path
: The pathname for the file to delete.

Return Value

The function returns zero if the file was successfully deleted or a negative error code if an error occurs.

Implementation

SWI implemented in file folio V20.

Associated Files

filefunctions.h

See Also

CreateFile(), OpenDiskFile(), OpenDiskFileInDir(), CloseDiskFile()

Debugging Memory Usage

Problems can occur when your application allocates or manages memory if the memory is misused or isn't freed up after use. A memory debugging module is provided to help spot memory management problems. The module replaces the memory allocation routines in your program with a superset that checks that the memory system is used correctly. These calls are activated when you do the following:

1. Add `CreateMemDebug()` as the first instruction in the `main()` routine of your program:

   ```
   Err CreateMemDebug(uint32 controlFlags, const TagArg *args);
   ```

   `controlFlags` is a set of bit flags that control various options of the memory debugging code; `args` is a pointer to an array of tag arguments containing extra data for this function. See the Kernel Folio Calls in the *3DO System Programmer's Reference* for a description of each flag. Currently, `args` must always be NULL.

2. Add `DumpMemDebug()` and `DeleteMemDebug()` as the last instructions in the `main()` routine of your program.

   ```
   Err DumpMemDebug(const TagArg *args);
   Err DeleteMemDebug(void);
   ```

   `args` is a pointer to an array of tag arguments containing extra data for this function. Currently `args` must always be NULL.

3. Recompile your entire application with `MEMDEBUG` defined on the ARM compiler's command line. For the ARM compiler, this is done by adding

   ```
   -DMEMDEBUG
   ```

   to the compile line.

4. Link your code with *memdebug.lib*

   A link order is important when using memdebug.lib. The memdebug library should come before *clib.lib*, but after everything else. That is, the link order should be:

   ```
   [many .lib files] memdebug.lib clib.lib
   ```

Other than `CreateMemDebug()`, `DumpMemDebug()`, and `DeleteMemDebug()`, there is no difference in the way your code works. When your program allocates memory, each memory allocation is tracked and managed. In addition, the memory debugging module makes sure that illegal or dangerous memory use is detected and flagged. When your program exits, any memory left allocated is displayed, along with the line number and source file from where memory was allocated.

When all the control flags are turned on, the debugging code checks and reports the following problems:

- Memory allocations of 0.
- Memory freed with a NULL or bogus memory pointer.
- Memory freed of a size that does not match the size of memory that was allocated.
- NULL memory lists passed to `AllocMemFromMemLists()` or `FreeMemToMemLists()`.
- Cookies on either side of all memory allocations are checked so they are not altered from the time a memory allocation is made to the time the memory is released. A change in the cookies indicates that your program is writing beyond the bounds of allocated memory.

When `MEMDEBUG` is defined during compilation, all standard memory allocation calls are automatically vectored through the debugging code. This includes memory allocation calls made in previously compiled .lib files with which you might be linking. However, you can get better debugging information if you recompile everything in your project, including .lib files, with `MEMDEBUG` defined.

If you call `DumpMemDebug()` at any time within your application, you can get a detailed listing of all memory currently allocated, showing the source line and source file from which the allocation occurred. This function also outputs statistics about general memory allocation patterns including: the number of memory allocation calls that have been performed, the maximum number of bytes allocated at any one time, current amount of allocated memory, and so on. All of this information is displayed on a per-thread basis, as well as globally for all threads. When using link libraries that haven't been recompiled with `MEMDEBUG` defined, the memory debugging subsystem will still can track the allocations, but will not report the source file or line number where the error occurred. It reports <unknown source file> instead.

An additional call, `SanityCheckMemDebug()`, can find problems with memory. This call checks outstanding memory allocations and checks all cookies to see if they have been corrupted. For example, if you want to find the location of a chunk of memory is being corrupted, you could place `SanityCheckMemDebug()` calls throughout the program. Look at the return data of each call until you locate the place in the program where the cookies are corrupted.

After you finish using any of these calls, turn off memory debugging before you create the final version of your program. Enabling memory debugging incurs an overhead of 32 bytes per allocation made. If you

use the MEMDEBUGF_PAD_COOKIES option, this overhead grows to 64 bytes per allocation.

Getting Information About Memory

The following sections explain how to find out how much memory is available, how to find out the type of memory you have, how to get the size of a memory page, and how to find out which VRAM bank (if any) contains a memory location.

Finding Out How Much Memory Is Available

You can find out how much memory is available by calling `AvailMem()`:

```
void AvailMem( MemInfo *minfo, uint32 memflags )
```

The `memflags` argument specifies the type of memory you want to find out about. If you want to find out how much of all types of memory is available, use `MEMTYPE_ANY` as the value of the flags argument. To find out how much DRAM is available, use `MEMTYPE_DRAM`; for VRAM, use `MEMTYPE_VRAM`. You can also use the flag `MEMTYPE_BANKSELECT` together with either `MEMTYPE_BANK1` or `MEMTYPE_BANK2` to get information about a specific VRAM bank. Other flags you can include are `MEMTYPE_DMA`, `MEMTYPE_CEL`, `MEMTYPE_AUDIO`, and `MEMTYPE_DSP`. For complete descriptions of these flags, see Allocating Memory.

Note: When you call `AvailMem()`, you can only request information about one memory type. Attempting to find out about more than one memory type can produce unexpected results.

The information about available memory is returned in a MemInfo structure pointed to by the `minfo` argument:

```
typedef struct MemInfo
{
    uint32 minfo_SysFree;
    uint32 minfo_SysLargest;
    uint32 minfo_TaskFree;
    uint32 minfo_TaskLargest;
} MemInfo;
```

This structure contains the following information:

- **minfo_SysFree**, The amount of memory, in bytes, of the specified memory type in the system-wide free memory pool. The pool contains only full pages of memory.

- **minfo_SysLargest**, The size, in bytes, of the largest series of contiguous pages of the specified memory type in the system-wide free memory pool.

- **minfo_TaskFree**, The amount of memory, in bytes, of the specified type in the task's free memory pool.

- **minfo_TaskLargest**, The size, in bytes, of the largest contiguous block of the specified memory type that can be allocated from the task's free memory pool.

  Warning: Do not depend on these numbers, as they may change if other tasks on the system allocate memory at the same time you are calling AvailMem.

Getting the Memory Type of a Memory Location

You can find out the memory type of a particular memory location by calling `GetMemType()`:

```
uint32 GetMemType( void *p )
```

The `p` argument is a pointer to the memory location to identify. The function returns memory allocation flags (such as `MEMTYPE_VRAM`) which describe the type of memory. For descriptions of these flags, see Allocating a Memory Block.

Getting the Size of a Memory Page

You can get the size of a memory page by calling the `GetPageSize()` function:

```
int32 GetPageSize( uint32 memflags )
```

The `memflags` argument contains memory allocation flags for the page size of the memory type you specify. There are currently only three page sizes: one for DRAM and two for VRAM. You can, however, include any of the following flags: `MEMTYPE_ANY`, `MEMTYPE_VRAM`, `MEMTYPE_DRAM`, `MEMTYPE_BANKSELECT`, `MEMTYPE_BANK1`, `MEMTYPE_BANK2`, `MEMTYPE_DMA`, `MEMTYPE_CEL`, `MEMTYPE_AUDIO`, and `MEMTYPE_DSP`. (For complete descriptions of these flags, see Allocating a Memory Block.) Normally, you should include just one flag which specifies the one type of memory in which you're interested. Including more than one flag can produce an unexpected result.

The return value contains the size of the page for the specified memory type, in bytes.

Finding Out Which VRAM Bank Contains a VRAM

Location

In the current 3DO hardware, 2 MB of video random access memory (VRAM) is divided into two banks, each containing 1 MB of VRAM. Fast VRAM memory transfers using the SPORT bus can take place only between memory locations in the same bank. You can find out which bank of VRAM-if any-contains a particular memory location by calling the `GetBankBits()` macro:

```
uint32 GetBankBits( void *a )
```

The `a` argument is a pointer to a memory location. The macro returns a set of memory flags in which the following flags can be set:

- **MEMTYPE_BANKSELECT**, Set if the memory is contained in VRAM.
- **MEMTYPE_BANK1**, Set if the memory is in VRAM bank 1.
- **MEMTYPE_BANK2**, Set if the memory is in VRAM bank 2.

If the memory is not in VRAM, the macro returns 0.

When a task allocates VRAM, the memory block contains memory from one VRAM bank or the other; it never contains memory from both banks. Thus, if any memory location in a block is in one VRAM bank, the entire block is in the same bank.

Validating Memory Pointers

When working with memory, sometimes it is necessary to verify the validity of memory pointers. This can be extremely useful when debugging, for example, in `ASSERT()` macros. Portfolio validates memory pointers that are passed to it, and rejects any corrupt ones.

The `IsMemReadable()` and `IsMemWritable()` functions let your code verify the validity of pointers.

```
bool IsMemReadable(void *mem, int32 size);
bool IsMemWritable(void *mem, int32 size);
```

To both functions, you supply a pointer to a memory region and the size of that region. The functions return TRUE if the current task or thread can access the complete memory region.

Allowing Other Tasks to Write to Your Memory

When a task owns memory, other tasks cannot write to that memory unless the task gives them permission. To give another task permission to write to one or more of your memory pages, or to revoke write permission that was previously granted, call `ControlMem()`:

```
Err ControlMem( void *p, int32 size, int32 cmd, Item task )
```

Each page of memory has a control status that specifies which task owns the memory and which tasks can write to it. Calls to `ControlMem()` change the control status for entire pages. The `p` argument, a pointer to a memory location, and the `size` argument, the amount of the contiguous memory, in bytes, beginning at the memory location specified by the `p` argument, specify which memory control status to change. If p and size arguments specify any part of a page, changes are made to the entire page. The `task` argument specifies which task to change the control status for. You can make multiple calls to `ControlMem()` to change the control status for more than one task. The `cmd` argument is a constant that specifies the change to be made to the control status. The possible values are given below:

- **MEMC_OKWRITE**. Grants permission to write to the memory of the task specified by the `task` argument. Currently, a task argument value of 0 is not supported for this operation.

- **MEMC_NOWRITE**. Revokes permission to write to the memory from the task specified by the task argument. A task argument value of 0 revokes the write permission for all tasks.

(One other possible value, `MEMC_GIVE`, is described in the next section.)

For a task to change the control status of memory pages with `ControlMem()`, it must own that memory, with one exception: a task that has write access to memory it doesn't own can relinquish its write access using `MEMC_NOWRITE` as the value of the `cmd` argument.

A task can use `ControlMem()` to prevent itself from writing to memory that it owns, which is useful during debugging.

Using Private Memory Pools

In addition to using private memory lists, a task can also use private memory pools. Each task gets a free memory pool when it is created. Tasks can also create additional free memory pools, which, like the pools created by the kernel, are simply lists of free memory lists. When a task creates a free memory pool containing its private memory lists, it can use a single function call to allocate or deallocate any of the types of memory in those lists. The functions for doing this are described in the following sections.

As with memory lists, tasks can only use memory pools that they own. This means that the system-wide free memory pool is off-limits to tasks, as are memory pools belonging to other tasks.

Creating a Memory Pool

To create a memory pool, you must prepare a List structure using the `InitList()` function:

```
void InitList(List *l, const char *name)
```

Once the List structure is initialized, you can then allocate some MemList structures using the technique described in the previous section. Every memory list you create can then be added to the memory pool by calling the `AddHead()` function:

```
void AddHead(List *l, Node *n)
```

Allocating Memory From a Specific Memory Pool

To allocate memory from a specific free memory pool, you use the `AllocMemFromMemLists()` function:

```
void *AllocMemFromMemLists( List *l, int32 size, uint32 memflags
)
```

The `l` argument is the free memory pool from which to allocate a memory block. As noted earlier, this pool must be owned by the calling task. The `size` argument specifies the size of the block to allocate, in bytes.

The `memflags` argument specifies the type of memory to allocate, the alignment of the block with respect to the page, and so on; these flags can include `MEMTYPE_ANY`, `MEMTYPE_VRAM`, `MEMTYPE_DRAM`, `MEMTYPE_BANKSELECT`, `MEMTYPE_BANK1`, `MEMTYPE_BANK2`,

MEMTYPE_DMA, MEMTYPE_CEL, MEMTYPE_AUDIO, MEMTYPE_DSP, MEMTYPE_FILL, MEMTYPE_INPAGE, MEMTYPE_SYSTEMPAGESIZE, MEMTYPE_STARTPAGE, and MEMTYPE_MYPOOL. For complete descriptions of these flags, see Allocating a Memory Block. The function returns a pointer to the block that was allocated, or NULL if there was not enough memory in the list to satisfy the request.

To free memory that you've allocated with AllocMemFromMemLists(), you can only use FreeMemToMemLists() (described in the next section). Each memory allocation function has a corresponding deallocation function; if you use another memory allocation function, you must use its corresponding deallocation function.

Freeing Memory to a Specific Memory Pool

To free a block of memory allocated with AllocMemFromMemLists(), you use the FreeMemToMemLists() function:

```
void FreeMemToMemLists( List *l, void *p, int32 size )
```

The l argument is a pointer to the memory pool from which the block was allocated. The p argument is a pointer to the block to free. The size argument is the size of the block to free, in bytes.

Getting Memory From the System-Wide Free Memory Pool

When there is insufficient memory in a task's free memory pool to allocate a block of memory, the kernel automatically provides additional memory pages from the system-wide free memory pool. Tasks can also get pages directly from the system-wide free memory pool by calling the `AllocMemBlocks()` function:

```
void *AllocMemBlocks( int32 size, uint32 typebits )
```

The `size` argument specifies the amount of memory to transfer, in bytes. If the specified size is not a multiple of the page size for the type of memory requested, the kernel transfers enough full pages to satisfy the request.

The first 4 bytes of the allocated block contain the actual number of bytes allocated. This isn't true when `MEMTYPE_FILL` is supplied.

The `typebits` argument specifies the type of memory to transfer. This argument can contain any of the following memory allocation flags: `MEMTYPE_ANY`, `MEMTYPE_VRAM`, `MEMTYPE_DRAM`, `MEMTYPE_BANKSELECT`, `MEMTYPE_BANK1`, `MEMTYPE_BANK2`, `MEMTYPE_DMA`, `MEMTYPE_CEL`, `MEMTYPE_AUDIO`, and `MEMTYPE_DSP`. For descriptions of these flags, see . You must always set the `MEMTYPE_TASKMEM` flag when using this function.

`AllocMemBlocks()` returns a pointer to the memory that was transferred. When you get the memory, you must call a function to free it, such as `FreeMemToMemList()` to move it to a specific memory list or `FreeMemToMemLists()` to move it to a specific memory pool. For either function, use the pointer returned by `AllocMemBlocks()` as the value of the `p` argument. Use the size value that is stored in the first 4 bytes in the memory returned by the `AllocMemBlocks()` for the size argument.

Example

Most memory allocation involves nothing more than calling `AllocMem()` to allocate memory and `FreeMem()` to free memory. Here is a simple example.

Example 1: *Allocating memory.*

```
#include "types.h"
#include "mem.h"
#include "stdio.h"

int main(int32 argc, char **argv)
{
void *memBlock;

    memBlock = AllocMem(123,MEMTYPE_ANY);  /* allocate any type of
memory */
    if (memBlock)
    {
        /* memBlock now contains the address of a block of memory
         * 123 bytes in size
         */

        FreeMem(memBlock,123); /* return the memory */
    }
    else
    {
        printf("Could not allocate memory!\n");
    }
}
```

Function Calls

The following list contains the memory calls. See Kernel Folio Calls in the *3DO System Programmer's Reference* for more information on these calls.

Allocating Memory

The following calls allocate memory:

- `AllocMem()` Allocates a specified memory block.
- `malloc()` Allocates memory.

Freeing Memory

The following calls free memory:

- `FreeMem()` Frees memory allocated with `AllocMem()`.
- `free()` Frees memory allocated with `malloc()`.

Getting Information About Memory

The following calls get information about available memory:

- `AvailMem()` Gets information about available memory.
- `GetBankBits()` Finds out which VRAM bank contains a memory location.
- `GetMemType()` Gets the type of the specified memory.
- `GetPageSize()` Gets the number of bytes in a memory page.

Reclaiming Unused Memory

The following call reclaims unused memory:

- `ScavengeMem()` Returns a task's unused memory pages to the system-wide memory pool.

Changing the Control Status of Memory

The following call changes the status of memory:

- `ControlMem()` Grants or revokes write permission to memory, or gives memory to another task.

Using Private Memory Lists

The following calls deal with private lists:

- `AllocMemFromMemList()` Allocates memory from a private memory pool.
- `FreeMemList()` Frees a memory list.
- `FreeMemToMemList()` Frees a private block of memory.

Using Memory Pools

The following calls allocate and free memory pools.

- `AllocMemBlocks()` Transfers pages of memory from the system-wide free memory pool.
- `AllocMemFromMemLists()` Allocates a memory block from a list of memory lists.
- `FreeMemToMemLists()` Returns a block of memory to a free memory pool.

Debugging Memory Problems

The following calls find problems with memory.

- `CreateMemDebug()` Initializes memory debugging package
- `DeleteMemDebug()` Releases memory debugging resources
- `DumpMemDebug()`Dumps memory allocation debugging information
- `SanityCheckMemDebug()`Checks all current memory allocations to make sure all of the allocation cookies are intact

Sharing System Resources

This chapter explains techniques for sharing system resources using semaphores. For details about the functions described here, see Kernel Folio Calls, in the *3DO System Programmer's Reference*.

This chapter contains the following topics:

About Semaphores

Tasks use special items known as *semaphores* to control access to shared resources. The following sections explain how they work.

The Problem: Sharing Resources Safely

One job of a multitasking operating system like Portfolio is to provide safe, orderly ways for tasks to share resources. A classic problem in computing is allowing multiple tasks to share the same resource. The problem affects any resource, including memory or a disk file, whose state can be changed by the tasks that share it. If a task changes a resource before another task finishes using it, and the change violates an assumption the first task has about the resource's contents, then crashes and corruption can occur.

The Solution: Locking Shared Resources When Using Them

To share a resource safely, a task must be able to lock the resource while it's using it and unlock it when it finishes using it. This is known as mutual exclusion: Other tasks are excluded from the resource until it is safe for them to use it.

The Implementation: Semaphores

There are many techniques for enforcing mutual exclusion. One of the simplest and most effective techniques uses special data structures known as semaphores. A semaphore keeps track of how many tasks are using or waiting to use a shared resource. This approach-described in all good textbooks on operating systems-is the one Portfolio uses. Tasks using Portfolio semaphores use the routines provided by the kernel for creating semaphores, for deleting them, and for locking and unlocking the resources they control.

On 3DO systems, it is up to each task to ensure that any shared resources it owns-in particular, memory pages that other tasks can write to-are shared safely. (To learn how tasks can grant other tasks write access to their memory, see Managing Memory.) Although tasks are not required to use semaphores, you should always use them, or another mutual-exclusion mechanism, when your task shares any of its memory with another task.

Using Semaphores

To control access to a shared resource it owns, a task

- Creates a semaphore
- Associates the semaphore with the resource
- Tells the other tasks that share the resource there is a semaphore for controlling access to the resource

Each task then must lock the semaphore before using the resource by calling `LockSemaphore()` and unlocks it as soon as it's done using it by calling `UnlockSemaphore()`. Except for creating the semaphore by calling `CreateSemaphore()`, sharing a resource requires the cooperation of the tasks that are involved. The tasks must agree on a way to communicate the information about the new semaphore, such as intertask messages or shared memory, and they need to agree on a format for this information. They must also agree to lock the semaphore before using the resource and unlock it when they're done.

If your task needs access to a shared resource and cannot continue without it, you can ask the kernel to put the task into wait state until the resource becomes available. You do this by including the `SEM_WAIT` flag in the `flags` argument of `LockSemaphore()` discussed below.

Creating a Semaphore

To create a semaphore, use `CreateSemaphore()`:

```
Item CreateSemaphore( const char *name, uint8 pri )
```

The `name` argument is the name to give to the new semaphore. You can use the `FindSemaphore()` macro described later in this chapter to find the semaphore by name. To ensure that `FindSemaphore()` finds the correct semaphore, you must provide a unique name for each semaphore you create.

The `pri` argument determines where this semaphore will be placed within the list of all semaphores in the system. A higher priority puts it closer to the start of the list. `CreateSemaphore()` returns the item number of the new semaphore or an error code (a negative value) if an error occurred.

To delete a semaphore created with `CreateSemaphore(),` use the `DeleteSemaphore()` function described in Deleting a Semaphore.

Deleting a Semaphore

To delete a semaphore, use the `DeleteSemaphore()` function:

```
Err DeleteSemaphore( Item s )
```

The `s` argument is the item number of the semaphore to delete. The function returns 0 if successful or a negative error code if an error occurred. As with all items, the item number of a deleted semaphore is not reused.

Locking a Semaphore

To lock a semaphore, thereby preventing other tasks from accessing the associated resource until your task is finished with it, call the `LockSemaphore()` function:

```
int32 LockSemaphore( Item s, uint32 flags )
```

The `s` argument is the item number of the semaphore for the resource you want to lock. Use the `flags` argument to specify what to do if the resource is already locked: if you include the `SEM_WAIT` flag, your task is put into wait state until the resource is available.

`LockSemaphore()` returns 1 if the resource was locked successfully. If the resource is already locked and you did not include the `SEM_WAIT` flag in the flags argument, the function returns 0. If an error occurs, the function returns a negative error code.

Unlocking a Semaphore

To unlock a semaphore that you've locked, use the `UnlockSemaphore()` function:

```
Err UnlockSemaphore( Item s )
```

The `s` argument is the item number of the semaphore for the resource you want to unlock. The function returns 0 if successful or an error code (a negative value) if an error occurred. It returns the `NOTOWNER` error code if your task did not call `LockSemaphore()` for the specified semaphore.

