Palm Computing® Platform

Developing Palm OS 3.0 Applications

Part III: Memory and Communications Management

Navigate this online document as follows:

To see bookmarks, type:	Command-7 (Mac OS) Ctrl-7 (Windows)
To navigate, click on:	any blue hypertext link any Table of Contents entry any Index entry arrows in the toolbar

Developing Palm OS 3.0 Applications

Part III: Memory and Communications Management

Contact Information:

Metrowerks U.S.A. and international	Metrowerks Corporation 2201 Donley Drive, Suite 310 Austin, TX 78758 U.S.A.
Metrowerks Canada	Metrowerks Inc. 1500 du College, Suite 300 Ville St-Laurent, QC Canada H4L 5G6
Metrowerks Mail order	Voice: 1-800-377–5416 Fax: 1-512-873–4901
3Com (Palm Computing Subsidiary) Mail Order	U.S.A.: 1-800-881-7256 Canada: 800-891-6342 elsewhere: 1-801-431-1536
Metrowerks World Wide Web	`http://www.metrowerks.com`
Palm Computing World Wide Web	`http://www.palm.com`
Registration information	`register@metrowerks.com`
Technical support	`support@metrowerks.com`
Sales, marketing, & licensing	`sales@metrowerks.com`
CompuServe	go `Metrowerks`

Table of Contents

About This Document

Developing Palm OS 3.0 Applications, Part III, is part of the Palm OS Software Development Kit. This introduction provides an overview of SDK documentation. It discusses the materials included and the conventions used in this document.

Palm OS SDK Documentation

The following documents are part of the SDK:

Document	Description
Palm OS 3.0 Tutorial	A number of Phases step developers through how to use the different parts of the system. Each phase includes example applications.
Developing Palm OS 3.0 Applications. Part I: Interface Management	A programmer's guide and reference document that introduces all important aspects of developing an applications. See What This Guide Contains for details.
Developing Palm OS 3.0 Applications. Part II: System Management	A programmer's guide and reference document for all system managers, such as the string manager or the system event manager.

Document	Description
Developing Palm OS 3.0 Applications. Part III. Memory and Communications Management	Programmer's guide and reference document for • Memory management; both the database manager and the memory manager. • The Palm OS communications library for serial communication. • The Palm OS net library, which provides basic network services. • The exchange manager and IR library, which provide infrared communication capabilities.
Palm OS 3.0 Cookbook	Information about using CodeWarrior for Palm OS to create projects and executables. Also provides a variety of design guidelines, including localization design guidelines.

What This Guide Contains

The following are chapter overviews for this guide.

- Chapter 1, "Palm OS Memory Management," helps you understand memory management on Palm OS. It first discusses memory layout and architecture, then explains how to use the three memory managers, which comprise the memory management API.
- Chapter 2, "Memory Management Functions," provides reference-style information for each memory manager function.
- Chapter 3, "Data and Resource Manager Functions," provides reference-style information for the data manager and resource manager functions.
- Chapter 4, "Palm OS Communications," discusses the communications software, which provides the serial communications capabilities for Palm OS.
- Chapter 5, "Communications Functions," provides reference information for the serial manager functions, serial link manager functions, and miscellaneous communications functions.
- Chapter 6, "Palm OS Net Library," introduces the Palm OS net library and explains how to use it.

- Chapter 7, "Net Library Functions," provides reference information for all net library functions, as well as an overview of the parallel Berkeley Sockets API calls.
- Chapter 8, "Exchange Manager," discusses the exchange manager, which provides a high-level interface to the IR communications capabilities of the Palm OS. This chapter also includes a reference for all the exchange manager functions.
- Chapter 9, "IR Library," discusses the IR library, which provides direct access to the IR communications capabilities of the Palm OS. This chapter also includes a reference for all the IR library functions.

Conventions Used in This Guide

This guide uses the following typographical conventions:

This style...	Is used for...
`fixed width font`	Code elements such as function, structure, field, bitfield.
`fixed width underline`	Emphasis (for code elements).
bold	Emphasis (for other elements).
blue and underlined	Hot links.
black and underlined	3.0 function names (headings only).
red and underlined	3.0 function names (in Table of Contents only)

1

Palm OS Memory Management

This chapter helps you understand memory use on Palm OS. It starts with an introduction to memory layout and memory architecture.