Finding a Semaphore

To find a semaphore by name, use the `FindSemaphore()` macro:

```
Item FindSemaphore( const char *name )
```

The `name` argument is the name of the semaphore to find. The macro returns the item number of the semaphore or a negative error code if an error occurred.

Function Calls

The following list contains the calls used to control resources.

Creating a Semaphore

The following call creates a semaphore:

- `CreateSemaphore()` Creates a semaphore.
- `CreateUniqueSemaphore()` Creates a semaphore with a unique name.

Locking a Resource

The following call locks a semaphore:

- `LockSemaphore()` Locks a semaphore.

Unlocking a Resource

The following call unlocks a semaphore:

- `UnlockSemaphore()` Unlocks a semaphore.

Finding a Semaphore

The following finds a semaphore:

- `FindSemaphore()` Finds a semaphore by name.

Deleting a Semaphore

The following deletes a semaphore:

- `DeleteSemaphore()` Deletes a semaphore

The International Folio

This chapter describes the International and the JString folios for use in creating titles suitable for international markets. For a complete description of the calls discussed in this chapter, see the International Folio Calls in the *3DO System Programmer's Reference*. This chapter contains the following topics:

International Folio Calls

This chapter describes the international folio procedure calls. The following table provides a brief description of each call.

- `intlCloseLocale` Terminate use of a locale item
- `intlCompareStrings` Compare two character strings
- `intlConvertString` Change attributes of a character string
- `intlFormatDate` Format a date according to a specified format
- `IntlFormatNumber` Format a number in a localized manner.
- `intlGetCharAttrs` Return attributes of a specified character
- `intlLookupLocale` Return a pointer to the locale structure for an Item
- `intlOpenLocale` Gain access to a locale item
- `intlTransliterateString` Convert a character string from one character set to another

intlCompareStrings

Compares two strings for collation purposes.

Synopsis

```
int32 intlCompareStrings(Item locItem,const unichar *string1,const
unichar *string2);
```

Description

Compares two strings according to the collation rules of the locale item's language.

Arguments

locItem
: A locale item, as obtained from intlOpenLocale().

string1
: The first string to compare.

string2
: The second string to compare.

Return Value

-1
: (string1 < string2)

0
: (string1 == string2)

1
: (string1>string2)

INTL_ERR_BADITEM
: locItem was not an existing locale item.

Implementation

Folio call implemented in international folio V24.

Associated Files

intl.h

See Also

`intlOpenLocale()`, `intlConvertString()`

intlConvertString

Changes certain attributes of a string.

Synopsis

```
int32 intlConvertString(Item locItem,const unichar *string,unichar
*result, uint32 resultSize,uint32 flags);
```

Description

Converts character attributes within a string. The flags argument specifies the type of conversion to perform.

Arguments

locItem
: A locale item, as obtained from intlOpenLocale().

string
: The string to convert.

result
: Where the result of the conversion is put. This area must be at least as large as the number of bytes in the string.

resultSize
: The number of bytes available in the result buffer. This limits the number of bytes that are put into the buffer.

flags
: Description of the conversion process to apply:

: INTL_CONVF_UPPERCASE will convert all characters to uppercase if possible.
: INTL_CONVF_LOWERCASE will convert all characters to lowercase if possible.
INTL_CONVF_HALFWIDTH will convert all FullKana characters to HalfKana.
INTL_CONVF_FULLWIDTH will convert all HalfKana characters to FullKana.
INTL_CONVF_STRIPDIACRITICALS will remove diacritical marks from all characters.

You can also specify:

(`INTL_CONVF_UPPERCASE|INTL_CONVF_STRIPDIACRITICALS`) or (`INTL_CONVF_LOWERCASE|INTL_CONVF_STRIPDIACRITICALS`) in order to perform two conversions in one call. If flags is 0, then a straight copy occurs.

Return Value

If positive, returns the number of characters in the result buffer. If negative, returns an error code. The string copied into the result buffer is guaranteed to be NULL-terminated.

> = 0
: Number of characters copied.

INTL_ERR_BADSOURCEBUFFER
: The string pointer supplied was bad.

INTL_ERR_BADRESULTBUFFER
: The result buffer pointer was NULL or wasn't invalid writable memory.

INTL_ERR_BUFFERTOOSMALL
: There was not enough room in the result buffer.

INTL_ERR_BADITEM
: locItem was not an existing locale item.

Implementation

Folio call implemented in international folio V24.

Associated Files

intl.h

Caveats

This function varies in intelligence depending on the language bound to the Locale argument. Specifically, most of the time, all characters above 0x0ff are not affected by the routine. The exception is with the Japanese language, where the Japanese characters are also affected by this routine.

See Also

intlOpenLocale(), intlCompareStrings()

intlFormatDate

Formats a date in a localized manner.

Synopsis

```
int32 intlFormatDate(Item locItem,DateSpec spec,const GregorianDate
*date,unichar *result, uint32 resultSize);
```

Description

This function formats a date according to a template and to the rules specified by the locale item provided. The formatting string works in a manner similar to the way `printf()` formatting strings are handled, but uses some specialized format commands tailored to date generation. The following format commands are supported:

%D - day of month

%H - hour using 24-hour style

%h - hour using 12-hour style

%M - minutes

%N - month name

%n - abbreviated month name

%O - month number

%P - AM or PM strings

%S - seconds

%W - weekday name

%w - abbreviated weekday name

%Y - year

In addition to straight format commands, the formatting string can also specify a field width, a field limit, and a field pad character. This is done in a manner identical to the way `printf()` formatting strings specify these values. That is

```
%[flags][width][.limit]command
```

where flags can be "-" or "0", width is a positive numeric value, limit is a positive numeric value, and command is one of the format commands mentioned above. Refer to documentation on the standard C `printf()` function for more information on how these numbers and flags interact.

A difference with standard `printf()` processing is that the limit value is applied starting from the rightmost digits, instead of the leftmost. For example;

```
%.2Y
```

Prints the rightmost two digits of the current year.

Arguments

locItem
: A locale item, as obtained from intlOpenLocale().

spec
: The formatting template describing the date layout. This value is typically taken from the locale structure (loc_Date, loc_ShortDate, loc_Time, loc_ShortTime), but it can also be built up by the title for custom formatting.

date
: The date to convert into a string.

result
: Where the result of the formatting is put.

resultSize
: The number of bytes available in the result buffer. This limits the number of bytes that are put into the buffer.

Return Value

If positive, then the number of characters in the result buffer. If negative, then an error code. The string

copied into the result buffer is guaranteed to be NULL-terminated.

> = 0
: Number of characters copied.

INTL_ERR_BADRESULTBUFFER

The result buffer pointer was NULL or wasn't invalid writable memory.

INTL_ERR_BUFFERTOOSMALL

There was not enough room in the result buffer.

INTL_ERR_BADDATESPEC
: The pointer to the DateSpec array was bad.

INTL_ERR_BADITEM
: locItem was not an existing locale item.

INTL_ERR_BADDATE
: The date specified in the GregorianDate structure is not a valid date. For example, the gd_Month is greater than 12.

Implementation

Folio call implemented in international folio V24.

Associated Files

intl.h

See Also

`intlOpenLocale()`, `IntlFormatNumber()`

IntlFormatNumber

Format a number in a localized manner.

Synopsis

```
int32 intlFormatNumber(Item locItem,const NumericSpec *spec,uint32
whole, uint32 frac,bool negative, bool doFrac,unichar *result, uint32
resultSize);
```

Description

This function formats a number according to the rules contained in the NumericSpec structure. The NumericSpec structure is normally taken from a Locale structure. The Locale structure contains three initialized NumericSpec structures (loc_Numbers, loc_Currency, and loc_SmallCurrency) which let you format numbers in an appropriate manner for the current system.

You can create your own NumericSpec structure which lets you use `intlFormatNumber()` to handle custom formatting needs. The fields in the structure have the following meaning:

- ns_PosGroups =A GroupingDesc value defining how digits are grouped to the left of the decimal character. A GroupingDesc is simply a 32-bit bitfield. Every ON bit in the bitfield indicates that the separator sequence should be inserted after the associated digit. So if the third bit (bit #2) is ON, it means that the grouping separator should be inserted after the third digit of the formatted number.

- ns_PosGroupSep =A string used to separate groups of digits to the left of the decimal character.

- ns_PosRadix =A string used as a decimal character.

- ns_PosFractionalGroups A GroupingDesc value defining how digits are grouped to the right of the decimal character.
- ns_PosFractionalGroupSep = A string used to separate groups of digits to the right of the decimal character.

- ns_PosFormat =This field is used to do post-processing on a formatted number. This is typically used to add currency notation around a numeric value. The string in this field is used as a format string in a `sprintf()` function call, and the formatted numeric value is supplied as a parameter to the same `sprintf()` call. For example, if the ns_PosFormat field is defined as "$%s", and the formatted numeric value is "0.02", then the result of the post-processing will be "$0.02".

When this field is NULL, no post-processing occurs.

- ns_PosMinFractionalDigits =Specifies the minimum number of digits to display to the right of the decimal character. If there are not enough digits, then the string will be padded on the right with 0s.

- ns_PosMaxFractionalDigits= Specifies the maximum number of digits to display to the right of the decimal character. Any excess digits are just removed.

- ns_NegGroups, ns_NegGroupSep, ns_NegRadix, ns_NegFractionalGroups, ns_NegFractionalGroupSep, ns_NegFormat, ns_NegMinFractionalDigits, ns_NegMaxFractionalDigits = These fields have the same function as the eight fields described above, except that they are used to process negative amounts, while the previous fields are used for positive amounts.

- ns_Zero =If the number being processed is 0, then this string pointer us used as-is'andiscopieddirectlyintotheresultbuffer.'If this field is NULL, the number is formatted as if it were a positive number.

- ns_Flags =This is reserved for future use and must always be 0.

Arguments

locItem
: A Locale Item, as obtained from intlOpenLocale().

spec
: The formatting template describing the number layout. This structure is typically taken from a Locale structure (loc_Numbers, loc_Currency, loc_SmallCurrency), but it can also be built up by the title for custom formatting.

whole
: The whole component of the number to format. (The part of the number to the left of the radix character.)

frac
: The decimal component of the number to format. (The part of the number to the right of the radix character.). This is specified in number of billionth. For example, to represent .5, you would use 500000000. To represent .0004, you would use 400000.

negative

TRUE if the number is negative, and FALSE if the number is positive.

doFrac
: TRUE if a complete number with a decimal mark and decimal digits is desired. FALSE if only the whole part of the number should be output.

result
: Where the result of the formatting is put.

resultSize
: The number of bytes available in the result buffer. This limits the number of bytes which are put into the buffer.

Return Value

If positive, then the number of characters in the result buffer. If negative, then an error code. The string copied into the result buffer is guaranteed to be NULL-terminated.

greater than or equal to 0 Number of characters copied.

INTL_ERR_BADNUMERICSPEC - The pointer to the NumericSpec structure was bad.

INTL_ERR_NORESULTBUFFER- "result" was NULL.

INTL_ERR_BUFFERTOOSMALL - There was not enough room in the result buffer.

INTL_ERR_BADITEM loc- Item was not an existing Locale Item.

Implementation

Folio call implemented in international folio V24.

Associated Files

intl.h

See Also

`intlOpenLocale()`, `intlFormatDate()`

intlGetCharAttrs

Returns attributes describing a given character.

Synopsis

```
uint32 intlGetCharAttrs(Item locItem, unichar character);
```

Description

This function examines the provided UniCode character and returns general information about it.

Arguments

locItem
: A locale item, as obtained from intlOpenLocale().

character
: The character to get the attribute of.

Return Value

Returns a bit mask, with bit sets to indicate various characteristics as defined by the UniCode standard. The possible bits are:

This character is uppercase.

INTL_ATTRF_LOWERCASE
: This character is lowercase.

INTL_ATTRF_PUNCTUATION
: This character is a punctuation mark.

INTL_ATTRF_DECIMAL_DIGIT
: This character is a numeric digit.

INTL_ATTRF_NUMBER
: This character represent a numerical value not representable as a single decimal digit. For example, a character 0x00bc represents the constant 1/2.

INTL_ATTRF_NONSPACING
: This character is a nonspacing mark.

INTL_ATTRF_SPACE
: This character is a space character.

INTL_ATTRF_HALF_WIDTH
: This character is HalfKana.

INTL_ATTRF_FULL_WIDTH
: This character is FullKana.

INTL_ATTRF_KANA
: This character is Kana (Katakana).

INTL_ATTRF_HIRAGANA
: This character is Hiragana.

INTL_ATTRF_KANJI
: This character is Kanji.

Implementation

Folio call implemented in international folio V24.

Associated Files

intl.h

Caveats

This function currently does not report any attributes for many upper UniCode characters. Only the ECMA Latin-1 character page (0x0000 to 0x00ff) is handled correctly at this time. If the language bound to the Locale structure is Japanese, then this function will also work correctly for Japanese characters.

See Also

`intlOpenLocale()`, `intlConvertString()`

intlTransliterateString

Converts a string between character sets.

Synopsis

```
int32 intlTransliterateString(const void *string,CharacterSets
stringSet,void *result,CharacterSets resultSet,uint32 resultSize,uint8
unknownFiller);
```

Description

Converts a string between two character sets. This is typically used to convert a string from or to UniCode. The conversion is done as well as possible. If certain characters from the source string cannot be represented in the destination character set, the unknownFiller byte will be inserted in the result buffer in their place.

Arguments

string
: The string to transliterate.

stringSet
: The character set of the string to transliterate. This describes the interpretation of the bytes in the source string.

result
: A memory buffer where the result of the transliteration can be put.

resultSet
: The character set to use for the resulting string.

resultSize
: The number of bytes available in the result buffer. This limits the number of bytes that are put into the buffer.

unknownFiller
: If a character sequence cannot be adequately converted from the source character set, then this byte will be put into the result buffer in place of the character sequence. When converting to a 16-bit character set, then this byte will be extended to 16-bits and inserted.

Return Value

If positive, returns the number of characters in the result buffer. If negative, returns an error code.

> = 0
: Number of characters in the result buffer.

INTL_ERR_BADSOURCEBUFFER

The string pointer supplied was bad.

INTL_ERR_BADCHARACTER

SET
: The "stringSet" or "resultSet" arguments did not specify valid character sets.

INTL_ERR_BADRESULTBUFFER

The result buffer pointer was NULL or wasn't invalid writable memory.

INTL_ERR_BUFFERTOOSMALL

There was not enough room in the result buffer.

INTL_ERR_BADITEM
: locItem was not an existing locale item.

Implementation

Folio call implemented in international folio V24.

Caveats

This function is not as smart as it could be. When converting from UniCode to ASCII, characters not in the first UniCode page (codes greater than 0x00ff) are never converted and always replaced with the unknownFiller byte. The upper pages of the UniCode set contain many characters which could be converted to equivalent ASCII characters, but these are not supported at this time.

Associated Files

intl.h

What Is Internationalization?

Internationalization is the process by which software is developed and modified so it transparently adapts to multiple cultural environments. *Localization* is the process of preparing specialized software, or special versions of existing software, targeted to individual cultural environments.

Internationalization of software has the following advantages:

- It lets you maintain a single set of source code for each title, instead of slightly different source code for each target culture. A single set of source considerably reduces the testing burden and increases confidence in code product reliability.

- It potentially lets both you and The 3DO Company deal with fewer master CD-ROMs than would otherwise be required. Instead of producing a different CD-ROM for each target culture, a single CD-ROM can support all cultures.

The International folio provides a structure and calls that behave in an expected manner to create a title for international environments. For example, using the 3DO International folio, your application can determine the current language and country codes, display dates, currency, and numeric values that are consistent with the current language and country codes of a target culture. The International folio also works with a separate folio, JString, to convert character sets from one format to another.

The International folio uses UniCode as its base character set. The functions and structures within the folio use the unichar data type, which is a 16-bit version of the standard char type.

UniCode is a standard defining a 16-bit character set, which is an international version of ASCII. UniCode specifies enough glyphs to represent all languages currently spoken around the world.

Although UniCode is the standard character set the International folio uses, it must support other character sets. For example, the Western computer world currently relies extensively on ASCII, and Japan relies on JIS and shift JIS. The International folio provides tools to convert strings from one character set to another, such as enabling applications to convert ASCII text into UniCode. The following books provide additional information on the UniCode standard:

- *The UniCode Standard Worldwide Character Encoding* Version 1.0, Volume 1 ISBN 0-201-56788-1

- *The UniCode Standard Worldwide Character Encoding* Version 1.0, Volume 2 ISBN 0-201-60845-6

The Locale Data Structure

The Locale data structure is like a little data base the system provides. Once you get a pointer to the Locale structure, it has, among other things, the current language being used, the country code, and the format of dates for the user's cultural environment.

The system creates a Locale structure when an application calls `intlOpenLocale()`. `intlOpenLocale()` returns an `Item`, which must be disposed of using the `intlCloseLocale()` macro.

To examine the contents of the Locale structure, an application calls `intlLookupLocale()`, which returns a pointer to the Locale structure for the specified `Item`. The definition of the Locale structure is shown below.

```
typedef struct Locale
{
    ItemNode          loc_Item;              /* system linkage */

    /* prefered language to use */
    LanguageCodes     loc_Language;

    /* An array of dialects for the current language, listed in order
     * of preference, and terminated with INTL_DIALECTS_ARRAY_END
     */
    DialectCodes     *loc_Dialects;

    /* ISO-3166 numeric-3 code for the user's country */
    CountryCodes      loc_Country;

    /* general description of the user's environment */
     int32             loc_GMTOffset;     /* minutes from GMT */
    MeasuringSystems  loc_MeasuringSystem; /* measuring system to use */
    CalendarTypes     loc_CalendarType;   /* calendar type to use */
    DrivingTypes      loc_DrivingType;    /* side of the street */

    /* number formatting */
    NumericSpec       loc_Numbers;
    NumericSpec       loc_Currency;
    NumericSpec       loc_SmallCurrency;

    /* date formatting */
    DateSpec          loc_Date;
    DateSpec          loc_ShortDate;
    DateSpec          loc_Time;
    DateSpec          loc_ShortTime;
} Locale;
```

The fields in the `Locale` structure are as follows:

- `loc_Item` provides system linkage for Locale structures which are allocated in system space so that they can be shared among multiple applications.
- `loc_Language` defines the language to use within an application. Each supported language has a code, which is taken from the ISO 639 Standard.
- `loc_Dialects` is a pointer to an array of dialects. Regional variations within a given language are accommodated through dialect tables, which are an array of dialect codes, terminated by `INTL_DIALECT_ARRAY_END`.
- The dialects appear in the array in decreasing order of user preference. For example, if the language is `INTL_LANG_ENGLISH`, then the dialect array could hold `INTL_ED_AMERICAN`, `INTL_ED_AUSTRALIAN`, `INTL_ED_BRITISH`, and so on.
- `loc_Country` contains the standard international country code for the current country. These codes are taken from the ISO 3166 Standard.
- `loc_GMTOffset` contains the offset in minutes of the current location relative to the standard GMT reference point.
- `loc_MeasuringSystem` indicates the measuring system to use. This can be `INTL_MS_METRIC`, `INTL_MS_AMERICAN`, or `INTL_MS_IMPERIAL`.
- `loc_CalendarType` indicates what type of calendar to use. This can be the traditional Gregorian calendar, with either Monday or Sunday as the first day of the week, or it can be the Arabic, Jewish, or Persian calendar.
- `loc_DrivingType` indicates on which side of the street cars usually travel in the current country.
- `loc_Numbers`, `loc_Currency`, `loc_SmallCurrency` specifies how to format numbers and currency. The `NumericSpec` structure contains the necessary information to properly localize number output and input. These three `NumericSpec` structures can be passed directly to the `intlFormatNumber()` function to apply localized formatting.
- `loc_Date`, `loc_ShortDate`, `loc_Time`, `loc_ShortTime` specifies how to format dates and time. The DateSpec array contains the necessary information needed to properly localize date and time output and input. These four `DateSpec` arrays can be passed directly to the `intlFormatDate()` function to apply localized formatting.

NumericSpec Structure

Each culture has its own way of writing numbers and currency. The Locale structure contains three NumericSpec

structures that contain number and currency formatting specifications and are used to produce correctly localized output for the target cultures.

The NumericSpec structure defines how numbers should be formatted. It is sufficiently flexible to cover plain numbers and currency values. The structure is shown below.

```
typedef struct NumericSpec
{
  /* how to generate a positive number */
  GroupingDesc  ns_PosGroups;  /* grouping description */
  unichar      *ns_PosGroupSep;  /* separates the groups */
  unichar      *ns_PosRadix;    /* decimal mark         */
  GroupingDesc  ns_PosFractionalGroups; /* grouping description */
  unichar      *ns_PosFractionalGroupSep; /* separates the groups */
  unichar      *ns_PosFormat;           /* for post-processing  */
  uint32              ns_PosMinFractionalDigits; /* min # of frac digits */
  uint32       ns_PosMaxFractionalDigits; /* max # of frac digits */

  /* how to generate a negative number */
  GroupingDesc  ns_NegGroups;   /* grouping description */
  unichar      *ns_NegGroupSep; /* separates the groups */
  unichar      *ns_NegRadix;   /* decimal mark        */
  GroupingDesc  ns_NegFractionalGroups;   /* grouping description */
  unichar      *ns_NegFractionalGroupSep;  /* separates the groups */
  unichar      *ns_NegFormat;              /* for post-processing  */
  uint32       ns_NegMinFractionalDigits; /* min # of frac digits */
  uint32       ns_NegMaxFractionalDigits; /* max # of frac digits */

  /* when the number is zero, this string is used 'as-is' */
  unichar          *ns_Zero;
} NumericSpec;
```

Using the fields in the NumericSpec structure, the `intlFormatNumber()` function can correctly format numbers and currency according to local rules and customs. The fields of the NumericSpec structure are as follows:

- `ns_PosGroups` defines how digits are grouped left of the decimal mark. An `EgroupingDesc` is a 32-bit bitfield in which every ON bit in the bitfield indicates a separator sequence should be inserted after the associated digit. If bit 0 is ON, the grouping separator is inserted after digit 0 of the formatted number.
- `ns_PosGroupSep` is a string that separates groups of digits to the left of the decimal mark.
- `ns_PosRadix` is a string used as a decimal character.
- `ns_PosFractionalGroups` is a `GroupingDesc` array that defines how digits are grouped to the right of the decimal character.

- `ns_PosFractionalGroupSep` is a string used to separate groups of digits to the right of the decimal mark.

- `ns_PosFormat` is used for post-processing on a formatted number. Typically it is used to add currency notation around a numeric value. The string in this field is used as a format string in `sprintf()`, and the formatted numeric value is also a parameter to `sprintf()`.

For example, if `ns_PosFormat` is defined as $%s, and the formatted numeric value is 0.02. The result of the post-processing will be $0.02. When this field is NULL, no post-processing occurs.

- `ns_PosMinFractionalDigits` specifies the minimum number of characters to the right of the decimal mark. If there are fewer characters than the minimum, the number is padded with 0s.

- `ns_PosMaxFractionalDigits` specifies the maximum number of characters to the right of the decimal mark. If there are more characters than the maximum, the string is truncated.

- `ns_NegGroups` is similar to `ns_PosGroups`, except it specifies negative numbers.

- `ns_NegGroupSep` is `similar to ns_PosGroupSep but for negative numbers.`

- `ns_NegRadix is similar to ns_PosRadix except it specifies negative numbers.`

- `ns_NegFractionalGroups is similar to ns_PosFractionalGroups, except it specifies negative numbers.`

- `ns_NegFractionalGroupSep` is similar to `ns_PosFractionalGroupSep,` except it specifies negative numbers.

- `ns_NegFormat similar to ns_PosFormat, except it specifies negative numbers.`

- `ns_NegMinFractionalDigits` is similar to `ns_PosMinFractionalDigits`, `except it specifies` negative numbers.

- `ns_NegMaxFractionalDigits` is similar to `ns_PosMaxFractionalDigits`, `except it specifies` negative numbers.

- `ns_Zer` If the number being processed is 0, then this string pointer is used as is and is copied directly into the resulting buffer. If this field is NULL, then the number is formatted as if it were a positive number.

Typically, an application doesn't have to deal with the interpretation of a NumericSpec structure. The Locale structure contains three NumericSpec structures initialized with the correct information for the target culture,

which can be passed to `intlFormatNumber()` to convert numbers into string form.

DateSpec Arrays

Dates also vary in format across different cultures. The Locale structure contains four `DateSpec` arrays which define the local formats for date and time representation.

The `DateSpec` arrays contain format commands similar to `printf()` format strings. The% commands are inserted in the formatting string to produce custom date output. An application can provide the `DateSpec` structure from the Locale structure to the `intlFormatDate()` function or it can create its own DateSpec structures for custom formatting.

Using the Locale Structure

The main use of the International folio is to obtain the target language and country codes. The folio also provides tools to format dates, numbers, and currency for the target language and country. This section describes how to use these tools.

Determining the Current Language and Country

The `intlOpenLocale()` call determines the current user settings for language or country:

```
Item intlOpenLocale(const TagArg *tags)
```

This call accepts one argument, `tags`, a pointer to a tag argument list. This pointer is currently unused and should be set to NULL. `intlOpenLocale()` returns an `Item`.

The application should then use the following call to obtain the Locale structure for the `Item`:

```
Locale *intlLookupLocale(Item locItem)
```

This call accepts one argument, `locItem`, an Item as returned from `intlOpenLocale()`. The language code in the `Locale` structure can then determine which set of messages and artwork to use.

For example, if the language code were "de," the application would use a directory containing the German message text and artwork ("de" being the code for Deutsch). The application has a directory containing message text and artwork for each language code it uses.

Note: An application must supply any text or artwork it displays. The application must supply text strings or artwork for each language it supports.

When an application finishes using an Item, use the following call to terminate its use of it:

```
Err intlCloseLocale(Item locItem)
```

`intlCloseLocale()` l accepts one argument, `locItem`, the item to be closed. It returns zero if the call was successful, otherwise, an error code is returned.

Working With International Character Strings

Comparing Character Strings

Applications use the `intlCompareStrings()` function to compare strings and sort text to be displayed. Each language sorts text in different ways. `intlCompareStrings()` adapts its behavior to the current language. By using this function, your title sorts text in the appropriate manner for the target culture:

```
int32 intlCompareStrings(Item locItem, const unichar *string1,
const char *string2)
```

This call accepts three arguments: `locItem` is a Locale Item obtained from `intlOpenLocale()`, `string1` is a pointer to the first string to compare, and `string2` is a pointer to the second string to compare.

The comparison is performed according to the collation rules of `locItem`. `intlCompareStrings()` is similar to the standard C `strcmp()` function, except that it handles sorting variations of different languages.

`IntlCompareStrings()` returns -1 if `string1` is less than `string2`; 0 if `string1` is equal to `string2`; and 1 if `string1` is greater than `string2`. `INTL_ERR_BADITEM` is returned if `locItem` was not a valid Item.

Changing Letters In Character Strings

`intlConvertString()` strips diacritical marks and changes the letter case of words or characters. This function handles the language differences in rules for case conversion. It is similar to the standard C `toupper()` and `tolower()` functions.

```
int32 intlConvertString(Item locItem, const unichar *string,
unichar *result, uint32 resultSize, uint32 flags);
```

The call accepts five arguments: `locItem` is an Item returned from `intlOpenLocale()`, `string` is the string to be changed, `result` is where the result of the conversion is placed, `resultSize` is the number of bytes available in `result`, and `flags` indicates the conversion to be performed. The string copied to result is guaranteed to be NULL-terminated.

If successful, `intlConvertString()` returns a positive number indicating the number of characters in `result`. Otherwise, it returns an error code.

Changing Character Sets

`intlTransliterateString()` converts a string from one character set to another character set.

The International folio always generates UniCode strings. For example, `intlFormatNumber()` generates a UniCode string holding the formatted number. The 3DO Portfolio currently supplies simple text output routines in Lib3DO which only accept ASCII text. To display the formatted number, an application must convert the UniCode string generated by the International folio to ASCII and call the text output routines.

`intlTransliterateString()` can be used to convert data files generated in ASCII to UniCode or to convert the output of the International folio into ASCII for use by other tools.

`intlTranliterateString()` is called as follows:

```
int32 intlTransliterateString(const void *string, CharacterSets
stringSet, void *result, CharacterSets resultSet, uint32
resultSize, uint8 unknownFiller);
```

The call accepts six arguments: `string` is the string to convert, `stringSet` is the character set of the string to convert, `result` is where to put the converted string, `resultSet` is the character set to use for the resulting string, `resultSize` is the number of bytes available in `result`, and `unknownFiller` is used as filler for characters that cannot be converted.

If successful, `intlTransliterateString()` returns a positive number indicating the number of characters in `result`. Otherwise, it returns an error code.

Providing C Functionality

`intlGetCharAttrs()` provides functionality similar to the standard C functions `isupper()`, `islower()`, etc. It is called as follows:

```
uint32 intlGetCharAttrs(Item locItem, unichar character);
```

The call accepts two arguments: `locItem`, an `Item` as returned by `intlOpenLocale()`; and `character`, the character for which attributes should be returned.

If successful, `intlGetCharAttrs()` returns a bit mask which indicates the attributes of the character. `INTL_ERR_BADITEM` is returned if `locItem` is not a valid `Item`.

Formatting Numbers or Currency

Numbers are represented in different ways in different countries, so a title should use the following call to format a number to be displayed in the correct manner:

```
int32 intlFormatNumber(Item locItem, const NumericSpec *spec,
uint32 whole, uint32 frac, bool negative, bool doFrac, unichar
*result, uint32 resultSize);
```

The call accepts seven arguments: `locItem` is an Item returned from `intlOpenLocale()`, `spec` is the format specification for the number (usually taken from the `Locale` structure), `whole` is the whole component of the number, `frac` is the decimal component of the number expressed in billionths (to the right of the decimal mark), `negative` is a Boolean that indicates whether the number is negative, `doFrac` is a flag indicating which portions of the number should be formatted (if TRUE, the entire number with a decimal mark is output, if FALSE, only the whole part of the number is output), `result` is where the formatted number is put, and `resultSize` is the number of bytes in `result`.

The number is formatted according to the rules specified in `locItem` and `spec`. The string copied to `result` is guaranteed to be NULL-terminated.

If successful, `intlFormatNumber()` returns a positive number indicating the number of characters in `result`. Otherwise, it returns an error code.

Formatting Dates

An application should use the following call to format a date to be displayed:

```
int32 intlFormatDate(Item locItem, DateSpec spec, const
GregorianDate *date, unichar *result, uint32 resultSize);
```

The call accepts five arguments: `locItem` is an Item as returned from `intlOpenLocale()`, `spec` is the format specification for the date (usually taken from the `Locale` structure), `date` is the date to format, `result` is where to put the formatted date, and `resultSize` is the number of bytes in `result`. The date is formatted according to the rules of `locItem` and `spec` arguments. The string copied into `result` is guaranteed to be NULL-terminated.

If successful, `intlFormatDate()` returns a positive number indicating the number of characters in result. Otherwise, an error code is returned.

Converting Character Sets With the JString Folio

The International folio uses calls from the JString folio to convert character sets from one format to another. The JString folio contains four calls:

- `ConvertASCII2ShiftJIS()` converts a string from ASCII to Shift-JIS.
- `ConvertShiftJIS2ASCII()` converts a string from Shift-JIS to ASCII.
- `ConvertShiftJIS2UniCode()` converts a string from Shift-JIS to UniCode.
- `ConvertUniCode2ShiftJIS()` converts a string from UniCode to Shift-JIS.

These calls are used to enter data or develop applications using the Shift-JIS character set and then convert the date to ASCII or UniCode at runtime.

For example, the International folio generates dates and currency in UniCode. Because the current 3DO text routines can only handle ASCII or Shift-JIS, you need to convert it to render the information on the display. You can parse the UniCode and render it using your own font technology, or use the 3DO JString calls to convert the text to Shift-JIS.

The International folio automatically loads the JString folio when you call `intlTransliterateString` and requests Shift-JIS translation.

Function Calls and Macros

Obtaining a Locale Structure

The following International folio macros obtain a Locale structure:

- `intlCloseLocale()` Terminates use of a Locale Item.
- `intlLookupLocale()` Returns a pointer to the Locale structure for an Item.
- `intlOpenLocale()` Gains access to a Locale Item.

Manipulating Characters and Dates

The following International folio calls manipulate characters, dates, and numbers:

- `intlCompareStrings()` Compares two character strings.
- `intlConvertString()` Changes attributes of a character string.
- `intlFormatDate()` Formats a date according to a specified format.
- `IntlFormatNumber()` Formats a number according to a specified format.
- `intlGetCharAttrs()`Returns attributes of a specified character.
- `intlTransliterateString()`Converts a character string from one character set to another.

Converting Character Sets

The following JString folio calls are used to convert from one character set to another.

- `ConvertASCII2ShiftJIS()`Converts a string from ASCII to Shift-JIS
- `ConvertShiftJIS2ASCII()` Converts a string from Shift-JIS to ASCII.
- `ConvertShiftJIS2UniCode()` Converts a string from Shift-JIS to UniCode.
- `ConvertUniCode2ShiftJIS()` Converts a string from UniCode to Shift-JIS.

ConvertASCII2ShiftJIS

Converts an ASCII string to Shift-JIS.

Synopsis

```
int32 ConvertASCII2ShiftJIS(const char *string,char *result, uint32
resultSize,uint8 unknownFiller);
```

Description

Converts a string from ASCII to Shift-JIS. The conversion is done as well as possible. If certain characters from the source string cannot be represented in ASCII, the unknownFiller byte will be inserted in the result buffer in their place.

Arguments

string
: The string to convert.

result
: A memory buffer where the result of the conversion can be put.

resultSize
: The number of bytes available in the result buffer. This limits the number of bytes that are put into the buffer.

unknownFiller
: If a character sequence cannot be adequately converted, then this byte will be put into the result buffer in place of the character sequence.

Return Value

If positive, returns the number of characters in the result buffer. If negative, returns an error code.

> = 0
: Number of characters in the result buffer.

JSTR_ERR_BUFFERTOO

SMALL
: There was not enough room in the result buffer.

Implementation

Folio call implemented in jstring folio V24.

Associated Files

jstring.h

ConvertShiftJIS2ASCII

Converts a Shift-JIS string to ASCII.

Synopsis

```
int32 ConvertShiftJIS2ASCII(const char *string,char *result, uint32
resultSize,uint8 unknownFiller);
```

Description

Converts a string from Shift-JIS to ASCII. The conversion is done as well as possible. If certain characters from the source string cannot be represented in ASCII, the unknownFiller byte will be inserted in the result buffer in their place.

Arguments

string
: The string to convert.

result
: A memory buffer where the result of the conversion can be put.

resultSize
: The number of bytes available in the result buffer. This limits the number of bytes that are put into the buffer.

unknownFiller
: If a character sequence cannot be adequately converted, then this byte will be put into the result buffer in place of the character sequence.

Return Value

If positive, then the number of characters in the result buffer. If negative, then an error code.

> = 0
: Number of characters in the result buffer.

JSTR_ERR_BUFFERTOO

SMALL
: There was not enough room in the result buffer.

Implementation

Folio call implemented in jstring folio V24.

Associated Files

jstring.h

ConvertShiftJIS2UniCode

Converts a Shift-JIS string to UniCode.

Synopsis

```
int32 ConvertShiftJIS2UniCode(const char *string,unichar *result,
uint32 resultSize,uint8 unknownFiller);
```

Description

Converts a string from Shift-JIS to UniCode. The conversion is done as well as possible. If certain characters from the source string cannot be represented in ASCII, the unknownFiller byte will be inserted in the result buffer in their place.

Arguments

string
: The string to convert.

result
: A memory buffer where the result of the conversion can be put.

resultSize
: The number of bytes available in the result buffer. This limits the number of bytes that are put into the buffer.

unknownFiller
: If a character sequence cannot be adequately converted, then this byte will be put into the result buffer in place of the character sequence.

Return Value

If positive, then the number of characters in the result buffer. If negative, then an error code.

> = 0
: Number of characters in the result buffer.

JSTR_ERR_BUFFERTOO

SMALL
: There was not enough room in the result buffer.

Implementation

Folio call implemented in jstring folio V24.

Associated Files

jstring.h

ConvertUniCode2ShiftJIS

Converts a UniCode string to Shift-JIS.

Synopsis

```
int32 ConvertUniCode2ShiftJIS(const unichar *string,char *result,
uint32 resultSize,uint8 unknownFiller);
```

Description

Converts a string from UniCode to Shift-JIS. The conversion is done as well as possible. If certain characters from the source string cannot be represented in ASCII, the unknownFiller byte will be inserted in the result buffer in their place.

Arguments

string
: The string to convert.

result
: A memory buffer where the result of the conversion can be put.

resultSize
: The number of bytes available in the result buffer. This limits the number of bytes that are put into the buffer.

unknownFiller
: If a character sequence cannot be adequately converted, then this byte will be put into the result buffer in place of the character sequence.

Return Value

If positive, then the number of characters in the result buffer. If negative, then an error code.

> = 0
: Number of characters in the result buffer.

JSTR_ERR_BUFFERTOO

SMALL
: There was not enough room in the result buffer.

Implementation

Folio call implemented in jstring folio V24.

Associated Files

jstring.h

The Compression Folio

This chapter describes the Compression folio that provides general purpose compression and decompression services. It contains the following topics:

Introduction

This section provides an overview of the Compression folio and describes how it works. See *3DO System Programmer's Reference* for complete descriptions of the calls mentioned in this section.

The compression folio provides services that compress and decompress data in a non-lossy manner. The compression quality differs depending on the type of data. Compression ratios range typically from 50 percent up to 80 percent.

The intended use of the Compression folio is to compress data that is being written to NVRAM. It is critical that data being written to NVRAM be as compact as possible because it is a limited resource. The interface to the compression and decompression engine is totally generic, allowing the folio to be used for any data that needs to be compressed.

3DO System Programmer's Reference

This book is a command reference. It contains examples of Portfolio procedure calls, syntax, and use. It contains these chapters:

File Folio Calls

This chapter lists the file folio calls in alphabetical order. The list below is a quick summary of each, followed by the page number where you'll find the procedure described.

Math Folio Calls

This chapter lists the math folio procedure calls in alphabetical order. The list below is a quick summary of each, followed by the page number where you'll find the procedure described.

- CosF16 Computes the 16.16 operamath cosine of a 16.16 fraction angle.
- CosF30 Computes the operamath 2.30 cosine of a 16.16 fraction.
- CosF32 Computes the operamath 32.32 cosine of a 16.16 fraction.
- Cross3_F16 Computes the cross-product of two vectors of 16.16 values.
- DivRemS32 Computes the quotient and remainder of a 32-bit division.
- DivRemSF16 Computes the quotient and remainder of a 16.16 division.
- DivRemU32 Computes the quotient and remainder of a 32-bit division.
- DivRemUF16 Computes the quotient and remainder of a 16.16 division.
- DivS64 Computes the quotient and remainder of a 64-bit division.
- DivSF16 Computes the quotient of a 16.16 division.
- DivU64 Computes the quotient and remainder of a 64-bit division.
- DivUF16 Computes the quotient of a 16.16 division.
- Dot3_F16 Multiplies two vectors of 16.16 values.
- Dot4_F16 Multiplies two vectors of 16.16 values.
- Mul32 Multiplies two 32-bit integers.
- Mul64 Multiplies two 64-bit integers.
- MulF14 Multiplies two 2.14 fractions.
- MulManyF16 Multiplies an array of 16.16 values by another array of 16.16 values.
- MulManyVec3Mat33_F16 Multiplies one or more vectors by a 3x3 matrix of 16.16 values.
- MulManyVec3Mat33DivZ_F16 Multiplies a 3x3 matrix of 16.16 values by 1 or more 3-coordinate vectors of 16.16 values, then multiplies x and y elements of the result vector by ratio of n/z.
- MulManyVec4Mat44_F16 Multiplies one or more vectors by a 4x4 matrix of 16.16 values.
- MulMat33Mat33_F16 Computes the product of two 3x3 matrices of 16.16values.
- MulMat44Mat44_F16 Computes the product of two 4x4 matrices of 16.16 values.
- MulObjectMat33_F16 Multiplies a matrix within an object structure by an external 3x3 matrix of 16.16 values.
- MulObjectMat44_F16 Multiplies a matrix within an object structure by an external 4x4 matrix of 16.16 values.
- MulObjectVec3Mat33_F16 Multiplies many vectors within an object structure by a 3x3 matrix of 16.16 values.
- MulObjectVec4Mat44_F16 Multiplies many vectors within an object structure by a 4x4 matrix of 16.16 values.
- MulS32_64 Multiplies two 32-bit integers and returns a 64-bit result.
- MulScalarF16 Multiplies a 16.16 scalar by an array of 16.16 values.
- MulSF16 Multiplies two signed 16.16 fractions.
- MulSF16_F32 Multiplies two signed 16.16 numbers and returns a 32.32 result.
- MulSF30 Multiplies two 2.30 fractions.
- MulSF30_F60 Multiplies two signed 2.30 numbers and returns a 4.60 result.

- MulU32_64 Multiplies two unsigned 32-bit integers and returns a 64-bit result.
- MulUF16 Multiplies two unsigned 16.16 fractions.
- MulUF16_F32 Multiplies two unsigned 16.16 numbers and returns a 32.32 result.
- MulUF30_F60 Multiplies two unsigned 2.30 numbers and returns a 4.60 result.
- MulVec3Mat33_F16 Computes the product of a 3x3 matrix and a vector.
- MulVec3Mat33DivZ_F16 Computes the product of a 3x3 matrix and a vector, then multiplies the x, y elements of the result by the ratio n/z.
- MulVec4Mat44_F16 Computes the product of a 4x4 matrix and a vector.
- Neg32 Computes the two's complement of a 32-bit integer.
- Neg64 Computes the two's complement of a 64-bit integer.
- NegF14 Computes the two's complement of a 2.14 fraction.
- NegF16 Computes the two's complement of a 16.16 fraction.
- NegF30 Computes the two's complement of a 2.30 fraction.
- NegF32 Computes the two's complement of a 32.32 fraction.
- NegF60 Computes the two's complement of a 4.60 fraction.
- Not32 Computes one's complement of a 32-bit integer.
- Not64 Computes one's complement of a 64-bit integer.
- NotF14 Computes one's complement of a 2.14 fraction.
- NotF16 Computes one's complement of a 16.16 fraction.
- NotF30 Computes one's complement of a 2.30 fraction.
- NotF32 Computes one's complement of a 32.32 number.
- NotF60 Computes one's complement of a 4.60 number.
- OpenMathFolio Opens the math folio and sets _MathBase global variable.
- RecipSF16 Computes the reciprocal of a signed 16.16 number.
- RecipUF16 Computes the reciprocal of an unsigned 16.16 number.
- SinF16 Computes the 16.16 sine of a 16.16 fraction angle.
- SinF30 Computes the operamath 2.30 sine of a 16.16 fraction.
- SinF32 Computes the operamath 32.32 sine of a 16.16 fraction.
- Sqrt32 Computes square root of an unsigned 32-bit number.
- Sqrt64_32 Computes the 32-bit square root of an unsigned 64-bit integer.
- SqrtF16 Computes the square root of an unsigned 16.16 number.
- SqrtF32_F16 Computes the square root of a 32.32 fraction.
- SqrtF60_F30 Computes the square root of a 4.60 fraction as a 2.30 fraction.
- Square64 Squares a 64-bit integer.
- SquareSF16 Squares a signed 16.16 fraction.
- SquareUF16 Squares an unsigned 16.16 fraction.
- Sub32 Subtracts a 32-bit integer from another.
- Sub64 Subtracts a 64-bit integer from another.
- SubF14 Subtracts a 2.14 fraction from another.