- Introduction to Memory Use on Palm OS provides information about Palm OS hardware relevant to memory management. For more information on Palm OS hardware, see “Basic Hardware” in Chapter 1 of “Developing Palm OS Applications, Part 1.”
- Memory Architecture discusses in detail how memory is structured on Palm OS. It also examines the structure of the basic building blocks of Palm OS memory: heaps, chunks, and records.

The second part of the chapter explains the different parts of the system—the managers—that you can use for memory management. The discussion of each manager includes a brief overview of the significant functions composing its API; in the online version of this book, each function name provides a hypertext link to its reference description.

- The Memory Manager maintains the location and size of each memory chunk in nonvolatile storage, volatile storage, and ROM. It provides functions for allocating chunks, disposing of chunks, resizing chunks, locking and unlocking chunks, and compacting the heap when it becomes fragmented.
- The Data Manager manages user data, which is stored in databases for convenient access.
- The Resource Manager can be used by applications to retrieve and save chunks of data. It’s similar to the data manager, but has the added capability of tagging each chunk with a unique resource type and ID. These tagged data chunks, called **resources**, reside in a resource database and commonly are used to store the application’s user interface elements (images, fonts,

dialog layouts, and so on) as well as application-specific static data (not user data or temporary data.)

Introduction to Memory Use on Palm OS

The Palm OS system software supports applications on low-cost, low-power, palm-top devices. Given these constraints, Palm OS is efficient in its use of both memory and processing resources. This section presents two aspects of Palm OS devices that contribute to this efficiency: Hardware Architecture and PC Connectivity.

Hardware Architecture

The first implementation of Palm OS provides nearly instantaneous response to user input while running on a 16 MHz Motorola® 68000 type processor with a minimum of 128K of nonvolatile storage memory and 512 KB of ROM. Subsequent Palm OS devices provide additional RAM and ROM in varying amounts.

The ROM and RAM for each Palm OS device resides on a memory module known as a **card**. Each memory card can contain ROM, RAM, or both. There is no RAM or ROM storage on the motherboard of the device.

Though all Palm OS devices available as of May 1998 hold one card in a user-accessible hardware slot, it is unwise to assume that any Palm OS device has a memory module that can be removed physically. A “card” is simply a logical construct used by the operating system—Palm OS devices can have one card, multiple cards, or no cards. For example, the Simulator provided by the Palm OS SDK can simulate a device that has two cards.

The ROM and RAM on each card is divided into one or more heaps. All the RAM-based heaps on a memory card are treated as the RAM store, and all the ROM-based heaps are treated as the ROM store. The heaps for a store do not have to be adjacent to each other in address space—they can be scattered throughout the memory space on the card—but they must all reside on the same card.

The main suite of applications provided with each Palm OS device is prebuilt into ROM. This design permits the user to replace the operating system and the entire applications suite simply by installing a single replacement module. Additional or replacement applications and system extensions can be loaded into RAM, but doing so is not always practical in this RAM-constrained environment.

PC Connectivity

PC connectivity is an integral component of the Palm OS device. The device comes with a cradle that connects to a desktop PC and with software for the PC that provides "one-button" backup and synchronization of all data on the device with the user's PC.

Because all user data can be backed up on the PC, replacement of the nonvolatile storage area of the Palm OS device becomes a simple matter of installing the new module in place of the old one and resynchronizing with the PC. The format of the user's data in storage RAM can change with a new version of the ROM; the connectivity software on the PC is responsible for translating the data into the correct format when downloading it onto a device with a new ROM.

Memory Architecture

WARNING: This section describes the current (June 1998) implementation of Palm OS memory architecture. This implementation may change as the Palm OS evolves. Do not rely on implementation-specific information described here; instead, always use the API provided to manipulate memory.

The Palm OS system software is designed around a 32-bit architecture. The system uses 32-bit addresses, and its basic data types are 8, 16, and 32 bits long.

The 32-bit addresses available to software provide a total of 4 GB of address space for storing code and data. This address space affords a large growth potential for future revisions of both the hardware and software without affecting the execution model. Although a large memory space is available, Palm OS was designed to work efficiently with small amounts of RAM. For example, the first commercial Palm OS device has less than 1 MB of memory, or .025% of this address space.

The Motorola 68328 processor's 32-bit registers and 32 internal address lines support a 32-bit execution model as well, although the external data bus is only 16 bits wide. This design reduces cost without impacting the software model. The processor's bus controller automatically breaks down 32-bit reads and writes into multiple 16-bit reads and writes externally.