- SubF16 Subtracts a 16.16 fraction from another.
- SubF30 Subtracts a 2.30 fraction from another.
- SubF32 Subtracts a 32.32 fraction from another.
- SubF60 Subtracts a 4.60 fraction from another.
- Transpose33_F16 Transposes a 3x3 matrix of 16.16 values.
- Transpose44_F16 Transposes a 4x4 matrix of 16.16 values.

AbsVec3_F16

Computes the absolute value of a vector of 16.16 values.

Synopsis

```
frac16 AbsVec3_F16( vec3f16 vec )
```

Description

This function computes the absolute value of a vector of 16.16 values.

Arguments

vec
: The vector whose absolute value to compute.

Return Value

The function returns the absolute value (a 16.16 fraction).

Implementation

SWI implemented in operamath V20.

Associated Files

operamath.h

Caveats

This function does not detect overflow conditions and will return unpredictable results in case of overflow.

See Also

AbsVec4_F16()

AbsVec4_F16

Computes the absolute value of a vector of 16.16 values.

Synopsis

```
frac16 AbsVec4_F16( vec4f16 vec )
```

Description

This function computes the absolute value of a vector of 16.16 values.

Arguments

vec
: The vector whose absolute value to compute.

Return Value

The function returns the absolute value (a 16.16 fraction).

Implementation

SWI implemented in operamath V20.

Associated Files

operamath.h

Caveats

This function does not detect overflow conditions and will return unpredictable results in case of overflow.

See Also

AbsVec3_F16()

Add32

Adds two 32-bit integer quantities together.

Synopsis

```
int32 Add32( int32 x, int32 y )
```

Description

This macro adds two 32-bit numbers together and returns the result. The macro is included for completeness.

The macro is actually defined as simple addition of its arguments, and does not check for or enforce any typecast requirements.

Arguments

x, y 32-bit integers.

Return Value

The macro returns the sum of its arguments.

Implementation

Macro implemented in operamath.h V20.

Associated Files

operamath.h

See Also

AddF16(), AddF14(), AddF30()

AddF16

Adds two 16.16-format fixed-point fractions together.

Synopsis

```
frac16 AddF16( frac16 x, frac16 y )
```

Description

This macro adds two 16.16-format fixed-point fractions together and returns the result. The macro is included for completeness.

The macro is actually defined as simple addition of its arguments, and does not check for or enforce any typecast requirements.

Arguments

x, y 16.16-format fixed-point fractions.

Return Value

The function returns the sum of the arguments.

Implementation

Macro implemented in operamath.h V20.

Associated Files

operamath.h

See Also

Add32(), AddF14(), AddF30()

AddF14

Adds two 2.14-format fixed-point fractions together.

Synopsis

```
frac14 AddF14( frac14 x, frac14 y )
```

Description

This macro adds two 2.14-format fixed-point fractions together and returns the result. The macro is included for completeness.

The macro is actually defined as simple addition of its arguments, and does not check for or enforce any typecast requirements.

Arguments

x, y 2.14-format fixed-point fractions.

Return Value

The macro returns the sum of the arguments.

Implementation

Macro implemented in operamath.h V20.

Associated Files

operamath.h

See Also

Add32(), AddF16(), AddF30()

AddF30

Adds two 2.30-format fixed-point fractions together.

Synopsis

```
frac30 AddF30( frac30 x, frac30 y )
```

Description

This macro adds two 2.30 format fixed-point fractions together and returns the result. The macro is included for completeness.

The macro is actually defined as the simple addition of its arguments. It does not check for or enforce any typecast requirements.

Arguments

x, y 2.30-format fixed-point fractions.

Return Value

The macro returns the sum of its arguments.

Implementation

Macro implemented in operamath.h V20.

Associated Files

operamath.h

See Also

Add32(), AddF16(), AddF14()

Add64

Adds two 64-bit integers together.

Synopsis

```
void Add64( int64 *r, int64 *a1, int64 *a2 )
```

Description

This function adds two 64-bit integers together and returns the result. The value deposited in r is the sum of the arguments. This function is actually the same function as `AddF32()`.

Arguments

r
: A pointer to a 64-bit integer structure to store the result.

a1, a2
: Pointers to 64-bit integer addends.

Implementation

Folio call implemented in operamath V20.

Associated Files

operamath.h, operamath.lib

See Also

`AddF32()`, `AddF60()`

AddF32

Adds two 32.32-format fixed-point fractions together.

Synopsis

```
void AddF32( frac32 *r, frac32 *a1, frac32 *a2 )
```

Description

This function adds two 32.32- format fixed-point fractions together and deposits the result in the location pointed to by the r argument. This function is actually the same function as `Add64()`.

Arguments

r
: A pointer to a 32.32-fraction structure to store the result.

a1, a2
: Pointers to 32.32-format fixed-point fraction addends.

Implementation

Folio call implemented in operamath V20.

Associated Files

operamath.h, operamath.lib

See Also

`Add64()`, `AddF60()`

AddF60

Adds two 4.60-format fixed-point fractions together.

Synopsis

```
void AddF60( frac60 *r, frac60 *a1, frac60 *a2 )
```

Description

This function adds two 4.60-format fixed-point fractions together and deposits the result in the location pointed to by the r argument.

Arguments

r
: A pointer to a 4.60-fraction structure to store the result.

a1, a2
: Pointers to 4.60-format fixed-point fraction addends.

Implementation

Folio call implemented in operamath V20.

Associated Files

operamath.h, operamath.lib

See Also

Add64(), AddF32()

Atan2F16

Computes the arctangent of a ratio.

Synopsis

```
frac16 Atan2F16( frac16 x, frac16 y )
```

Description

This function computes the arctangent of the ratio y/x.

The result assumes 256.0 units in the circle (or 16,777,216 units if used as an integer). A correct 16.16 result is returned if the arguments are int32, frac30 or frac14, as long as both arguments are the same type.

Arguments

x, y 16.16-format fractions.

Return Value

The function returns the arctangent of the ratio of the arguments.

Implementation

Folio call implemented in operamath V20.

Associated Files

operamath.h, operamath.lib

CloseMathFolio

Closes the math folio and resets the mathbase global variable to zero.

Synopsis

```
Err CloseMathFolio(void)
```

Description

CloseMathFolio attempts to close the math folio structure. If successful, it also sets _Mathfolio to zero and returns zero. On failure a negative error is returned.

Implementation

Convenience call implemented in operamath.lib V22.

See Also

OpenMathFolio()

OpenMathFolio

Opens the math folio and sets _MathBase global variable.

Synopsis

```
int32 OpenMathFolio( void )
```

Description

This function attempts to open the math folio structure. If it successfully locates the math folio, it sets the global variable _MathBase to point to the math folio structure. If it fails to open the folio, it returns a negative number that can be printed with the PrintfSysError function.

Programs must call OpenMathFolio before attempting to use any of the math folio functions.

Return Value

If successful, the function sets the global variable _MathBase to point to the math folio structure. If unsuccessful, it returns a negative number.

Implementation

Convenience call implemented in operamath.lib V20.

Associated Files

operamath.h, operamath.lib

CompareS64

Compares two signed 64-bit integer quantities.

Synopsis

```
int32 CompareS64( int64 *s1, int64 *s2 )
```

Description

This function compares two signed 64-bit integers. This function is actually the same function as `CompareSF32()` and `CompareSF60()`.

Arguments

s1, s2 Pointers to signed 64-bit integers.

Return Value

s1 == s2

Implementation

Folio call implemented in operamath V20.

Associated Files

operamath.h, operamath.lib

See Also

`CompareSF32()`, `CompareU64()`, `CompareUF32()`, `CompareUF60()`, `CompareSF60()`

CompareSF32

Compares two signed 32.32-format fractions.

Synopsis

```
int32 CompareSF32( frac32 *s1, frac32 *s2 )
```

Description

This function compares two signed 32.32-format fractions. This function is actually the same function as `CompareS64()` and `CompareSF60()`.

Arguments

s1, s2 Pointers to signed 32.32 fractions.

Return Value

s1 == s2

Implementation

Folio call implemented in operamath V20.

Associated Files

operamath.h, operamath.lib

See Also

`CompareS64()`, `CompareU64()`, `CompareUF32()`, `CompareUF60()`, `CompareSF60()`

CompareU64

Compares two unsigned 64-bit integer quantities.

Synopsis

```
int32 CompareU64( uint64 *s1, uint64 *s2 )
```

Description

This function compares two unsigned 64-bit integers. This function is actually the same function as `CompareUF32()` and `CompareUF60()`.

Arguments

s1, s2 Pointers to unsigned 64-bit integers.

Return Value

The function returns 1 if s1 > s2, zero if s1 == s2, or -1 if s1 < s2.

Implementation

Folio call implemented in operamath V20.

Associated Files

operamath.h, operamath.lib

See Also

`CompareSF32()`, `CompareS64()`, `CompareUF32()`, `CompareSF60()`, `CompareUF60()`

CompareUF32

Compares two unsigned 32.32-format fractions.

Synopsis

```
int32 CompareUF32( ufrac32 *s1, ufrac32 *s2 )
```

Description

This function compares two unsigned 32.32-format fractions. This function is actually the same function as `CompareU64()` and `CompareUF60()`.

Arguments

s1, s2 Pointers to unsigned 32.32 fractions.

Return Value

The function returns 1 if s1 > s2, zero if s1 == s2, or -1 if s1 < s2.

Implementation

Folio call implemented in operamath V20.

Associated Files

operamath.h, operamath.lib

See Also

`CompareS64()`, `CompareSF32()`, `CompareU64()`, `CompareUF60()`, `CompareSF60()`

CompareUF60

Compares two unsigned 4.60-format fractions.

Synopsis

```
int32 CompareUF60( ufrac60 *s1, ufrac60 *s2 )
```

Description

This function compares two unsigned 4.60-format fractions. This function is actually the same function as `CompareU64()` and `CompareUF32()`.

Arguments

s1, s2 Pointers to unsigned 4.60 fractions.

Return Value

The function returns 1 if s1 > s2, zero if s1 == s2, or -1 if s1 < s2.

Implementation

Folio call implemented in operamath V20.

Associated Files

operamath.h, operamath.lib

See Also

`CompareS64()`, `CompareSF32()`, `CompareU64()`, `CompareUF32()`, `CompareSF60()`

CompareSF60

Compares two signed 4.60-format fractions.

Synopsis

```
int32 CompareSF60( frac60 *s1, frac60 *s2 )
```

Description

This function compares two signed 4.60-format fractions. This function is actually the same function as `CompareS64()` and `CompareSF32()`.

Arguments

s1, s2 Pointers to signed 4.60 fractions.

Return Value

The function returns 1 if s1 > s2, zero if s1 == s2, or -1 if s1 < s2.

Implementation

Folio call implemented in operamath V20.

Associated Files

operamath.h, operamath.lib

See Also

`CompareS64()`, `CompareU64()`, `CompareUF32()`, `CompareUF60()`, `CompareSF32()`

Convert32_F16

Converts a 32-bit integer to a 16.16 fraction.

Synopsis

```
frac16 Convert32_F16( int32 x )
```

Description

This macro converts a 32-bit integer into a 16.16 fraction.

Arguments

x
: The 32-bit integer to be converted.

Return Value

The function returns the 16.16 result of the conversion. The fractional 1- bits of the result are zero, and the upper 16-bits of the argument are lost.

Implementation

Macro implemented in operamath.h V20.

Associated Files

operamath.h

See Also

ConvertF14_F16(), ConvertF16_F14(), ConvertF16_F30(), ConvertF30_F16(), Convert32_F32(), ConvertF16_32(), ConvertF32_F16(), ConvertS32_64(), ConvertSF16_F32(), ConvertU32_64(), ConvertUF16_F32()

ConvertF14_F16

Converts a 2.14 fraction to a 16.16 fraction.

Synopsis

```
frac16 ConvertF14_F16( frac14 x )
```

Description

This macro converts a 2.14 fraction to a 16.16 fraction.

Arguments

x
: The 2.14 fraction to be converted.

Return Value

The function returns the 16.16 fraction result of the conversion.

Implementation

Macro implemented in operamath.h V20.

Associated Files

operamath.h

See Also

ConvertF16_F14(), ConvertF16_F30(), ConvertF16_32(), ConvertF30_F16(), ConvertF32_F16(), Convert32_F32(), Convert32_F16(), ConvertS32_64(), ConvertSF16_F32(), ConvertU32_64(), ConvertUF16_F32()

ConvertF16_F14

Converts a 16.16 fraction to a 2.14 fraction.

Synopsis

```
frac14 ConvertF16_F14( frac16 x )
```

Description

This macro converts a 16.16 fraction to a 2.14 fraction, using the upper 16-bits of a 32-bit quantity.

Arguments

x
: The 16.16 fraction to be converted.

Return Value

The function returns the 2.14 fraction result of the conversion.

Implementation

Macro implemented in operamath.h V20.

Associated Files

operamath.h

See Also

ConvertF14_F16(), ConvertF16_F30(), ConvertF16_32(), ConvertF30_F16(), ConvertF32_F16(), Convert32_F32(), Convert32_F16(), ConvertS32_64(), ConvertSF16_F32(), ConvertU32_64(), ConvertUF16_F32()

ConvertF16_F30

Converts a 16.16 fraction to a 2.30 fraction.

Synopsis

```
frac30 ConvertF16_F30( frac16 x )
```

Description

This macro converts a 16.16 fraction to a 2.30 fraction.

Arguments

x
: The 16.16 fraction to be converted.

Return Value

The function returns the 2.30 fraction result of the conversion.

Implementation

Macro implemented in operamath.h V20.

Associated Files

operamath.h

See Also

ConvertF14_F16(), ConvertF16_F14(), ConvertF16_32(), ConvertF30_F16(), ConvertF32_F16(), Convert32_F32(), Convert32_F16(), ConvertS32_64(), ConvertSF16_F32(), ConvertU32_64(), ConvertUF16_F32()

ConvertF16_32

Converts a 16.16 fraction to a 32-bit integer.

Synopsis

```
int32 ConvertF16_32( frac16 x )
```

Description

This macro converts a 16.16 fraction to a 32-bit integer. The upper 16-bits are zero or are filled with the sign bit of the argument. The fraction bits of the argument are lost.

Arguments

x
: The 16.16 fraction to be converted.

Return Value

The function returns the 32-bit integer result of the conversion.

Implementation

Macro implemented in operamath.h V20.

Associated Files

operamath.h

See Also

ConvertF14_F16(), ConvertF16_F14(), ConvertF16_F30(), ConvertF30_F16(), ConvertF32_F16(), Convert32_F32(), Convert32_F16(), ConvertS32_64(), ConvertSF16_F32(), ConvertU32_64(), ConvertUF16_F32()

ConvertF30_F16

Converts a 2.30 fraction to a 16.16 fraction.

Synopsis

```
frac16 ConvertF30_F16( frac30 x )
```

Description

This macro converts a 2.30 fraction to a 16.16 fraction. The upper 14-bits will be filled with the sign bit of the argument.

Arguments

x
: A 2.30 fraction to be converted.

Return Value

The function returns the 16.16 fraction result of the conversion.

Implementation

Macro implemented in operamath.h V20.

Associated Files

operamath.h

See Also

ConvertF14_F16(), ConvertF16_F14(), ConvertF16_F30(), ConvertF16_32, ConvertF32_F16(), Convert32_F32(), Convert32_F16(), ConvertS32_64(), ConvertSF16_F32(), ConvertU32_64(), ConvertUF16_F32()

ConvertF32_F16

Converts a 32.32 fraction to a 16.16 fraction.

Synopsis

```
frac16 ConvertF32_F16( frac32 *x )
```

Description

This macro converts a 32.32 fraction to a 16.16 fraction. The result will be correct only if the integer part of the argument can be expressed in 16-bits. Only the most-significant 16-bits of the fractional part of the argument are preserved.

Arguments

x
: A pointer to a 32.32 fraction to be converted.

Return Value

The function returns the 16.16 fraction result of the conversion.

Implementation

Macro implemented in operamath.h V20.

Associated Files

operamath.h

See Also

ConvertF14_F16(), ConvertF16_F14(), ConvertF16_F30(), ConvertF30_F16(), ConvertF16_32(), Convert32_F32(), Convert32_F16(), ConvertS32_64(), ConvertSF16_F32(), ConvertU32_64(), ConvertUF16_F32()

Convert32_F32

Converts a 32-bit integer to a 32.32 fraction.

Synopsis

```
void Convert32_F32( frac32 *d, int32 x )
```

Description

This macro converts a 32-bit integer to a 32.32 fraction. The fractional portion of the result is zero. The 32.32 result of the conversion is deposited in the location pointed to by the dest argument.

Arguments

d
: A pointer to a 32.32 structure to store the result.

x
: A 32-bit integer to be converted.

Implementation

Macro implemented in operamath.h V20.

Associated Files

operamath.h

See Also

ConvertF14_F16(), ConvertF16_F14(), ConvertF16_F30(), ConvertF30_F16(), ConvertF32_F16(), ConvertF16_32(), ConvertF32_F16(), ConvertS32_64(), ConvertSF16_F32(), ConvertU32_64(), ConvertUF16_F32()

ConvertS32_64

Converts a 32-bit integer to a 64-bit integer.

Synopsis

```
void ConvertS32_64( int64 *d, int32 x )
```

Description

This macro converts a 32-bit integer to a 64-bit integer. The upper 32-bits of the result are the sign bit of the argument. The 64-bit integer result is deposited in the location pointed to by the d argument.

Arguments

d
: A pointer to a 64-bit integer structure to store the result.

x
: A 32-bit integer to be converted.

Implementation

Macro implemented in operamath.h V20.

Associated Files

operamath.h

See Also

ConvertF14_F16(), ConvertF16_F14(), ConvertF16_F30(), ConvertF30_F16(), ConvertF16_32(), Convert32_F32(), ConvertF32_F16(), Convert32_F16(), ConvertSF16_F32(), ConvertU32_64(), ConvertUF16_F32()

ConvertSF16_F32

Converts a 16.16 fraction to a 32.32 fraction.

Synopsis

```
void ConvertSF16_F32( frac32 *d, frac16 x )
```

Description

This macro converts a 16.16 fraction to a 32.32 fraction. The upper 16-bits of the result are the sign bit of the argument, the least 16-bits of the result are zero. The 32.32 fraction result is deposited in the location pointed to by the d argument.

Arguments

d
: A pointer to a 32.32 fraction structure to store the result.

x
: A 16.16 fraction to be converted.

Implementation

Macro implemented in operamath.h V20.

Associated Files

operamath.h

See Also

ConvertF14_F16(), ConvertF16_F14(), ConvertF16_F30(), ConvertF30_F16(), ConvertF16_32(), Convert32_F32(), ConvertF32_F16(), ConvertS32_64(), Convert32_F16(), ConvertU32_64(), ConvertUF16_F32()

ConvertU32_64

Converts an unsigned 32-bit integer to an unsigned 64-bit integer.

Synopsis

```
void ConvertU32_64( uint64 *d, uint32 x )
```

Description

This macro converts an unsigned 32-bit integer to an unsigned 64-bit integer. The upper 32-bits of the result are zero. The 64-bit unsigned integer result is deposited in the location pointed to by the d argument.

Arguments

d
: A pointer to a unsigned 64-bit integer structure to store the result.

x
: The unsigned 32-bit integer to be converted.

Implementation

Macro implemented in operamath.h V20.

Associated Files

operamath.h

See Also

ConvertF14_F16(), ConvertF16_F14(), ConvertF16_F30(), ConvertF30_F16(), ConvertF16_32(), Convert32_F32(), ConvertF32_F16(), Convert32_F16(), ConvertSF16_F32(), ConvertUF16_F32()

ConvertUF16_F32

Converts an unsigned 16.16 fraction to an unsigned 32.32fraction.

Synopsis

```
void ConvertUF16_F32( frac32 *d, ufrac16 x )
```

Description

This macro converts an unsigned 16.16 fraction to a 32.32 fraction. The upper 16-bits of the result and the least 16-bits of the result are zero. The 32.32 fraction result is deposited in the location pointed to by the d argument.

Arguments

d
: A pointer to a 32.32 fraction to store the result.

x
: An unsigned 16.16 fraction to be converted.

Implementation

Macro implemented in operamath.h V20.

Associated Files

operamath.h

See Also

ConvertF14_F16(), ConvertF16_F14(), ConvertF16_F30(), ConvertF30_F16(), ConvertF16_32(), Convert32_F32(), ConvertF32_F16(), ConvertS32_64(), Convert32_F16(), ConvertU32_64()

CosF16

Computes the 16.16 operamath cosine of a 16.16 fraction angle.

Synopsis

```
frac16 CosF16( frac16 x )
```

Description

This function returns the 16.16 operamath cosine of a 16.16 fraction angle. In operamath coordinates, there are 256.0 units in a circle.

Arguments

x
: A 16.16 fraction describing the angle of the circle.

Return Value

The value returned is the cosine of the input angle.

Implementation

Folio call implemented in operamath V20.

Associated Files

operamath.h, operamath.lib

See Also

SinF16(), CosF32(), SinF32(), CosF30(), SinF30()

SinF16

Computes the 16.16 sine of a 16.16 fraction angle.

Synopsis

```
frac16 SinF16( frac16 x )
```

Description

This function returns the 16.16 sine of a 16.16 fraction angle. In operamath coordinates, there are 256.0 units in a circle.

Arguments

x
: A 16.16 fraction describing the angle of the circle.

Return Value

The value returned is the sine of the input angle.

Implementation

Folio call implemented in operamath V20.

Associated Files

operamath.h, operamath.lib

See Also

CosF16(), CosF32(), SinF32(), CosF30(), SinF30()

CosF32

Computes the operamath 32.32 cosine of a 16.16 fraction.

Synopsis

```
frac32 CosF32( frac32 *c, frac16 x )
```

Description

This macro returns the operamath 32.32 cosine of a 16.16 fraction. In operamath coordinates, there are 256.0 units in a circle.

The macro actually calls the `SinF32()` function with one quarter circle added to the input value.

Arguments

c
: A pointer to a 32.32 fraction containing the result.

x
: A 16.16 fraction describing the angle of the circle.

Return Value

The macro returns the cosine of the input angle.

Implementation

Macro implemented in operamath.h V20.

Associated Files

operamath.h

See Also

SinF16(), CosF16(), SinF32(), CosF30(), SinF30()

SinF32

Computes the operamath 32.32 sine of a 16.16 fraction.

Synopsis

```
void SinF32( frac32 *dest, frac16 x )
```

Description

This function returns the operamath 32.32 sine of a 16.16 fraction. In operamath coordinates, there are 256.0 units in a circle.

Arguments

dest
: A pointer to a 32.32 fraction containing the result.

x
: A 16.16 fraction describing the angle of the circle.

Implementation

Folio call implemented in operamath V20.

Associated Files

operamath.h, operamath.lib

See Also

SinF16(), CosF16(), CosF32(), CosF30(), SinF30()

CosF30

Computes the operamath 2.30 cosine of a 16.16 fraction.

Synopsis

```
frac30 CosF30( frac16 x )
```

Description

This function returns the operamath 2.30 cosine of a 16.16 fraction. In operamath coordinates, there are 256.0 units in a circle.

Arguments

x
: A 16.16 fraction describing the angle of the circle.

Return Value

The function returns the 2.30 cosine of the input angle.

Implementation

Folio call implemented in operamath V20.

Associated Files

operamath.h, operamath.lib

See Also

SinF16(), CosF16(), SinF32(), CosF32(), SinF30()

SinF30

Computes the operamath 2.30 sine of a 16.16 fraction.

Synopsis

```
frac30 SinF30( frac16 x )
```

Description

This function returns the operamath 2.30 sine of a 16.16 fraction. In operamath coordinates, there are 256.0 units in a circle.

Arguments

x
: A 16.16 fraction describing the angle of the circle.

Return Value

The function returns the 2.30 sine of the input angle.

Implementation

Folio call implemented in operamath V20.

Associated Files

operamath.h, operamath.lib

See Also

SinF16(), CosF16(), SinF32(), CosF32(), CosF30()

Cross3_F16

Computes the cross-product of two vectors of 16.16 values.

Synopsis

```
void Cross3_F16( vec3f16 *dest, vec3f16 v1, vec3f16 v2 )
```

Description

This function multiplies two 3-coordinate vectors of 16.16 values together and deposits the cross-product in the location pointed to by dest.

Arguments

dest
: A pointer to a destination vector to store the resulting cross product.

v1, v2
: 3-coordinate vector multiplicands.

Implementation

SWI implemented in operamath V20.

Associated Files

operamath.h

DivRemS32

Computes the quotient and remainder of a 32-bit division.

Synopsis

```
int32 DivRemS32( int32 *rem, int32 d1, int32 d2 )
```

Description

This function divides one 32-bit integer by another and returns the quotient and remainder.

This function calls the standard C function for division. If you need only the quotient or only the remainder, use the standard C notation for division instead.

Arguments

rem
: A pointer to a 32-bit integer to store the remainder.

d1
: The dividend.

d2
: The divisor.

Return Value

The function returns the quotient of the division. The remainder is deposited in the location pointed to by the rem argument.

Implementation

Folio call implemented in operamath V20.

Associated Files

operamath.h, operamath.lib

Caveats

The function does not detect overflows.

See Also

DivRemU32(), DivS64(), DivU64(), DivSF16(), DivUF16(), DivRemUF16(), DivRemSF16()

DivRemU32

Computes the quotient and remainder of a 32-bit division.

Synopsis

```
uint32 DivRemU32( uint32 *rem, uint32 d1, uint32 d2 )
```

Description

This function divides one unsigned 32-bit integer by another and returns the quotient and remainder.

This function calls the standard C function for division. If only the quotient or only the remainder is desired, the standard C notation for divide should be used.

Arguments

rem
: A pointer to a unsigned 32-bit integer to store the remainder.

d1
: The dividend.

d2
: The divisor.

Return Value

The function returns the quotient of the division. The remainder is deposited in the location pointed to by the rem argument.

Implementation

Folio call implemented in operamath V20.

Associated Files

operamath.h, operamath.lib

Caveats

The function does not detect overflows.

See Also

DivRemSF16(), DivRemS32(), DivU64(), DivSF16(), DivUF16(), DivRemUF16(), DivS64()

DivRemSF16

Computes the quotient and remainder of a 16.16 division.

Synopsis

```
frac16 DivRemSF16( frac16 *rem, frac16 d1, frac16 d2 )
```

Description

This function divides one signed 16.16 fraction by another and returns the quotient and remainder. This function returns a correct result if the arguments are int32 or frac30, as long as both d1 and d2 are the same type.

The remainder is not a 16.16 fraction; instead, it is a signed fraction in 0.32 format. The most-significant bit of the remainder is a sign bit, and this must be extended if the remainder is to be used in subsequent calculations. An overflow value is denoted by maximum positive return in both values.

Arguments

rem
: A pointer to the remainder of the division, a signed fraction in 0.32 format.

d1
: The dividend.

d2
: The divisor.

Return Value

The function returns the quotient of the division. The remainder is deposited in the location pointed to by the rem argument.

Implementation

Folio call implemented in operamath V20.

Associated Files

operamath.h, operamath.lib

See Also

DivRemU32(), DivRemS32(), DivU64(), DivUF16(), DivSF16(), DivRemUF16(), DivS64()

DivS64

Computes the quotient and remainder of a 64-bit division.

Synopsis

```
int64 *DivS64( int64 *q, int64 *r, int64 *d1, int64 *d2 )
```

Description

This function divides one 64-bit integer by another and returns the quotient and remainder.

Arguments

q
: A pointer to a 64-bit integer to store the quotient.

r
: A pointer to a 64-bit integer to store the remainder.

d1
: A pointer to a dividend.

d2
: A pointer to a divisor.

Return Value

The function returns a pointer to q.

Implementation

Folio call implemented in operamath V20.

Associated Files

operamath.h, operamath.lib

Caveats

The function does not detect overflows.

See Also

DivRemU32(), DivRemS32(), DivU64(), DivSF16(), DivUF16(), DivRemUF16(), DivRemSF16()

DivU64

Computes the quotient and remainder of a 64-bit division.

Synopsis

```
int64 *DivU64( uint64 *q, uint64 *r, uint64 *d1, uint64 *d2 )
```

Description

This function divides one unsigned 64-bit integer by another and returns the quotient and remainder.

Arguments

q
: A pointer to a 64-bit integer to store the quotient.

r
: A pointer to a 64-bit integer to store the remainder.

d1
: A pointer to the dividend.

d2
: A pointer to the divisor.

Return Value

The function returns a pointer to q. The quotient is deposited in the location pointed to by q. The remainder is deposited in the location pointed to by r.

Implementation

Folio call implemented in operamath V20.

Associated Files

operamath.h, operamath.lib

Caveats

The function does not detect overflows.

See Also

DivRemSF16(), DivRemS32(), DivRemU32(), DivSF16(), DivUF16(), DivRemUF16(), DivS64()

DivSF16

Computes the quotient of a 16.16 division.

Synopsis

```
frac16 DivSF16( frac16 d1, frac16 d2 )
```

Description

This function divides one 16.16 fraction by another and returns the quotient.

This function also returns a correct result if the arguments are int32 or frac30, as long as both d1 and d2 are the same type.

Arguments

d1
: The dividend.

d2
: The divisor.

Return Value

The function returns the quotient of the division.

Implementation

Folio call implemented in operamath V20.

Associated Files

operamath.h, operamath.lib

Caveats

The function does not detect overflows.

See Also

DivRemU32(), DivRemS32(), DivU64(), DivRemSF16(), DivUF16(), DivRemUF16(), DivS64()

DivUF16

Computes the quotient of a 16.16 division.

Synopsis

```
ufrac16 DivUF16( ufrac16 d1, ufrac16 d2 )
```

Description

This function divides one unsigned 16.16 fraction by another and returns the quotient. This function returns a correct 16.16 result if the arguments are uint32 or ufrac30, as long as both arguments, d1 and d2, are the same type.

Arguments

d1
: The dividend.

d2
: The divisor.

Return Value

The function returns the quotient of the division.

Implementation

Folio call implemented in operamath V20.

Associated Files

operamath.h, operamath.lib

Caveats

The function does not detect overflows.

See Also

DivRemSF16(), DivRemS32(), DivRemU32(), DivSF16(), DivRemUF16(), DivU64(), DivS64()

DivRemUF16

Computes the quotient and remainder of a 16.16 division.

Synopsis

```
ufrac16 DivRemUF16( ufrac16 *rem, ufrac16 d1, ufrac16 d2 )
```

Description

This function divides one unsigned 16.16 fraction by another and returns the quotient and remainder. This function returns a correct 16.16 result if the arguments are uint32 or ufrac30, as long as both arguments, d1 and d2, are the same type.

The remainder is not a 16.16 fraction; instead, it is an unsigned fraction in 0.32 format. An overflow condition is signaled by maximum return in both values.

Arguments

rem
: A pointer to a unsigned fraction in 0.32 format to store the remainder.

d1
: The dividend.

d2
: The divisor.

Return Value

The function returns the quotient of the division. The remainder is deposited in the location pointed to by the rem argument.

Implementation

Folio call implemented in operamath V20.

Associated Files

operamath.h, operamath.lib

Caveats

The function does not detect overflows.

See Also

DivRemSF16(), DivRemS32(), DivRemU32(), DivSF16(), DivUF16(), DivU64(), DivS64()

Dot3_F16

Multiplies two vectors of 16.16 values.

Synopsis

```
frac16 Dot3_F16( vec3f16 v1, vec3f16 v2 )
```

Description

This function multiplies two 3-coordinate vectors of 16.16 values together and returns the dot product.

Arguments

v1, v2
: 3-coordinate vector multiplicands.

Return Value

The function returns the dot product of the two vectors.

Implementation

SWI implemented in operamath V20.

Associated Files

operamath.h

See Also

Dot4_F16()

Dot4_F16

Multiplies two vectors of 16.16 values.

Synopsis

```
frac16 Dot4_F16( vec4f16 v1, vec4f16 v2 )
```

Description

This function multiplies two 4-coordinate vectors of 16.16 values together and returns the dot product.

Arguments

v1, v2 4-coordinate vector multiplicands.

Return Value

The function returns the dot product of the two vectors.

Implementation

SWI implemented in operamath V20.

Associated Files

operamath.h

See Also

Dot3_F16()

Mul32

Multiplies two 32-bit integers.

Synopsis

```
int32 Mul32( int32 x, int32 y )
```

Description

This macro multiplies two 32-bit integers together and returns the product. The macro is only included for completeness.

Arguments

x, y The multiplicands.

Return Value

The function returns the product of the two arguments.

Implementation

Macro implemented in operamath.h V20.

Associated Files

operamath.h

See Also

Mul64(), MulS32_64(), MulU32_64(), MulF14(), MulSF16(), MulSF30(), MulUF16(), MulSF16_F32(), MulUF16_F32(), MulSF30_F60(), MulUF30_F60()

Mul64

Multiplies two 64-bit integers.

Synopsis

```
void Mul64( int64 *p, int64 *m1, int64 *m2 )
```

Description

This function multiplies two 64-bit integers together and returns the product. An overflow condition is not detected. The 64-bit integer result is deposited in the location pointed to by the p argument.

Arguments

p
: A pointer to a the location to store the 64-bit integer result.

m1, m2
: Pointers to the locations of the two 64-bit integer multiplicands.

Implementation

Folio call implemented in operamath V20.

Associated Files

operamath.h, operamath.lib

Caveats

The function does not detect overflows.

See Also

Mul32(), MulS32_64(), MulU32_64(), MulF14(), MulSF16(), MulSF30(), MulUF16(), MulSF16_F32(), MulUF16_F32(), MulSF30_F60(), MulUF30_F60()

MulS32_64

Multiplies two 32-bit integers and returns a 64-bit result.

Synopsis

```
void MulS32_64( int64 *prod, int32 m1, int32 m2 )
```

Description

This function multiplies two signed 32-bit integers together and deposits the 64-bit product in the location pointed to by the prod argument.

Arguments

prod
: A pointer to the location to store the 64-bit result.

m1, m2
: Two 32-bit multiplicands.

Implementation

Folio call implemented in operamath V20.

Associated Files

operamath.h, operamath.lib

See Also

Mul32(), Mul64(), MulSF16(), MulU32_64(), MulF14(), MulSF30(), MulUF16(), MulSF16_F32(), MulUF16_F32(), MulSF30_F60(), MulUF30_F60()

MulSF16

Multiplies two signed 16.16 fractions.

Synopsis

```
frac16 MulSF16( frac16 m1, frac16 m2 )
```

Description

This function multiplies two signed 16.16 fractions together and returns the 16.16 resulting product. The lower bits of the result are truncated.

Arguments

m1, m2 The multiplicands.

Return Value

The function returns the product of the two arguments.

Implementation

Folio call implemented in operamath V20.

Associated Files

operamath.h, operamath.lib

Caveats

The function does not detect overflows.

See Also

Mul32(), Mul64(), MulS32_64(), MulU32_64(), MulF14(), MulSF30(), MulUF16(), MulSF16_F32(), MulUF16_F32(), MulSF30_F60(), MulUF30_F60()

MulU32_64

Multiplies two unsigned 32-bit integers and returns a 64-bit result.

Synopsis

```
void MulU32_64( uint64 *prod, uint32 m1, uint32 m2 )
```

Description

This function multiplies two unsigned 32-bit integers together and deposits the unsigned 64-bit result in the location pointed to by the prod argument.

Arguments

prod
: A pointer to the location to store the 64-bit result.

m1, m2
: Two unsigned 32-bit multiplicands.

Implementation

Folio call implemented in operamath V20.

Associated Files

operamath.h, operamath.lib

See Also

Mul32(), Mul64(), MulSF16(), MulS32_64(), MulF14(), MulSF30(), MulUF16(), MulSF16_F32(), MulUF16_F32(), MulSF30_F60(), MulUF30_F60()

MulF14

Multiplies two 2.14 fractions.

Synopsis

```
frac14 MulF14( frac14 x, frac14 y )
```

Description

This macro multiplies two 2.14 fractions together and returns the product.

Arguments

x, y The multiplicands.

Return Value

The function returns the product of the two arguments.

Implementation

Macro implemented in operamath.h V20.

Associated Files

operamath.h

Caveats

The function does not detect overflows.

See Also

Mul32(), Mul64(), MulS32_64(), MulU32_64(), MulSF16(), MulUF16(), MulSF30(), MulSF16_F32(), MulUF16_F32(), MulSF30_F60(), MulUF30_F60()

MulUF16

Multiplies two unsigned 16.16 fractions.

Synopsis

```
ufrac16 MulUF16( ufrac16 m1, ufrac16 m2 )
```

Description

This function multiplies two unsigned 16.16 fractions together and returns the 16.16 resulting product. The lower bits of the result are truncated.

Arguments

m1, m2 The multiplicands.

Return Value

The function returns the product of the two arguments.

Implementation

Folio call implemented in operamath V20.

Associated Files

operamath.h, operamath.lib

Caveats

The function does not detect overflows.

See Also

Mul32(), Mul64(), MulS32_64(), MulU32_64(), MulF14(), MulSF30(), MulSF16(), MulSF16_F32(), MulUF16_F32(), MulSF30_F60(), MulUF30_F60()

MulSF30

Multiplies two 2.30 fractions.

Synopsis

```
frac30 MulSF30( frac30 m1, frac30 m2 )
```

Description

This function multiplies two 2.30 fractions together and returns the 2.30 resulting product. The lower bits of the result are truncated.

Arguments

m1, m2 The multiplicands.

Return Value

The function returns the product of the two arguments.

Implementation

Folio call implemented in operamath V20.

Associated Files

operamath.h, operamath.lib

Caveats

The function does not detect overflows.

See Also

Mul32(), Mul64(), MulS32_64(), MulU32_64(), MulF14(), MulSF16(), MulUF16(), MulSF16_F32(), MulUF16_F32(), MulSF30_F60(), MulUF30_F60()

MulSF16_F32

Multiplies two signed 16.16 numbers and returns a 32.32 result.

Synopsis

```
void MulSF16_F32( frac32 *prod, frac16 m1, frac16 m2 )
```

Description

This function multiplies two signed 16.16 numbers together and deposits the 32.32 result in the location pointed to by the prod argument.

Arguments

prod
: A pointer to the location to store the 32-bit result.

m1, m2
: Two 16.16 multiplicands.

Implementation

Folio call implemented in operamath V20.

Associated Files

operamath.h, operamath.lib

See Also

Mul32(), Mul64(), MulSF16(), MulU32_64(), MulF14(), MulSF30(), MulUF16(), MulS32_64(), MulUF16_F32(), MulSF30_F60(), MulUF30_F60()

MulUF16_F32

Multiplies two unsigned 16.16 numbers and returns a 32.32 result.

Synopsis

```
void MulUF16_F32( ufrac32 *prod, ufrac16 m1, ufrac16 m2 )
```

Description

This function multiplies two unsigned 16.16 numbers together and deposits the unsigned 32.32 result in the location pointed to by the prod argument.

Arguments

prod
: A pointer to the location to store the result.

m1, m2
: Two unsigned 16.16 multiplicands.

Implementation

Folio call implemented in operamath V20.

Associated Files

operamath.h, operamath.lib

See Also

Mul32(), Mul64(), MulSF16(), MulU32_64(), MulF14(), MulSF30(), MulUF16(), MulS32_64(), MulSF16_F32(), MulSF30_F60(), MulUF30_F60()

MulSF30_F60

Multiplies two signed 2.30 numbers and returns a 4.60 result.

Synopsis

```
void MulSF30_F60( frac60 *prod, frac30 m1, frac30 m2 )
```

Description

This function multiplies two signed 2.30 numbers together and deposits the 4.60 result in the location pointed to by the prod argument.

Arguments

prod
: A pointer to the location to store the 4.60 result.

m1, m2
: Two 2.30 multiplicands.

Implementation

Folio call implemented in operamath V20.

Associated Files

operamath.h, operamath.lib

See Also

Mul32(), Mul64(), MulSF16(), MulU32_64(), MulF14(), MulSF30(), MulUF16(), MulS32_64(), MulUF16_F32(), MulSF16_F32(), MulUF30_F60()

MulUF30_F60

Multiplies two unsigned 2.30 numbers and returns a 4.60 result.

Synopsis

```
void MulUF30_F60( ufrac60 *prod, ufrac30 m1, ufrac30 m2 )
```

Description

This function multiplies two unsigned 2.30 numbers together and deposits the unsigned 4.60 result in the location pointed to by the prod argument.

Arguments

prod
: A pointer to the location to store the unsigned 4.60 result.

m1, m2
: Two unsigned 2.30 multiplicands.

Implementation

Folio call implemented in operamath V20.

Associated Files

operamath.h, operamath.lib

See Also

Mul32(), Mul64(), MulSF16(), MulU32_64(), MulF14(), MulSF30(), MulUF16(), MulS32_64(), MulUF16_F32(), MulSF16_F32(), MulSF30_F60()

MulManyF16

Multiplies an array of 16.16 values by another array of 16.16 values.

Synopsis

```
void MulManyF16( frac16 *dest, frac16 *src1, frac16 *src2,int32 count
)
```

Description

This function multiplies an array of 16.16 fractions by another array of 16.16 fractions. Every element of the first array is multiplied by the corresponding element in the other array, and the results are deposited in the destination array.

Arguments

dest
: A pointer to the destination array to store the results.

src1
: Pointer to source array of 16.16 fractions to be multiplied.

src2
: Pointer to source array of 16.16 fractions to be multiplied.

count
: Number of vectors for the multiplication.

Implementation

SWI implemented in operamath V20.

Associated Files

operamath.h

See Also

MulScalarF16()

MulScalarF16

Multiplies a 16.16 scalar by an array of 16.16 values.

Synopsis

```
void MulScalarF16( frac16 *dest, frac16 *src, frac16 scalar,int32
count )
```

Description

This function multiplies a 16.16 scalar by an array of 16.16 fractions. Every element of the array of frac16 values is multiplied by the same value, and the results are deposited in the destination array.

Arguments

dest
: A pointer to the destination array to store the results.

src
: A pointer to the source array of 16.16 fractions to be multiplied.

scalar
: A 16.16 scalar.

count
: The number of elements in the vector.

Implementation

SWI implemented in operamath V20.

Associated Files

operamath.h

See Also

MulManyF16()

MulManyVec3Mat33_F16

Multiplies one or more vectors by a 3x3 matrix of 16.16 values.