Each memory card in the Palm OS device has 256 MB of address space reserved for it. Memory card 0 starts at address $1000000, memory card 1 starts at address $2000000, and so on.

The Palm OS divides the total available RAM store into two logical areas: **dynamic** RAM and **storage** RAM. Dynamic RAM is used as working space for temporary allocations, and is analogous to the RAM installed in a typical desktop system. The remainder of the available RAM on the card is designated as storage RAM and is analogous to disk storage on a typical desktop system.

Because power is always applied to the memory system, both areas of RAM preserve their contents when the device is turned "off" (i.e., is in low-power sleep mode.) See "Palm OS Power Modes" in Chapter 6, "Using Palm OS Managers," of "Developing Palm OS Applications, Part 1." All of storage memory is preserved even when the device is reset explicitly. As part of the boot sequence, the system software reinitializes the dynamic area, and leaves the storage area intact.

The entire dynamic area of RAM is used to implement a single heap that provides memory for dynamic allocations. From this **dynamic heap**, the system provides memory for dynamic data such as global variables, system dynamic allocations (TCP/IP, IrDA, and so on, as applicable), application stacks, temporary memory allocations, and application dynamic allocations (such as those performed when the application calls the `MemHandleNew` function.)

The entire amount of RAM reserved for the dynamic heap is always dedicated to this use, regardless of whether it is actually used for allocations. The size of the dynamic area of RAM on a particular device varies according to the OS version running, the amount of physical RAM available, and the requirements of pre-installed software such as the TCP/IP stack or IrDA stack. Table 1.1 on page 25 provides more information about the dynamic heap space that currently available combinations of OS and hardware provide.

Table 1.1 Dynamic heap space

RAM Usage	OS 3.0 > 1 MB TCP/IP & IrDA (Palm III)	OS 2.0 1 MB TCP/IP only (Professional)	OS 2.0/1.0 512 KB no TCP/IP or IrDA (Personal)
Total dynamic area	96 KB	64 KB	32 KB
System Globals (screen buffer, UI globals, database references, etc.)	~2.5 KB	~2.5 KB	~2.5 KB
TCP / IP stack	32 KB	32 KB	0 KB
System dynamic allocation (IrDA, "Find" window, temporary allocations)	variable amount	~15 KB (no IrDA in this OS)	~15 KB
Application stack (call stack and local vars)	4 KB (default)	2.5 KB	2.5 KB
Remaining dynamic space (dynamic allocations, application global variables, and static variables)	≤ 36 KB	≤ 12 KB	≤ 12 KB

The remaining portion of RAM not dedicated to the dynamic heap is configured as one or more **storage heaps** used to hold nonvolatile user data such as appointments, to do lists, memos, address lists, and so on. An application accesses a storage heap by calling the database manager or resource manager, according to whether it needs to manipulate user data or resources.

The size and number of storage heaps available on a particular device varies according to the OS version that is running; the amount of physical RAM that is available; and the storage requirements of end-user application software such as the Address List, Date Book, or third-party applications.

Versions 1.0 and 2.0 of Palm OS subdivide storage RAM into multiple storage heaps of 64 KB each. Palm OS 3.0 configures all storage RAM on a card as a single storage heap. Under all versions of Palm OS, system overhead limits the maximum usable data storage available in a single chunk to slightly less than 64 KB.

In the Palm OS environment, all data are stored in memory manager chunks. A **chunk** is an area of contiguous memory between 1 byte and slightly less than 64 KB in size that has been allocated by the Palm OS memory manager. (Because system overhead requirements may vary, an exact figure for the maximum amount of usable data storage for all chunks cannot be specified.) Currently, all Palm OS implementations limit the maximum size of any chunk to slightly less than 64 KB; however, the API does not have this constraint, and it may be relaxed in the future.

Each chunk resides in a heap. Some heaps are ROM-based and contain only nonmovable chunks; some are RAM-based and may contain movable or nonmovable chunks. A RAM-based heap may be a dynamic heap or a storage heap. The Palm OS memory manager allocates memory in the dynamic heap (for dynamic allocations, stacks, global variables, and so on). The Palm OS data manager allocates memory in one or more storage heaps (for nonvolatile user data.)

Every memory chunk used to hold storage data (as opposed to memory chunks that store dynamic data) is a **record** in a database implemented by the Palm OS data manager. In the Palm OS environment, a **database** is simply a list of memory chunks and associated database header information. Normally, the items in a database share some logical association; for example, a database may hold a collection of all address book entries, all datebook entries, and so on.