Synopsis

```
void MulManyVec3Mat33_F16( vec3f16 *dest, vec3f16 *src,mat33f16 mat,
int32 count )
```

Description

This function multiplies an array of one or more vectors by a 3x3 matrix of 16.16 fractions. The results of the products are deposited in the array of vectors pointed to by the dest argument.

Arguments

dest
: A pointer to the array of destination vectors to store the results.

src
: A pointer to the array of source vectors to be multiplied with the matrix.

mat
: A 3x3 matrix of 16.16 fractions.

count
: The number of vectors for the multiplication.

Implementation

SWI implemented in operamath V20.

Associated Files

operamath.h

See Also

MulManyVec4Mat44_F16()

MulManyVec4Mat44_F16

Multiplies one or more vectors by a 4x4 matrix of 16.16 values.

Synopsis

```
void MulManyVec4Mat44_F16( vec4f16 *dest, vec4f16 *src,mat44f16 mat,
int32 count )
```

Description

This function multiplies an array of one or more vectors by a 3x3 matrix of 16.16 fractions. The results of the products are deposited in the array of vectors pointed to by the dest argument.

Arguments

dest
: A pointer to the array of destination vectors to store the results.

src
: A pointer to the array of source vectors to be multiplied with the matrix.

mat
: A 4x4 matrix of 16.16 fractions.

count
: The number of vectors for the multiplication.

Implementation

SWI implemented in operamath V20.

Associated Files

operamath.h

See Also

MulManyVec3Mat33_F16()

MulManyVec3Mat33DivZ_F16

Multiplies a 3x3 matrix of 16.16 values by 1 or more 3-coordinate vectors of 16.16 values, then multiplies x and y elements of the result vector by ratio of n/z.

Synopsis

```
void MulManyVec3Mat33DivZ_F16( mmv3m33d *s )
```

Description

This function multiplies a 3x3 matrix of 16.16 fractions by one or more 3-coordinate vectors of 16.16 values. It then multiplies the x and y elements of the result vector {x, y, z} by the ratio of n/z, and deposits the result vector {x*n/z, y*n/z, z} in the product vector dest. This function can be used to perform the final projection transformation on points just before rendering.

This function uses the structure mmv3m33d. Here is its definition:

// structure to pass arguments to MulManyVec3Mat33DivZ_F16 function typedef struct mmv3m3dd{vec3f16*dest; vec3f16*src;mat33f16*mat; frac16n; uint32count;} mmv3m33d;

Arguments

s->dest
: A pointer to the destination vector to store results.

s->vec
: A pointer to the vector to multiply.

s->mat
: A pointer to the 3 X 3 matrix of 16.16 fractions to multiply.

s->n
: Scale factor to be multiplied by x and y projected results.

s->count

Number of vectors for the multiplication.

Implementation

SWI implemented in operamath V20.

Associated Files

operamath.h

See Also

`MulManyVec3Mat33_F16()`

MulMat33Mat33_F16

Computes the product of two 3x3 matrices of 16.16values.

Synopsis

```
void MulMat33Mat33_F16( mat33f16 dest, mat33f16 src1,mat33f16 src2 )
```

Description

This function multiplies two 3x3 matrices of 16.16 fractions together. The results of the product are deposited in the location for the matrix dest. Note that incorrect results may occur if the destination matrix is the same matrix as one of the source matrices.

Arguments

dest
: A pointer to the destination matrix to store the results.

src1, src2
: The source matrices to be multiplied.

Implementation

SWI implemented in operamath V20.

Associated Files

operamath.h

See Also

MulVec3Mat33_F16(), MulVec4Mat44_F16(), MulMat44Mat44_F16()

MulVec3Mat33_F16

Computes the product of a 3x3 matrix and a vector.

Synopsis

```
void MulVec3Mat33_F16( vec3f16 dest, vec3f16 vec, mat33f16mat )
```

Description

This function multiplies a 3x3 matrix of 16.16 fractions by a 3-coordinate vector of 16.16 values and deposits the results of the product in the vector dest.

Arguments

dest
: The destination vector to store the results.

vec
: The vector to multiply.

mat
: The matrix to multiply.

Implementation

SWI implemented in operamath V20.

Associated Files

operamath.h

See Also

MulVec4Mat44_F16(), MulMat33Mat33_F16(), MulMat44Mat44_F16()

MulVec4Mat44_F16

Computes the product of a 4x4 matrix and a vector.

Synopsis

```
void MulVec4Mat44_F16( vec4f16 dest, vec4f16 vec, mat44f16mat )
```

Description

This function multiplies a 4x4 matrix of 16.16 fractions by a 4-coordinate vector of 16.16 values and deposits the results of the product in the vector dest.

Arguments

dest
: The destination vector to store the results.

vec
: The vector to multiply.

mat
: The matrix to multiply.

Implementation

SWI implemented in operamath V20.

Associated Files

operamath.h

See Also

MulVec3Mat33_F16(), MulMat33Mat33_F16(), MulMat44Mat44_F16()

MulMat44Mat44_F16

Computes the product of two 4x4 matrices of 16.16 values.

Synopsis

```
void MulMat44Mat44_F16( mat44f16 dest, mat44f16 src1,mat44f16 src2 )
```

Description

This function multiplies two 4x4 matrices of 16.16 fractions together. The results of the product are deposited in the matrix dest. Note that incorrect results may occur if the destination matrix is the same matrix as one of the source matrices.

Arguments

dest
: The destination matrix to store the results.

src1, src2
: The source matrices to be multiplied together.

Implementation

SWI implemented in operamath V20.

Associated Files

operamath.h

See Also

MulVec3Mat33_F16(), MulVec4Mat44_F16(), MulMat33Mat33_F16()

MulObjectMat33_F16

Multiplies a matrix within an object structure by an external 3x3 matrix of 16.16 values.

Synopsis

```
void MulObjectMat33_F16( void *objectlist[],ObjOffset2 *offsetstruct,
mat33f16 mat, int32 count )
```

Description

This function multiplies a matrix within object structures by an external 3x3 matrix of 16.16 fractions, and repeats over a number of objects. The results of the product are deposited in the destination matrix in each object structure pointed to by the objectlist array.

The object structure argument defines offsets within objects to the elements to be manipulated. The definition for ObjOffset2 is as follows:

typedef struct ObjOffset2 {int32 oo2_DestMatOffset; int32 oo2_SrcMatOffset;} ObjOffset2;

- oo2_DestMatOffset is the offset (in bytes) from the beginning of an object structure to where to write the result of the matrix to multiply.
- oo2_SrcMatOffset is the offset (in bytes) from the beginning of an object structure to the location of the matrix to be multiplied.

Arguments

objectlist
: An array of pointers to object structures to modify.

offsetstruct
: A pointer to the source object structure that defines offsets within object's to the elements to be multiplied.

mat
: A 3x3 matrix of 16.16 fractions to be multiplied.

count
: The number of vectors for the multiplication.

Implementation

SWI implemented in operamath V20.

Associated Files

operamath.h

See Also

MulObjectMat44_F16()

MulObjectMat44_F16

Multiplies a matrix within an object structure by an external 4x4 matrix of 16.16 values.

Synopsis

```
void MulObjectMat44_F16( void *objectlist[],ObjOffset2 *offsetstruct,
mat44f16 mat, int32 count )
```

Description

This function multiplies a matrix within object structures by an external 4x4 matrix of 16.16 fractions, and repeats over a number of objects. The results of the product are deposited in the destination matrix in each object structure pointed to by the objectlist array.

The object structure argument defines offsets within objects to the elements to be manipulated. The definition for ObjOffset2 is as follows:

```
typedef struct ObjOffset2 {int32 oo2_DestMatOffset; int32
oo2_SrcMatOffset;} ObjOffset2;
```

- oo2_DestMatOffset is the offset (in bytes) from the beginning of an object structure to where to write the result of the matrix multiply.
- oo2_SrcMatOffset is the offset (in bytes) from the beginning of an object structure to the location of the matrix to be multiplied.

Arguments

objectlist
: An array of pointers to object structures to modify.

offsetstruct
: A pointer to the source object structure that defines offsets within object's to the elements to be multiplied.

mat
: A 4x4 matrix of 16.16 fractions to be multiplied.

count

The number of vectors for the multiplication.

Implementation

SWI implemented in operamath V20.

Associated Files

operamath.h

See Also

`MulObjectMat33_F16()`

MulObjectVec3Mat33_F16

Multiplies many vectors within an object structure by a 3x3 matrix of 16.16 values.

Synopsis

```
void MulObjectVec3Mat33_F16( void *objectlist[],ObjOffset1
*offsetstruct, int32 count )
```

Description

This function multiplies one or more vectors within object structures by a 3x3 matrix of 16.16 fractions also within that structure, and repeats over a number of objects. The results of the product are deposited in the destination arrays pointed to by object structures pointed to by the objectlist array.

The object structure argument defines offsets within objects to the elements to be manipulated. The object structure has a pointer to an array of points associated with the object, a pointer to an array of transformed points, a count of the number of points, and an orientation matrix. The definition for ObjOffset1 is as follows:

typedef struct ObjOffset1 {int32 oo1_DestArrayPtrOffset; int32 oo1_SrcArrayPtrOffset; int32 oo1_MatOffset; int32 oo1_CountOffset;} ObjOffset1;

- oo1_DestArrayPtrOffset is the offset (in bytes) from the beginning of an object structure to the location of the pointer to an array of vectors to fill with the results of the multiplies.
- oo1_SrcArrayPtrOffset is the offset (in bytes) from the beginning of an object structure to the location of the pointer to an array of vectors to be multiplied.
- oo1_MatOffset is the offset (in bytes) from the beginning of an object structure to the location of the matrix to be multiplied.
- oo1_CountOffset is the offset (in bytes) from the beginning of an object structure to the location of the count of the number of vectors to be multiplied.

Arguments

objectlist
: An array of pointers to object structures to modify.

offsetstruct
: A pointer to the source object structure that defines offsets within object's to the elements to be multiplied.

count
: The number of vectors for the multiplication.

Implementation

SWI implemented in operamath V20.

Associated Files

operamath.h

See Also

`MulObjectVec4Mat44_F16()`

MulObjectVec4Mat44_F16

Multiplies many vectors within an object structure by a 4x4 matrix of 16.16 values.

Synopsis

```
void MulObjectVec4Mat44_F16( void *objectlist[],ObjOffset1
*offsetstruct, int32 count )
```

Description

This function multiplies one or more vectors within object structures by a 3x3 matrix of 16.16 fractions also within that structure, and repeats over a number of objects. The results of the product are deposited in the destination arrays pointed to by object structures pointed to by the objectlist array.

The object structure argument defines offsets within objects to the elements to be manipulated. The object structure has a pointer to an array of points associated with the object, a pointer to an array of transformed points, a count of the number of points, and an orientation matrix. The definition for ObjOffset1 is as follows:

```
typedef struct ObjOffset1 {int32 oo1_DestArrayPtrOffset; int32
oo1_SrcArrayPtrOffset; int32 oo1_MatOffset; int32 oo1_CountOffset;}
ObjOffset1;
```

- oo1_DestArrayPtrOffset is the offset (in bytes) from the beginning of an object structure to the location of the pointer to an array of vectors to fill with the results of the multiplies.
- oo1_SrcArrayPtrOffset is the offset (in bytes) from the beginning of an object structure to the location of the pointer to an array of vectors to be multiplied.
- oo1_MatOffset is the offset (in bytes) from the beginning of an object structure to the location of the matrix to be multiplied.
- oo1_CountOffset is the offset (in bytes) from the beginning of an object structure to the location of the count of the number of vectors to be multiplied.

Arguments

objectlist
: An array of pointers to object structures to modify.

offsetstruct
: A pointer to the source object structure that defines offsets within object's to the elements to be

multiplied.

count
: The number of vectors for the multiplication.

Implementation

SWI implemented in operamath V20.

Associated Files

operamath.h

See Also

`MulObjectVec3Mat33_F16()`

MulVec3Mat33DivZ_F16

Computes the product of a 3x3 matrix and a vector, then multiplies the x, y elements of the result by the ratio n/z.

Synopsis

```
void MulVec3Mat33DivZ_F16( vec3f16 dest, vec3f16 vec,mat33f16 mat,
frac16 n )
```

Description

This function multiplies a 3x3 matrix of 16.16 fractions by a 3-coordinate vector of 16.16 values, then multiplies the x and y elements of the result vector {x, y, z} by the ratio n/z, and deposits the result vector {x*n/z, y*n/z, z} in the product vector dest. This function can be used to perform the final projection transformation on points just before rendering.

Arguments

dest
: The destination vector to store the results.

vec
: The vector to multiply.

mat
: The matrix to multiply.

n
: The scale factor to be multiplied by x and y projected results.

Implementation

SWI implemented in operamath V20.

Associated Files

operamath.h

See Also

MulVec3Mat33_F16(), MulManyVec3Mat33DivZ_F16()

Neg32

Computes the two's complement of a 32-bit integer.

Synopsis

```
int32 Neg32( int32 x )
```

Description

This macro returns the two's complement of a 32-bit integer.

Arguments

x
: A 32-bit integer for which to get the two's complement.

Return Value

The function returns the two's complement of x.

Implementation

Macro implemented in operamath.h V20.

Associated Files

operamath.h

See Also

NegF16(), NegF30(), NegF14(), Neg64(), NegF32()

NegF16

Computes the two's complement of a 16.16 fraction.

Synopsis

```
frac16 NegF16( frac16 x )
```

Description

This macro returns the two's complement of a 16.16 fraction.

Arguments

x
: A 16.16 fraction for which the two's complement is desired.

Return Value

The function returns the two's complement of x.

Implementation

Macro implemented in operamath.h V20.

Associated Files

operamath.h

See Also

NegF14(), NegF30(), Neg32(), Neg64(), NegF32()

NegF14

Computes the two's complement of a 2.14 fraction.

Synopsis

```
frac14 NegF14( frac14 x )
```

Description

This macro returns the two's complement of a 2.14 fraction.

Arguments

x
: A 2.14 fraction for which the two's complement is desired.

Return Value

The function returns the two's complement of x.

Implementation

Macro implemented in operamath.h V20.

Associated Files

operamath.h

See Also

NegF16(), NegF30(), Neg32(), Neg64(), NegF32()

NegF30

Computes the two's complement of a 2.30 fraction.

Synopsis

```
frac30 NegF30( frac30 x )
```

Description

This function returns the two's complement of a 2.30 fraction.

Arguments

x
: A 2.30 fraction for which the two's complement is desired.

Return Value

The function returns the two's complement of x.

Implementation

Macro implemented in operamath.h V20.

Associated Files

operamath.h, operamath.lib

See Also

NegF16(), NegF14(), Neg32(), Neg64(), NegF32()

Neg64

Computes the two's complement of a 64-bit integer.

Synopsis

```
void Neg64( int64 *dest, int64 *src )
```

Description

This function computes the two's complement of a 64-bit integer. The two's complement of the location pointed to by src is deposited in the location pointed to by dest.

Arguments

dest
: A pointer to the location to store the 64-bit integer result.

src
: A pointer to the location of the 64-bit integer to be acted upon.

Implementation

Folio call implemented in operamath V20.

Associated Files

operamath.h, operamath.lib

See Also

NegF16(), NegF14(), Neg32(), NegF30(), NegF32()

NegF32

Computes the two's complement of a 32.32 fraction.

Synopsis

```
void NegF32( frac32 *dest, frac32 *src )
```

Description

This function returns the two's complement of a 32.32 fraction.

Arguments

dest
: A pointer to the location to store the frac32 result.

src
: A pointer to the location of the 32.32 fraction to be acted upon.

Return Value

The function returns the two's complement of x.

Implementation

Folio call implemented in operamath V20.

Associated Files

operamath.h, operamath.lib

See Also

NegF16(), NegF14(), Neg32(), Neg64(), NegF30()

NegF60

Computes the two's complement of a 4.60 fraction.

Synopsis

```
void NegF60( frac60 *dest, frac60 *src )
```

Description

This function returns the two's complement of a 4.60 fraction.

Arguments

dest
: A pointer to the location to store the 4.60 fraction result.

src
: A pointer to the location of the 4.60 fraction to be acted upon.

Return Value

The function returns the two's complement of x.

Implementation

Folio call implemented in operamath V20.

Associated Files

operamath.h, operamath.lib

See Also

NegF16(), NegF14(), Neg32(), Neg64(), NegF30(), NegF32()

Not32

Computes one's complement of a 32-bit integer.

Synopsis

```
int32 Not32( int32 x )
```

Description

This macro returns the one's complement of a 32-bit integer.

Arguments

x
: A 32-bit integer for which the one's complement is desired.

Return Value

The function returns the one's complement of x.

Implementation

Macro implemented in operamath.h V20.

Associated Files

operamath.h

See Also

NotF16(), NotF30(), NotF14(), Not64(), NotF32()

NotF16

Computes one's complement of a 16.16 fraction.

Synopsis

```
frac16 NotF16( frac16 x )
```

Description

This macro returns the one's complement of a 16.16 fraction.

Arguments

x
: A 16.16 fraction for which the one's complement is desired.

Return Value

The function returns the one's complement of x.

Implementation

Macro implemented in operamath.h V20.

Associated Files

operamath.h

See Also

NotF14(), NotF30(), Not32(), Not64(), NotF32()

NotF14

Computes one's complement of a 2.14 fraction.

Synopsis

```
frac14 NotF14( frac14 x )
```

Description

This macro returns the one's complement of a 2.14 fraction.

Arguments

x
: A 2.14 fraction for which the one's complement is desired.

Return Value

The function returns the one's complement of x.

Implementation

Macro implemented in operamath.h V20.

Associated Files

operamath.h

See Also

NotF16(), NotF30(), Not32(), Not64(), NotF32()

NotF30

Computes one's complement of a 2.30 fraction.

Synopsis

```
frac30 NotF30( frac30 x )
```

Description

This macro returns the one's complement of a 2.30 fraction.

Arguments

x
: A 2.30 fraction for which the one's complement is desired.

Return Value

The function returns the one's complement of x.

Implementation

Macro implemented in operamath.h V20.

Associated Files

operamath.h

See Also

NotF16(), NotF14(), Not32(), Not64(), NotF32()

Not64

Computes one's complement of a 64-bit integer.

Synopsis

```
void Not64( int64 *dest, int64 *src )
```

Description

This function computes the one's complement of a 64-bit integer in the location pointed to by src. The result is deposited in the location pointed to by dest.

Arguments

dest
: A pointer to the location to store the 64-bit integer result.

src
: A pointer to the location of the 64-bit integer to be acted upon.

Implementation

Folio call implemented in operamath V20.

Associated Files

operamath.h, operamath.lib

See Also

NotF16(), NotF14(), Not32(), NotF30(), NotF32()

NotF32

Computes one's complement of a 32.32 number.

Synopsis

```
void NotF32( frac32 *dest, frac32 *src )
```

Description

This function computes the one's complement of a 32.32 number in the location pointed to by src. The result is deposited in the location pointed to by dest.

Arguments

dest
: A pointer to the location to store the 32.32 number result.

src
: A pointer to the location of the 32.32 number to be acted upon.

Implementation

Folio call implemented in operamath V20.

Associated Files

operamath.h, operamath.lib

See Also

NotF16(), NotF14(), Not32(), NotF30(), NotF60(), Not64()

NotF60

Computes one's complement of a 4.60 number.

Synopsis

```
void NotF60( frac60 *dest, frac60 *src )
```

Description

This function computes the one's complement of a 4.60 number in the location pointed to by src. The result is deposited in the location pointed to by dest.

Arguments

dest
: A pointer to the location to store the 4.60 number result.

src
: A pointer to the location of the 4.60 number to be acted upon.

Implementation

Folio call implemented in operamath V20.

Associated Files

operamath.h, operamath.lib

See Also

NotF16(), NotF14(), Not32(), NotF30(), Not64(), NotF32()

RecipSF16

Computes the reciprocal of a signed 16.16 number.

Synopsis

```
frac16 RecipSF16( frac16 d )
```

Description

This function computes the reciprocal of a signed 16.16 number and returns the 16.16 result. An overflow is signaled by all bits set in the return value.

Arguments

d
: 16.16 format fractions.

Return Value

The function returns the reciprocal number.

Implementation

Folio call implemented in operamath V20.

Associated Files

operamath.h, operamath.lib

See Also

RecipUF16()

RecipUF16

Computes the reciprocal of an unsigned 16.16 number.

Synopsis

```
ufrac16 RecipUF16( ufrac16 d )
```

Description

This function computes the reciprocal of an unsigned 16.16 number and returns the 16.16 result. An overflow is signaled by all bits set in the return value.

Arguments

d
: 16.16 format fractions.

Return Value

The function returns the reciprocal number.

Implementation

Folio call implemented in operamath V20.

Associated Files

operamath.h, operamath.lib

See Also

RecipSF16()

Sqrt32

Computes square root of an unsigned 32-bit number.

Synopsis

```
uint32 Sqrt32( uint32 x )
```

Description

This function computes the positive square root of an unsigned 32-bit number.

Arguments

x
: The number for which the square root is desired.

Return Value

The function returns the square root of x.

Implementation

Folio call implemented in operamath V20.

Associated Files

operamath.h, operamath.lib

See Also

SqrtF16(), Sqrt64_32(), SqrtF32_F16()

SqrtF16

Computes the square root of an unsigned 16.16 number.

Synopsis

```
ufrac16 SqrtF16( ufrac16 x )
```

Description

This function computes the positive square root of an unsigned 16.16 number.

Arguments

x
: The number for which the square root is desired.

Return Value

The function returns the square root of x.

Implementation

Folio call implemented in operamath V20.

Associated Files

operamath.h, operamath.lib

See Also

Sqrt32(), Sqrt64_32(), SqrtF32_F16()

Sqrt64_32

Computes the 32-bit square root of an unsigned 64-bit integer.