A database is analogous to a file in a desktop system. Just as a traditional file system can create, delete, open, and close files, Palm OS applications can create, delete, open, and close databases as necessary. There is no restriction on where the records for a particular database reside as long as they are all on the same memory card. The records from one database can be interspersed with the records from one or more other databases in memory.

Storing data by database fits nicely with the Palm OS memory manager design. Each record in a database is in fact a memory manager chunk. The data manager can use memory manager calls to allocate, delete, and resize database records. All heaps except for the dynamic heap are nonvolatile, so database records can be stored in any heap except the dynamic heap. Because records can be stored anywhere on the memory card, databases can be distributed over multiple discontiguous areas of physical RAM, but all records belonging to a particular database must reside on the same card.

To understand how database records are manipulated, it helps to know something about the way the memory manager allocates and tracks memory chunks, as the next section describes.

Heap Overview

WARNING: This section describes the current (June 1998) implementation of Palm OS memory architecture. This implementation may change as the Palm OS evolves. Do not rely on implementation-specific information described here; instead, always use the API provided to manipulate memory.

Recall that a **heap** is a contiguous area of memory used to contain and manage one or more smaller chunks of memory. When applications work with memory (allocate, resize, lock, etc.) they usually work with chunks of memory. An application can specify whether to allocate a new chunk of memory in the storage heap or the dynamic heap. The memory manager manages each heap independently and rearranges chunks as necessary to defragment heaps and merge free space.

Heaps in the Palm OS environment are referenced through heap IDs. A **heap ID** is a unique 16-bit value that the memory manager uses to identify a heap within the Palm OS address space. Heap IDs start at 0 and increment sequentially by units of 1. Values are assigned beginning with the RAM heaps on card 0, continuing with the ROM heaps on card 0, and then continuing through RAM and ROM heaps on subsequent cards. The sequence of heap IDs is continuous; that is, no values in the sequence are skipped.

The first heap (heap 0) on card 0 is the dynamic heap. This heap is reinitialized every time the Palm OS device is reset. When an application quits, the system frees any chunks allocated by that application in the dynamic heap. All other heaps are nonvolatile storage heaps that retain their contents through soft reset cycles.

When a Palm OS device is presented with multiple dynamic heaps, the first heap (heap 0) on card 0 is the active dynamic heap. All other potential dynamic heaps are ignored. For example, it is possible that a future Palm OS device supporting multiple cards might be presented with two cards, each having its own dynamic heap; if so, only the dynamic heap residing on card 0 would be active—the system would not treat any heaps on other cards as dynamic heaps, nor would heap IDs be assigned to these heaps. Subsequent storage heaps would be assigned IDs in sequential order, as always beginning with RAM heaps, followed by ROM heaps.

Overview of Memory Chunk Structure

Memory chunks can be movable or nonmovable. Applications need to store data in movable chunks whenever feasible, thereby enabling the memory manager to move chunks as necessary to create contiguous free space in memory for allocation requests.

When the memory manager allocates a nonmovable chunk it returns a pointer to that chunk. The pointer is simply that chunk's address in memory. Because the chunk cannot move, its pointer remains valid for the chunk's lifetime; thus, the pointer can be passed "as is" to the caller that requested the allocation.

When the memory manager allocates a moveable chunk, it generates a pointer to that chunk, just as it did for the nonmovable chunk, but it does not return the pointer to the caller. Instead, it stores the pointer to the chunk, called the **master chunk pointer**, in a **master pointer table** that is used to track all of the moveable chunks in the heap, and returns a reference to the master chunk pointer. This reference to the master chunk pointer is known as a **handle**. It is this handle that the memory manager returns to the caller that requested the allocation of a moveable chunk.

Using handles imposes a slight performance penalty over direct pointer access but permits the memory manager to move chunks around in the heap without invalidating any chunk references that an application might have stored away. As long as an application uses handles to reference data, only the master pointer to a chunk needs to be updated by the memory manager when it moves a chunk during defragmentation.

An application typically locks a chunk handle for a short time while it has to read or manipulate the contents of the chunk. The process of locking a chunk tells the memory manager to mark that data chunk as immobile. When an application no longer needs the data chunk, it should unlock the handle immediately to keep heap fragmentation to a minimum.

Note that any handle is good only until the system is reset. When the system resets, it reinitializes all dynamic memory areas and relaunches applications. Therefore, you must not store a handle in a database record or a resource.