Synopsis

```
uint32 Sqrt64_32( uint64 *x )
```

Description

This function computes the 32-bit square root of an unsigned 64-bit integer.

Arguments

x
: A pointer to the 64-bit number for which the square root is desired.

Return Value

The function returns the 32-bit square root of x.

Implementation

Folio call implemented in operamath V20.

Associated Files

operamath.h, operamath.lib

See Also

Sqrt32(), SqrtF32_F16(), SqrtF60_F30()

SqrtF32_F16

Computes the square root of a 32.32 fraction.

Synopsis

```
ufrac16 SqrtF32_F16( ufrac32 *x )
```

Description

This function computes the 16.16 fraction square root of a 32.32 fraction.

Arguments

x
: A pointer to the 32.32 fraction for which the square root is desired.

Return Value

The function returns the 16.16 fraction square root of x.

Implementation

Folio call implemented in operamath V20.

Associated Files

operamath.h, operamath.lib

See Also

Sqrt32(), Sqrt64_32(), SqrtF60_F30()

SqrtF60_F30

Computes the square root of a 4.60 fraction as a 2.30 fraction.

Synopsis

```
frac30 SqrtF60_F30( frac60 *x )
```

Description

This function computes the 2.30 fraction square root of a 4.60 fraction.

Arguments

x
: A pointer to the 4.60 fraction for which the square root is desired.

Return Value

The function returns the 2.30 fraction square root of x.

Implementation

Folio call implemented in operamath V20.

Associated Files

operamath.h, operamath.lib

See Also

Sqrt32(), Sqrt64_32(), SqrtF32_F16()

Square64

Squares a 64-bit integer.

Synopsis

```
void Square64( uint64 *p, int64 *m )
```

Description

This function computes the square of a 64-bit integer in the location pointed to by m. The result is deposited in the location pointed to by p.

Arguments

p
: A pointer to the result of the square operation.

m
: A pointer to the location of the 64-bit integer to be squared.

Implementation

Folio call implemented in operamath V20.

Associated Files

operamath.h, operamath.lib

Caveats

The function does not detect overflows.

See Also

SquareSF16(), SquareUF16()

SquareSF16

Squares a signed 16.16 fraction.

Synopsis

```
ufrac16 SquareSF16( frac16 m )
```

Description

This function computes the square of a 16.16 number. The lower bits of the result are truncated.

Arguments

m
: The number to be squared.

Return Value

The function returns the square of m.

Implementation

Folio call implemented in operamath V20.

Associated Files

operamath.h, operamath.lib

Caveats

The function does not detect overflows.

See Also

SquareUF16(), Square64()

SquareUF16

Squares an unsigned 16.16 fraction.

Synopsis

```
ufrac16 SquareUF16( ufrac16 m )
```

Description

This function computes the square of an unsigned 16.16 number. The lower bits of the result are truncated.

Arguments

m
: The number to be squared.

Return Value

The function returns the square of m.

Implementation

Folio call implemented in operamath V20.

Associated Files

operamath.h, operamath.lib

Caveats

The function does not detect overflows.

See Also

SquareSF16(), Square64()

Sub32

Subtracts a 32-bit integer from another.

Synopsis

```
int32 Sub32( int32 x, int32 y )
```

Description

This macro returns the difference between two 32-bit integers.

Arguments

x, y Two 32-bit integers for the subtraction. y is subtracted from x.

Return Value

The macro returns the value (x - y).

Implementation

Macro implemented in operamath.h V20.

Associated Files

operamath.h

See Also

SubF32(), SubF60(), SubF16(), SubF30(), SubF14(), Sub64()

SubF32

Subtracts a 32.32 fraction from another.

Synopsis

```
void SubF32( frac32 *r, frac32 *s1, frac32 *s2 )
```

Description

This function subtracts two 32.32 fractions and deposits the result in the location pointed to by r.

Arguments

r
: A pointer to the 32.32 fraction result of the subtraction.

s1, s2
: Pointers to the two 32.32 fractions for the subtraction. s2 is subtracted from s1.

Implementation

Folio call implemented in operamath V20.

Associated Files

operamath.h, operamath.lib

See Also

Sub64(), SubF60(), SubF16(), SubF30(), SubF14(), Sub32()

Sub64

Subtracts a 64-bit integer from another.

Synopsis

```
void Sub64( int64 *r, int64 *s1, int64 *s2 )
```

Description

This function subtracts one 64-bit integer from another and deposits the 64-bit result in the location pointed to by r.

Arguments

r
: A pointer to the 64-bit result of the subtraction.

s1, s2
: Pointers to the two 64-bit numbers for the subtraction. s2 is subtracted from s1.

Implementation

Folio call implemented in operamath V20.

Associated Files

operamath.h, operamath.lib

See Also

SubF32(), SubF60(), SubF16(), SubF30(), SubF14(), Sub32()

SubF60

Subtracts a 4.60 fraction from another.

Synopsis

```
void SubF60( frac60 *r, frac60 *s1, frac60 *s2 )
```

Description

This function subtracts two 4.60 fractions and deposits the result in the location pointed to by r.

Arguments

r
: A pointer to the 4.60 fraction result of the subtraction.

s1, s2
: Pointers to the two 4.60 fractions for the subtraction. s2 is subtracted from s1.

Implementation

Folio call implemented in operamath V20.

Associated Files

operamath.h, operamath.lib

See Also

Sub64(), SubF32(), SubF16(), SubF30(), SubF14(), Sub32()

SubF16

Subtracts a 16.16 fraction from another.

Synopsis

```
frac16 SubF16( frac16 x, frac16 y )
```

Description

This macro returns the difference between two 16.16 fractions.

Arguments

x, y
: Two 16.16 fractions for the subtraction. y is subtracted from x.

Return Value

The macro returns the value (x - y).

Implementation

Macro implemented in operamath.h V20.

Associated Files

operamath.h

See Also

SubF32(), SubF60(), Sub32(), SubF30(), SubF14(), Sub64()

SubF30

Subtracts a 2.30 fraction from another.

Synopsis

```
frac30 SubF30( frac30 x, frac30 y )
```

Description

This macro returns the difference between two 2.30 fractions.

Arguments

x, y
: Two 2.30 fractions for the subtraction. y is subtracted from x.

Return Value

The macro returns the value (x - y).

Implementation

Macro implemented in operamath.h V20.

Associated Files

operamath.h

See Also

SubF32(), SubF60(), Sub32(), SubF16(), SubF14(), Sub64()

SubF14

Subtracts a 2.14 fraction from another.

Synopsis

```
frac14 SubF14( frac14 x, frac14 y )
```

Description

This macro returns the difference between two 2.14 fractions.

Arguments

x, y
: Two 2.14 fractions for the subtraction. y is subtracted from x.

Return Value

The macro returns the value (x - y).

Implementation

Macro implemented in operamath.h V20.

Associated Files

operamath.h

See Also

SubF32(), SubF60(), Sub32(), SubF30(), SubF16(), Sub64()

Transpose33_F16

Transposes a 3x3 matrix of 16.16 values.

Synopsis

```
void Transpose33_F16( mat33f16 dest, mat33f16 src )
```

Description

This function transposes a 3-by-3 matrix of 16.16 values and deposits the result in the matrix dest.

Arguments

dest
: Pointer to destination matrix to store the result.

src
: The matrix to transpose.

Implementation

Folio call implemented in operamath V20.

Associated Files

operamath.h, operamath.lib

See Also

Transpose44_F16()

Transpose44_F16

Transposes a 4x4 matrix of 16.16 values.

Synopsis

```
void Transpose44_F16( mat44f16 dest, mat44f16 src )
```

Description

This function transposes a 4 x 4 matrix of 16.16 values and deposits the result in the matrix dest.

Arguments

dest
: Pointer to destination matrix to store the result.

src
: The matrix to transpose.

Implementation

Folio call implemented in operamath V20.

Associated Files

operamath.h, operamath.lib

See Also

Transpose33_F16()

Event Broker Calls

This chapter lists event broker calls in alphabetical order. The list below is a quick summary of each, followed by the page number where you'll find the procedure described.

Timer Calls

This chapter lists timer calls in alphabetical order. The list below is a quick summary of each, followed by the page number where you'll find the procedure described.

AddTimes

Adds two time values together.

Synopsis

```
void AddTimes(const TimeVal *tv1, const TimeVal *tv2,TimeVal *result);
```

Description

Adds two time values together, yielding the total time for both.

Arguments

tv1
: The first time value to add.

tv2
: The second time value to add.

result
: A pointer to the location where the resulting time value will be stored. This pointer can match either tv1 or tv2.

Implementation

Convenience call implemented in clib.lib V24.

Associated Files

time.h, clib.lib

See Also

SubTimes(), CompareTimes()

SubTimes

Subtracts one time value from another.

Synopsis

```
void SubTimes(const TimeVal *tv1, const TimeVal *tv2,TimeVal *result);
```

Description

Subtracts two time values, yielding the difference in time between the two.

Arguments

tv1
: The first time value.

tv2
: The second time value.

result
: A pointer to the location where the resulting time value will be stored. This pointer can match either of tv1 or tv2. The value stored corresponds to (tv2 - tv1).

Implementation

Convenience call implemented in clib.lib V24.

Associated Files

time.h, clib.lib

See Also

AddTimes(), CompareTimes()

CompareTimes

Compares two time values.

Synopsis

```
int32 CompareTimes(const TimeVal *tv1,const TimeVal *tv2);
```

Description

Compares two time values to determine which came first.

Arguments

tv1
: The first time value.

tv2
: The second time value.

Return Value

< 0 if (tv1 < tv2)

== 0 if (tv1 == tv2)

> 0 if (tv1 > tv2)

Implementation

Convenience call implemented in clib.lib V24.

Associated Files

time.h, clib.lib

See Also

AddTimes(), SubTimes(), TimeLaterThan(), TimeLaterThanOrEqual()

TimeLaterThan

Returns whether a time value comes before another.

Synopsis

```
bool TimeLaterThan(const TimeVal *tv1,const TimeVal *tv2);
```

Description

Returns whether tv1 comes chronologically after tv2.

Arguments

tv1
: The first time value.

tv2
: The second time value.

Return Value

TRUE
: if tv1 comes after tv2

FALSE
: if tv1 comes before or is the same as tv2

Implementation

Macro implemented in time.h V24.

Associated Files

time.h

See Also

CompareTimes(), TimeLaterThanOrEqual()

TimeLaterThanOrEqual

Returns whether a time value comes before or at the same time as another.

Synopsis

```
bool TimeLaterThanOrEqual(const TimeVal *tv1,const TimeVal *tv2);
```

Description

Returns whether tv1 comes chronologically after tv2, or is the same as tv2.

Arguments

tv1
: The first time value.

tv2
: The second time value.

Return Value

TRUE
: if tv1 comes after tv2 or is the same as tv2

FALSE
: if tv1 comes before tv2

Implementation

Macro implemented in time.h V24.

Associated Files

time.h

See Also

CompareTimes(), TimeLaterThan()

CreateTimerIOReq

Creates a timer device I/O request.

Synopsis

```
Item CreateTimerIOReq(void);
```

Description

Creates an I/O request for communication with the timer device.

Return Value

Returns a timer I/O request item, or a negative error code for failure.

Implementation

Convenience call implemented in clib.lib V24.

Associated Files

time.h, clib.lib

See Also

DeleteTimerIOReq(), WaitTime(), WaitUntil()

DeleteTimerIOReq

Delete a timer device I/O request.

Synopsis

```
Err DeleteTimerIOReq(Item ioreq);
```

Description

Frees any resources used by a previous call to `CreateTimerIOReq()`.

Arguments

ioreq
: The I/O request item, as returned by a previous call to CreateTimerIOReq().

Return Value

Return >=0 if successful, or a negative error code for failure.

Implementation

Convenience call implemented in clib.lib V24.

Associated Files

time.h, clib.lib

See Also

`CreateTimerIOReq()`, `WaitTime()`, `WaitUntil()`

WaitTime

Waits for a given amount of time to pass.

Synopsis

```
Err WaitTime(Item ioreq, uint32 seconds, uint32 micros);
```

Description

Puts the current context to sleep for a specific amount of time.

Arguments

ioreq
: An active timer device I/O request, as obtained from CreateTimerIOReq().

seconds
: The number of seconds to wait for.

micros
: The number of microseconds to wait for.

Return Value

Returns >=0 if successful, or a negative error code if it fails.

Implementation

Convenience call implemented in clib.lib V24.

Associated Files

time.h, clib.lib

See Also

CreateTimerIOReq(), DeleteTimerIOReq(), WaitUntil()

WaitTime

WaitUntil

Waits for a given amount of time to pass.

Synopsis

```
Err WaitTime(Item ioreq, uint32 seconds, uint32 micros);
```

Description

Puts the current context to sleep until the system clock reaches a given time.

Arguments

ioreq
: An active timer device I/O request, as obtained from CreateTimerIOReq().

seconds
: The seconds value that the timer must reach.

micros
: The microseconds value that the timer must reach.

Return Value

Returns >= 0 for success, or a negative error code for failure.

Implementation

Convenience call implemented in clib.lib V24.

Associated Files

time.h, clib.lib

See Also

CreateTimerIOReq(), DeleteTimerIOReq(), WaitTime()

JString Folio Calls

This chapter describes the international folio procedure calls. The following list provides a brief description of each call.

- `ConvertASCII2ShiftJIS` Converts an ASCII string to Shift-JIS.
- `ConvertShiftJIS2ASCII` Converts a Shift-JIS string to ASCII.
- `ConvertShiftJIS2UniCode` Converts a Shift-JIS string to UniCode.
- `ConvertUniCode2ShiftJIS` Converts a UniCode string to Shift-JIS.

Compression Folio Calls

This chapter describes the compression folio procedure calls. The following list provides a brief description of each call.

- `CreateCompressor` Creates a compression engine.
- `CreateDecompressor` Creates a decompression engine.
- `DeleteCompressor` Deletes a compression engine.
- `DeleteDecompressor` Deletes a decompression engine.
- `FeedCompressor` Gives data to a compression engine.
- `FeedDecompressor` Gives data to the decompression engine.
- `GetCompressorWorkBufferSize` Gets the size of the work buffer needed by a compression engine.
- `GetDecompressorWorkBufferSize` Gets the size of the work buffer needed by a decompression engine.
- `SimpleCompress` Compresses some data in memory.
- `SimpleDecompress` Decompresses some data in memory.

CreateCompressor

Creates a compression engine.

Synopsis

```
Err CreateCompressor(Compressor **comp, CompFunc cf,const TagArg
*tags);
Err CreateCompressorVA(Compressor **comp, CompFunc cf,uint32 tags,
...);
```

Description

Creates a compression engine. Once the engine is created, you can call `FeedCompressor()` to have the engine compress the data you supply.

Arguments

comp
: A pointer to a compressor variable, where a handle to the compression engine can be put.

cf
: A data output function. Every word of compressed data is sent to this function. This function is called with two parameters: one is a user-data value as supplied with the COMP_TAG_USERDATA tag. The other is the word of compressed data being output by the compressor.

tags
: A pointer to an array of tag arguments containing extra data for this function. See below for a description of the tags supported.

Tag Arguments

The following tag arguments may be supplied in array form to this function. The array must be terminated with TAG_END.

COMP_TAG_WORKBUFFER(void *)
: A pointer to a work buffer. This buffer is used by the compressor to store state information. If this tag is not supplied, the buffer is allocated and freed by the folio automatically. To obtain the

required size for the buffer, call the GetCompressorWorkBufferSize() function. The buffer you supply must be aligned on a 4-byte boundary, and must remain valid until DeleteCompressor() is called. When you supply a work buffer, this routine allocates no memory of its own.

COMP_TAG_USERDATA(void *)
A value that the compressor will pass to cf when it is called. This value can be anything you want. For example, it can be a pointer to a private data structure containing some context such as a file handle. If this tag is not supplied, then NULL is passed to cf when it is called.

Return Value

Returns >= 0 for success, or a negative error code if the compression engine could not be created. Possible error codes include:

COMP_ERR_BADPTR
An invalid output function pointer or work buffer was supplied.

COMP_ERR_BADTAG
An unknown tag was supplied.

COMP_ERR_NOMEM
There was not enough memory to initialize the compressor.

Implementation

Folio call implemented in compression folio V24.

Associated Files

compression.h

See Also

FeedCompressor(), DeleteCompressor(), GetCompressorWorkBufferSize(), CreateDecompressor()

FeedCompressor

Gives data to a compression engine.

Synopsis

```
Err FeedCompressor(Compressor *comp, void *data,uint32 numDataWords);
```

Description

Gives data to a compressor engine for compression. As data is compressed, the call back function supplied when the compressor was created is called for every word of compressed data generated.

Arguments

comp
: An active compression handle, as obtained from CreateCompressor().

data
: A pointer to the data to compress.

numDataWords
: The number of words of data being given to the compressor.

Return Value

Returns >= 0 for success, or a negative error code if it fails. Possible error codes include:

COMP_ERR_BADPTR
: An invalid compression handle was supplied.

Implementation

Folio call implemented in compression folio V24.

Associated Files

compression.h

See Also

CreateCompressor(), DeleteCompressor()

DeleteCompressor

Deletes a compression engine.

Synopsis

```
Err DeleteCompressor(Compressor *comp);
```

Description

Deletes a compression engine previously created by `CreateCompressor()`. This flushes any data left to be output by the compressor and generally cleans things up.

Arguments

comp
: An active compression handle, as obtained from CreateCompressor(). Once this call is made, the compression handle becomes invalid and can no longer be used.

Return Value

Returns > = 0 if successful, or a negative error code if it fails. Possible error codes include:

COMP_ERR_BADPTR
: An invalid compression handle was supplied.

Implementation

Folio call implemented in compression folio V24.

Associated Files

compression.h

See Also

`CreateCompressor()`

GetCompressorWorkBufferSize

Gets the size of the work buffer needed by a compression engine.

Synopsis

```
int32 GetCompressorWorkBufferSize(const TagArg *tags);
int32 GetCompressorWorkBufferSizeVA(uint32 tags, ...);
```

Description

Returns the size of the work buffer needed by a compression engine. You can then allocate a buffer of that size and supply the pointer with the COMP_TAG_WORKBUFFER tag when creating a compression engine. If the COMP_TAG_WORKBUFFER tag is not supplied when creating a compressor, the folio automatically allocates the memory needed for the compression engine.

Arguments

tags
: A pointer to an array of tag arguments containing extra data for this function. This must currently always be NULL.

Return Value

A positive value indicates the size of the work buffer needed in bytes, while a negative value indicates an error. Possible error codes currently include:

COMP_ERR_BADTAG
: An unknown tag was supplied.

Implementation

Folio call implemented in compression folio V24.

Associated Files

compression.h

See Also

CreateCompressor(), DeleteCompressor()

CreateDecompressor

Creates a decompression engine.

Synopsis

```
Err CreateDecompressor(Decompressor **comp, CompFunc cf,const TagArg
*tags);
Err CreateDecompressorVA(Decompressor **comp, CompFunc cf,uint32 tags,
...);
```

Description

Creates a decompression engine. Once the engine is created, you can call `FeedDecompressor()` to have the engine decompress the data you supply.

Arguments

decomp
: A pointer to a decompressor variable, where a handle to the decompression engine can be put.

cf
: A data output function. Every word of decompressed data is sent to this function. This function is called with two parameters: one is a user-data value as supplied with the COMP_TAG_USERDATA tag. The other is the word of decompressed data being output by the decompressor.

tags
: A pointer to an array of tag arguments containing extra data for this function. See below for a description of the tags supported.

Tag Arguments

The following tag arguments may be supplied in array form to this function. The array must be terminated with TAG_END.

COMP_TAG_WORKBUFFER(void *)
: A pointer to a work buffer. This buffer is used by the decompressor to store state information. If this tag is not supplied, the buffer is allocated and freed by the folio automatically. To obtain the

required size for the buffer, call the GetDecompressorWorkBufferSize() function. The buffer you supply must be aligned on a 4-byte boundary, and must remain valid until DeleteDecompressor() is called. When you supply a work buffer, this routine allocates no memory of its own.

COMP_TAG_USERDATA(void *)
A value that the decompressor will pass to cf when it is called. This value can be anything you want. For example, it can be a pointer to a private data structure containing some context such as a file handle. If this tag is not supplied, then NULL is passed to cf when it is called.

Return Value

Returns >= 0 for success, or a negative error code if the decompression engine could not be created. Possible error codes include:

COMP_ERR_BADPTR
An invalid output function pointer or work buffer was supplied.

COMP_ERR_BADTAG
An unknown tag was supplied.

COMP_ERR_NOMEM
There was not enough memory to initialize the compressor.

Implementation

Folio call implemented in compression folio V24.

Associated Files

compression.h

See Also

FeedDecompressor(), DeleteDecompressor(), GetDecompressorWorkBufferSize(), CreateCompressor()

FeedDecompressor

Gives data to the decompression engine.

Synopsis

```
Err FeedDecompressor(Decompressor *decomp, void *data,uint32
numDataWords);
```

Description

Gives data to the decompressor engine for decompression. As data is decompressed, the call back function supplied when the decompressor was created is called for every word of decompressed data generated.

Arguments

decomp
: An active decompression handle, as obtained from CreateDecompressor().

data
: A pointer to the data to decompress.

numDataWords
: The number of words of compressed data being given to the decompressor.

Return Value

Returns >= 0 for success, or a negative error code if it fail. Possible error codes include:

COMP_ERR_BADPTR
: An invalid decompression handle was supplied.

Implementation

Folio call implemented in compression folio V24.

Associated Files

compression.h

See Also

`CreateDecompressor`(), `DeleteDecompressor`()

DeleteDecompressor

Deletes a decompression engine.

Synopsis

```
Err DeleteDecompressor(Decompressor *decomp);
```

Description

Deletes a decompression engine previously created by `CreateDecompressor()`. This flushes any data left to be output by the decompressor and generally cleans things up.

Arguments

decomp
: An active decompression handle, as obtained from CreateDecompressor(). Once this call is made, the decompression handle becomes invalid and can no longer be used.

Return Value

Returns >= 0 for success, or a negative error code if it fails. Possible error codes include:

COMP_ERR_BADPTR
: An invalid decompression handle was supplied.

COMP_ERR_DATAREMAINS
: The decompressor thinks it is finished, but there remains extra data in its buffers. This happens when the compressed data is somehow corrupt.

COMP_ERR_DATAMISSING
: The decompressor thinks that not all of the compressed data was given to be decompressed. This happens when the compressed data is somehow corrupt.

Implementation

Folio call implemented in compression folio V24.

Associated Files

compression.h

See Also

`CreateCompressor()`

GetDecompressorWorkBufferSize

Gets the size of the work buffer needed by a decompression engine.

Synopsis

```
int32 GetDecompressorWorkBufferSize(const TagArg *tags);
int32 GetDecompressorWorkBufferSizeVA(uint32 tags, ...);
```

Description

Returns the size of the work buffer needed by a decompression engine. You can then allocate a buffer of that size and supply the pointer with the COMP_TAG_WORKBUFFER tag when creating a decompression engine. If the COMP_TAG_WORKBUFFER tag is not supplied when creating a decompressor, the folio automatically allocates the memory needed for the decompression engine.

Arguments

tags
: A pointer to an array of tag arguments containing extra data for this function. This must currently always be NULL.

Return Value

A positive value indicates the size of the work buffer needed in bytes, while a negative value indicates an error. Possible error codes currently include:

COMP_ERR_BADTAG
: An unknown tag was supplied.

Implementation

Folio call implemented in compression folio V24.

Associated Files

compression.h

See Also

`CreateDecompressor`(), `DeleteDecompressor`()

SimpleCompress

Compresses some data in memory.

Synopsis

```
Err SimpleCompress(void *source, uint32 sourceWords,void *result,
uint32 resultWords);
```

Description

Compresses a chunk of memory to a different chunk of memory.

Arguments

source
: A pointer to memory containing the data to be compressed. This pointer must be on a 4-byte boundary.

sourceWords
: The number of words of data to compress.

result
: A pointer to where the compressed data is to be deposited.

resultWords
: The number of words available in the result buffer. If the compressed data is larger than this size, an overflow will be reported.

Return Value

If the return value is positive, it indicates the number of words copied to the result buffer. If the return value is negative, it indicates an error code. Possible error codes include:

COMP_ERR_NOMEM
: There was not enough memory to initialize the compressor.

COMP_ERR_OVERFLOW
: There was not enough room in the result buffer to hold all of the compressed data.

Implementation

Convenience call implemented in compression.lib V24.

Associated Files

compression.h

See Also

`SimpleDecompress()`

SimpleDecompress

Decompresses some data in memory.

Synopsis

```
Err SimpleDecompress(void *source, uint32 sourceWords,void *result,
uint32 resultWords);
```

Description

Decompresses a chunk of memory to a different chunk of memory.

Arguments

source
: A pointer to memory containing the data to be decompressed. This pointer must be on a 4-byte boundary.

sourceWords
: The number of words of data to decompress.

result
: A pointer to where the decompressed data is to be deposited.

resultWords
: The number of words available in the result buffer. If the decompressed data is larger than this size, an overflow will be reported.

Return Value

If positive, returns the number of words copied to the result buffer. If negative, returns an error code. Possible error codes include:

COMP_ERR_NOMEM
: There was not enough memory to initialize the decompressor.

COMP_ERR_OVERFLOW
: There was not enough room in the result buffer to hold all of the decompressed data.

Implementation

Convenience call implemented in compression.lib V24.

Associated Files

compression.h

See Also

`SimpleCompress()`

How the Compression Folio Works

The sections below describe how to compress and decompress data.

Compressing Data

To compress data, you must create a compression engine by calling `CreateCompressor()`.

You must also allocate a buffer of the correct size and supply a pointer to it with the `COMP_TAG_WORKBUFFER` tag and pass in a pointer to your output function.

Once you create the engine, call `FeedCompressor()` to have the engine compress the data you supply. `FeedCompressor()` then gives data to the compression engine, which in turn sends it to your output function, one word at a time. When the compression engine is finished compressing the data, call `DeleteCompressor()` to delete the engine and clean up.

Decompressing Data

Decompression works almost the same as compressing data. First, you create a decompression engine with `CreateDecompressor()`. Then, allocate a buffer of the correct size and supply a pointer to it with the `COMP_TAG_WORKBUFFER` tag and provide a pointer to your output function.

Feed compressed data to the decompression engine with the `FeedDecompressorEngine()` call, which in turn sends it to the output call. Again when you are finished decompressing data, call `DeleteDecompressor()` to delete the engine and clean up.

The Callback Function

When you create a compression or decompression engine, you must supply a function pointer. This function is called whenever either engine has data to output. The function then takes the data and do whatever the application wants with the data. For example, the callback function can deposit the data in a buffer, and once the engine is deleted, the buffer can be written out to a file.

The callback function has two parameters passed to it, and is defined as:

```
typedef void (* CompFunc)(void *userData, uint32 word);
```

The `userData` field is normally NULL, unless you supply the `COMP_TAG_USERDATA` tag when creating the engine. Whatever value is associated with the tag gets passed directly to the callback function. The purpose of the `userData` field is to pass things like a file pointer to the callback function. You would pass the file pointer as the data value associated with the `COMP_TAG_USERDATA` tag. The engines then pass the value to the callback function whenever it is called.

The `word` parameter to the callback is the word of data being generated by the engine. You must do something useful with this word. For example, you could copy it into a memory buffer, or write it to a file.

Controlling Memory Allocations

When you create a compression or decompression engine, the `CreateCompressor()` and `CreateDecompressor()` functions normally allocate some memory to hold the state of the engines. The compression engine needs around 50 KB of storage, while the decompressor needs around 5 KB.

If your title has sophisticated memory management needs, you can supply a memory buffer yourself to avoid having the folio allocate any memory. To do this, call the `GetCompressorWorkBufferSize()` routine to obtain the size of the buffer needed for the compression engine, or you can call `GetDecompressorWorkBufferSize()` to get the size of the buffer needed for the decompression engine. Once you have determined the size, then you can allocate a memory buffer, and pass a pointer to the buffer using the `COMP_TAG_WORKBUFFER` tag when you create the compressor or decompressor.

When you supply the work buffer, the Compression folio allocates no, or almost no, resources when a compression or decompression engine is created. This allows your title to control where memory resources come from.

Convenience Calls

There are two function calls supplied in *compression.lib* that provide a simple interface to the lower-level functions of the Compression folio. The two functions implement the most common uses for the folio functions.

The `Compress()` function compresses data from one memory buffer into another memory buffer. You give it a source buffer pointer, the size of the source buffer, a destination buffer pointer, and the size of the destination buffer. `Compress()` then compresses the data and deposits the compressed result in the destination buffer. This function returns the number of words of data copied into the destination buffer, or a negative error if something went wrong, such as a lack of memory or a lack of space in the destination buffer.

The `Decompress()` function decompresses in much the same way as the `Compress()` function compresses data. This time, the source buffer contains the compressed data, and the destination buffer gets filled with decompressed data. This function returns the number of words of decompressed data that were put in the destination buffer, or a negative error code if something went wrong.

Example

Example 1 contains the sample program *compressexample.c* (located in the Examples folder), which shows how to use the Compression folio. This program loads itself into a memory buffer, compresses the data, decompresses it, and finally compares the original data with the decompressed data to make sure the compression process was successful.

Example 1: *Example of compressing and decompressing data (compressexample.c).*

```
#include "types.h"
#include "filestream.h"
#include "filestreamfunctions.h"
#include "stdio.h"
#include "mem.h"
#include "compression.h"


/*****************************************************************************/
int main(int argc, char **argv)
{
Stream       *stream;
Err           err;
bool          same;
uint32        i;
int32         fileSize;
uint32       *originalData;
uint32       *compressedData;
uint32       *finalData;
int32         numFinalWords;
int32         numCompWords;

    err = OpenCompressionFolio();
    if (err >= 0)
    {
        stream = OpenDiskStream(argv[0],0);
        if (stream)
        {
            fileSize       = stream->st_FileLength & 0xfffffffc;
            originalData   = (uint32 *)malloc(fileSize);
            compressedData = (uint32 *)malloc(fileSize);
            finalData      = (uint32 *)malloc(fileSize);

            if (originalData && compressedData && finalData)
            {
                if (ReadDiskStream(stream,(char *)originalData,fileSize) ==

                   fileSize)
                {
                    err = Compress(originalData, fileSize / sizeof(uint32),
```

```
                    compressedData, fileSize / sizeof(uint32));
    if (err >= 0)
    {
        numCompWords = err;
        err = Decompress(compressedData, numCompWords,
                         finalData, fileSize / sizeof(uint32));
        if (err >= 0)
        {
            numFinalWords = err;
            printf("Original data size    : %d\n",fileSize /

                sizeof(uint32));
            printf("Compressed data size  : %d\n",numCompWords);
            printf("Uncompressed data size: %d\n",numFinalWords);

            same = TRUE;
            for (i = 0; i < fileSize / sizeof(uint32); i++)
            {
                if (originalData[i] != finalData[i])
                {
                    same = FALSE;
                    break;
                }
            }

            if (same)
            {
                printf("Uncompressed data matched original\n");
            }
            else
            {
                printf("Uncompressed data differed with

                        original!\n");
                for (i = 0; i < 10; i++)
                {
                    printf("orig $%08x, final $%08x, comp

                          $%08x\n",
                           originalData[i],
                           finalData[i],
                           compressedData[i]);
                }
            }
        }
        else
        {
            PrintError(NULL,"decompress data",NULL,err);
        }
    }
```

```
                    else
                    {
                        PrintError(NULL,"compress data",NULL,err);
                    }
                }
                else
                {
                    printf("Could not read whole file\n");
                }
            }
            else
            {
                printf("Could not allocate memory buffers\n");
            }

            free(originalData);
            free(compressedData);
            free(finalData);

            CloseDiskStream(stream);
        }
        else
        {
            printf("Could not open '%s' as an input file\n",argv[0]);
        }
        CloseCompressionFolio();
    }
    else
    {
        PrintError(NULL,"open the compression folio",NULL,err);
    }

    return (0);
}
```

Function Calls

The Compression folio currently includes the following calls, which are listed here by the type of function they perform.

Convenience calls

- `Compress()` Compresses a chunk of memory to a different chunk of memory.
- `Decompress()` Decompresses a chunk of memory to a different chunk of memory

The following calls are convenience calls:

Compression Calls

- `CreateCompressor()` Creates a compression engine.
- `FeedCompressor()` Gives data to a compressor engine to compress and pass to the output function.
- `DeleteCompressor()` Deletes the compressor engine.

The following calls compress data:

Decompression Calls

The following calls compress data:

- `CreateDecompressor()` Creates a decompression engine.
- `FeedDecompressor()` Gives data to a decompressor engine to have it decompressed and pass it to the output function.
- `DeleteDecompressor()` Deletes the decompressor engine.

Buffer Calls

The following calls get the size of working buffers:

- `GetCompressorWorkBufferSize()` Gets the size of the work buffer needed by a

compression engine.

- `GetDecompressorWorkBufferSize()` Gets the size of the work buffer needed by a decompression engine.

The Math Folio

The Math folio offers a large number of functions that perform many common mathematical operations. Many of these functions use the special math hardware built into the 3DO architecture to give better performance.

This chapter organizes the many Math folio calls in logical groupings so you can find the appropriate call for the math operation you want to perform. Most of these calls are very straightforward. For detailed information on the operation of each call, see the *3DO System Programmer's Reference*.

Like all other folios, each task or thread that uses calls from the Math folio must open the folio explicitly using `OpenMathFolio()`.

This chapter contains the following topics:

Dealing With Overflow

Most Math folio function calls do not detect overflow. When overflow occurs, the result is most often a garbage value. The exceptions to this rule are many division calls. If overflow occurs when using an unsigned division call, the call normally returns a value with all bits set (the maximum possible value). When overflow occurs using a signed division call, the call returns either a value with all bits set (the smallest possible negative value), or all bits clear except the least-significant bit (the smallest positive value).

List of Math Calls

Addition Calls

These calls add two values of the same data type together and return the result in the same data type.

Integers

- Add32() Adds 32-bit integers.
- Add64() Adds 64-bit integers.

Fractions

- AddF14() Adds 2.14 format fixed-point fractions.
- AddF16() Adds 16.16 format fixed-point fractions.
- AddF30() Adds 2.30 format fixed-point fractions.
- AddF32() Adds 32.32 format fixed-point fractions.
- AddF60() Adds 4.60 format fixed-point fractions.

Subtraction Calls

These calls subtract one value from another and return the result. Both the values and the result are the same data type.

Integers

- Sub32() Subtracts two 32-bit integers.
- Sub64() Subtracts two 64-bit integers.

Fractions

- SubF14() Subtracts two 2.14 fractions.

- SubF16() Subtracts two 16.16 fractions.
- SubF30() Subtracts two 2.30 fractions.
- SubF32() Subtracts two 32.32 fractions.
- SubF60() Subtracts two 4.60 fractions.

Division Calls

These calls take two values of the same data type and divide the first by the second to return a quotient and/or a remainder of the same data type.

Integers

- DivRemS32() 32-bit quotient, remainder of a 32-bit division.
- DivS64() 64-bit quotient, remainder of a 64-bit division.
- DivRemU32() 32-bit quotient, remainder of a 32-bit division.
- DivU64() 64-bit quotient, remainder of a 64-bit division.

Fractions

- DivSF16() 16.16 quotient of a 16.16 division.
- DivRemSF16() 16.16 quotient, remainder of a 16.16 division.
- DivUF16() 16.16 quotient of a 16.16 division.
- DivRemUF16() 16.16 quotient, remainder of a 16.16 division.

Same-Product Multiplication Calls

These calls multiply two values that can be numbers, vectors, or matrices. They return products of the same type as the multipliers.

Cross-Product Multiplication

- Cross3_F16() 16.16 cross-product of 2 vectors of 16.16 values.

Integer Multiplication

- Mul32() Multiplies two 32-bit integers; 32-bit integer result.
- Mul64() Multiplies two 64-bit integers; 64-bit integer result.

Fraction Multiplication

- MulF14() Multiplies two 2.14 fractions; 2.14 fraction result.
- MulSF16() Multiplies two signed 16.16 fractions; 16.16 result.
- MulUF16() Multiplies two unsigned 16.16 fractions; 16.16 result.

Scalar Multiplication

- MulScalarF16() Multiplies 16.16 scalar by an array of 16.16 values; 16.16 array result.

Vector by Vector Multiplication

- MulManyF16() Multiplies an array of 16.16 values by another array of 16.16 values.

Matrix by Matrix Multiplication

These calls multiply a matrix within an object structure by an external matrix.

- MulMat33Mat33_F16() Multiplies two 3x3 matrices of 16.16 values.
- MulMat44Mat44_F16()Multiplies two 4x4 matrices of 16.16 values.
- MulObjectMat33_F16()Multiplies matrix within an object structure by an external 3x3 matrix of 16.16 values.
- MulObjectMat44_F16()Multiplies matrix within an object structure by an external 4x4 matrix of 16.16 values.

Vector by Matrix Multiplication

These calls multiply many vectors within an object structure by an external matrix.

- MulManyVec3Mat33_F16() Multiplies 16.16 vector(s) by a 3x3 matrix of 16.16 values.
- MulManyVec4Mat44_F16() Multiplies 16.16 vector(s) by a 4x4 matrix of 16.16 values.
- MulVec3Mat33_F16() Multiplies a 16.16 vector and a 3x3 matrix.
- MulVec4Mat44_F16() Multiplies a 16.16 vector and a 4x4 matrix.

- MulObjectVec3Mat33_F16() Multiplies many 16.16 vectors within an object structure by a 3x3 matrix of 16.16 values.
- MulObjectVec4Mat44_F16() Multiplies many 16.16 vectors within an object structure by a 4x4 matrix of 16.16 values.
- MulVec3Mat33DivZ_F16() Multiplies a 3x3 matrix of 16.16 fractions by a 3-coordinate vector of 16.16 values, then multiply the x and y elements of the results vector by the ratio n/z.
- MulManyVec3Mat33DivZ_F16() Multiplies a 3x3 matrix of 16.16 fractions by one or more 3-coordinate vectors of 16.16 values, then multiply the x and y elements of the result vector by the ration n/z.

Different-Product Multiplication Calls

These calls multiply two values and return a product of a different type than the arguments.

Standard Multiplication

- MulS32_64() Multiplies two 32-bit integers; 64-bit result.
- MulSF16_F32() Multiplies two signed 16.16 numbers; 32.32 result.
- MulSF30_F60() Multiplies two signed 2.30 numbers; a 4.60 result.
- MulU32_64() Multiplies two unsigned 32-bit integers; 64-bit result.
- MulUF16_F32() Multiplies two unsigned 16.16 numbers; 32.32 result.
- MulUF30_F60() Multiplies two unsigned 2.30 numbers; 4.60 result.

Dot Product Multiplication

- Dot3_F16() Dot product of 2 three-coordinate vectors of 16.16 values.
- Dot4_F16() Dot product of 2 four-coordinate vectors of 16.16 values.

Matrix Transposition Calls

These calls transpose matrices.

- Transpose33_F16() Transposes a 3x3 matrix of 16.16 values.
- Transpose44_F16() Transposes a 4x4 matrix of 16.16 values.

Geometry Calls

These calls compute geometric functions: sines, cosines, and so on.

Arctangent

- `Atan2F16()` Arctangent of a ratio.

Cosines

- `CosF16()` 16.16 cosine of a 16.16 fraction angle.
- `CosF30()` 2.30 cosine of a 16.16 fraction.
- `CosF32()` 32.32 cosine of a 16.16 fraction.

Sines

- `SinF16()` 16.16 sine of a 16.16 fraction angle.
- `SinF30()` 2.30 sine of a 16.16 fraction.
- `SinF32()` 32.32 sine of a 16.16 fraction.

Conversion Calls

These calls convert a value in one data type to the corresponding value in another data type.

Integers to Fractions

- `Convert32_F16()` Converts 32-bit integer to 16.16 fraction.
- `Convert32_F32()` Converts 32-bit integer to 32.32 fraction.

Integers to Integers

- `ConvertS32_64()` Converts 32-bit integer to 64-bit integer.
- `ConvertU32_64()` Converts unsigned 32-bit integer to unsigned 64-bit integer.

Fractions to Fractions

- `ConvertF14_F16()` Converts 2.14 fraction to 16.16 fraction.
- `ConvertF16_F14()` Converts 16.16 fraction to 2.14 fraction.
- `ConvertF16_F30()` Converts 16.16 fraction to 2.30 fraction.
- `ConvertF30_F16()` Converts 2.30 fraction to 16.16 fraction.
- `ConvertF32_F16()` Converts 32.32 fraction to 16.16 fraction.
- `ConvertSF16_F32()` Converts 16.16 fraction to 32.32 fraction.
- `ConvertUF16_F32()` Converts unsigned 16.16 fraction to unsigned 32.32 fraction.

Fraction to Integers

- `ConvertF16_32()` Converts 16.16 fraction to 32-bit integer.

Comparative Calls

These calls compare two values of the same data type and return a result that is a positive value if the first value is greater than the second, a 0 if the two values are equal, and a negative number if the second value is greater than the first.

Integers

- `CompareS64()` Compares signed 64-bit integers.
- `CompareU64()` Compares unsigned 64-bit integers.

Fractions

- `CompareSF32()` Compares signed 32.32 format fractions.
- `CompareSF60()` Compares signed 4.60 format fractions.
- `CompareUF32()` Compares unsigned 32.32 format fractions.
- `CompareUF60()` Compares unsigned 4.60 format fractions.

One's and Two's Complement Calls

These calls return one's and two's complements of a value.

Two's Complements

- Neg32() Two's complement of a 32-bit integer.
- NegF14() Two's complement of a 2.14 fraction.
- NegF16() Two's complement of a 16.16 fraction.
- NegF30() Two's complement of a 2.30 fraction.
- Neg64() Two's complement of a 64-bit integer.
- NegF32() Two's complement of a 32.32 number.

One's Complements

- Not32() One's complement of a 32-bit integer.
- NotF14() One's complement of a 2.14 fraction.
- NotF16() One's complement of a 16.16 fraction.
- NotF30() One's complement of a 2.30 fraction.
- Not64() One's complement of a 64-bit integer.
- NotF32() One's complement of a 32.32 number.

Reciprocal Calls

These calls return the reciprocal of a number.

- RecipSF16() Reciprocal of a signed 16.16 number.
- RecipUF16() Reciprocal of an unsigned 16.16 number.

Squaring and Square Root Calls

These calls return the square or the square root of a number.

Squares

- Square64() Square a 64-bit integer.
- SquareSF16() Square a signed 16.16 fraction.
- SquareUF16() Square an unsigned 16.16 fraction.

Square Roots

- `Sqrt32()` Unsigned 32-bit square root of an unsigned 32-bit number.
- `SqrtF16()` 16.16 fraction square root of an unsigned 16.16 number.
- `Sqrt64_32()` 32-bit square root of an unsigned 64-bit integer.
- `SqrtF32_F16()` 16.16 square root of a32.32 fraction.

Absolue Value

The following calls compute the absolute value of a vector call.

- `AbsVec3_F16` Computes the absolute value of a vector of 16.16 values.
- `AbsVec4_F16` Computes the absolute value of a vector of 16.16 values.