Each chunk on a memory card is actually located by means of a card–relative reference called a **local ID**. A local ID is a reference to a data chunk that the system computes from the base address of the card. The local ID of a nonmovable chunk is simply the offset of the chunk from the base address of the card. The local ID of a movable chunk is the offset of the master pointer to the chunk from the base address of the card, but with the low-order bit set. Since chunks are always aligned on word boundaries, only local IDs of movable chunks have the low-order bit set. Once the base address of the card is determined at runtime, a local ID can be converted quickly to a pointer or handle.

For example, when an application needs the handle to a particular data record, it calls the data manager to retrieve the record by index from the appropriate database. The data manager fetches the local ID of the record out of the database header and uses it to compute the handle to the record. The handle to the record is never actually stored in the database itself.

Although currently available Palm OS devices do not provide hardware support for multiple cards, the use of local IDs provides support in software for future devices that may allow the user to remove or insert memory cards. If the user moves a memory card to a slot having a different base address, the handle to a memory chunk on that card is likely to change, but the local ID associated with that chunk does not change.

The Memory Manager

The Palm OS memory manager is responsible for maintaining the location and size of every memory chunk in nonvolatile storage, volatile storage, and ROM. It provides an API for allocating new chunks, disposing of chunks, resizing chunks, locking and unlocking chunks, and compacting heaps when they become fragmented. Because of the limited RAM and processor resources of the Palm OS device, the memory manager is efficient in its use of processing power and memory.

This section provides background information on the organization of memory in Palm OS and provides an overview of the memory manager API, discussing these topics:

- Memory Manager Structures
- Using the Memory Manager
- Memory Manager Function Summary

Memory Manager Structures

This section discusses the different structures the memory manager uses:

- Heap Structures
- Chunk Structures
- Local ID Structures

Heap Structures

WARNING: Expect the heap structure to change in the future. Use the API to work with heaps.

A heap consists of the heap header, master pointer table, and the heap chunks.

- **Heap header**. The heap header is located at the beginning of the heap. It holds the size of the heap and contains flags for the heap that provide certain information to the memory manager; for example, whether the heap is ROM-based.

- **Master pointer table**. Following the heap header is a master pointer table. It is used to store 32-bit pointers to movable chunks in the heap.
  - When the memory manager moves a chunk to compact the heap, the pointer for that chunk in the master pointer table is updated to the chunk's new location. The handles an application uses to track movable chunks reference the address of the master pointer to the chunk, not the chunk itself. In this way, handles remain valid even after a chunk is moved. The OS compacts the heap automatically when available contiguous space is not sufficient to fulfill an allocation request.
  - If the master pointer table becomes full, another is allocated and its offset is stored in the `nextMstrPtrTable` field of the previous master pointer table. Any number of master pointer tables can be linked in this way. Because additional master pointer chunks are nonmovable, they are allocated at the end of the heap, according to the guidelines described in the "Heap chunks" section following immediately.
- **Heap chunks**. Following the master pointer table are the actual chunks in the heap.
  - Movable chunks are generally allocated at the beginning of the heap, and nonmovable chunks at the end of the heap.
  - Nonmovable chunks do not need an entry in the master pointer table since they are never relocated by the memory manager.
  - Applications can easily walk the heap by hopping from chunk to chunk because each chunk header contains the size of the chunk. All free and nonmovable chunks can be found in this manner by checking the flags in each chunk header.

Because heaps can be ROM-based, there is no information in the header that must be changed when using a heap. Also, ROM-based heaps contain only nonmovable chunks and have a master pointer table with 0 entries.

Chunk Structures

WARNING: Expect the chunk structure to change in the future. Use the API to work with chunks.

Each chunk begins with an 8-byte header followed by that chunk's data. The chunk header consists of a `Flags:size` adjustment byte, 3 bytes of size information, a `lock:owner` byte, and 3 bytes of `hOffset` information.

- **`Flags:sizeAdj` byte.** This byte contains flags in the high nibble and a size adjustment in the low nibble.
  - The flags nibble has 1 bit currently defined, which is set for free chunks.
  - The size adjustment nibble can be used to calculate the requested size of the chunk, given the actual size. The requested size is computed by taking the size as stored in the chunk header and subtracting the size of the header and the size adjustment field. The actual size of a chunk is always a multiple of two so that chunks always start on a word boundary.
- **`size` field (3 bytes)**. This three-byte value describes the size of the chunk, which is **larger** than the size requested by the application and includes the size of the chunk header itself. The maximum data size for a chunk is slightly less than 64 KB.
- **`Lock:owner` byte**. Following the size information is a byte that holds the lock count in the high nibble and the owner ID in the low nibble.
  - The lock count is incremented every time a chunk is locked and decremented every time a chunk is unlocked. A movable chunk can be locked a maximum of 14 times before being unlocked. Nonmovable chunks always have 15 in the lock field.
  - The owner ID determines the owner of a memory chunk and is set by the memory manager when allocating a new chunk. Owner ID is information is useful for debugging and for garbage collection when an application terminates abnormally.
- **`hOffset` field (3 bytes)**. The last three bytes in the chunk header is the distance from the master pointer for the chunk to the chunk's header, divided by two. Note that this offset could be a negative value if the master pointer table is at a higher

address than the chunk itself. For nonmovable chunks that do not need an entry in the master pointer table, this field is 0.

Local ID Structures

WARNING: Expect the local ID structure to change in the future. Use the API to work with chunks.

Chunks that contain database records or other database information are tracked by the data manager through local IDs. A local ID is card relative and is always valid no matter what memory slot the card resides in. A local ID can be easily converted to a pointer or the handle to a chunk once the base address of the card is known.

The upper 31 bits of a local ID contain the offset of the chunk or master pointer to the chunk from the beginning of the card. The low-order bit is set for local IDs of handles and clear for local IDs of pointers.

The `MemLocalIDToGlobal` function converts a local ID and card number (either 0 or 1) to a pointer or handle. It looks at the card number and adds the appropriate card base address to convert the local ID to a pointer or handle for that card.

Using the Memory Manager

Use the memory manager API to allocate memory in the dynamic heap (for dynamic allocations, stacks, global variables, and so on) and use the data manager API to allocate memory in one or more storage heaps (for user data.) The data manager calls the memory manager as appropriate to perform low-level allocations. (See "The Data Manager" on page 37 for more information.)

Overview of the Memory Manager API

To allocate a movable chunk, call `MemHandleNew` and pass the desired chunk size. Before you can read or write data to this chunk, you must call `MemHandleLock` to lock it and get a pointer to it. Every time you lock a chunk, its lock count is incremented. You can lock a chunk a maximum of 14 times before an error is returned. (Recall that unmovable chunks hold the value15 in the lock field.) `MemHandleUnlock` reverses the effect of `MemHandleLock`—it decrements the value of the lock field by 1. When the lock count is reduced to 0, the chunk is free to be moved by the memory manager.

When an application allocates memory in the dynamic heap, the memory manager uses an owner ID to associate that chunk with the application. The system further distinguishes chunks belonging to the currently active allocation by setting a special bit in the ownerID information. When the application quits, all chunks in the dynamic heap having this bit set are freed automatically.

If the system needs to allocate a chunk that is not disposed of when an application quits, it changes the chunk's owner ID to 0 by calling the system function `MemHandleSetOwner`. This function is not used by applications, except in special circumstances. For example, when passing a parameter block to an application that is being launched, the owner of the block must be set to the system; otherwise, when the application exits, the system deletes the block when it frees all memory allocated by the application.

To determine the size of a movable chunk, pass its handle to `MemHandleSize`. To resize it, call `MemHandleResize`. You generally cannot increase the size of a chunk if it's locked unless there happens to be free space in the heap immediately following the chunk. If the chunk is unlocked, the memory manager is allowed to move it to another area of the heap to increase its size.When you no longer need the chunk, call `MemHandleFree`, which releases the chunk even if it is locked.

If you have a pointer to a locked, movable chunk, you can recover the handle by calling `MemPtrRecoverHandle`. In fact, all of the `MemPtr`*Xxx* calls, including `MemPtrSize`, also work on pointers to locked, movable chunks.

To allocate a nonmovable chunk, call `MemPtrNew` and pass the desired size of the chunk. This call returns a pointer to the chunk, which can be used directly to read or write to it.

To determine the size of a nonmovable chunk, call `MemPtrSize`. To resize it, call `MemPtrResize`. You generally can't increase the size of a nonmovable chunk unless there is free space in the heap immediately following the chunk. When you no longer need the chunk, call `MemPtrFree`, which releases the chunk even if it's locked.

Use the memory manager utility routines `MemMove` and `MemSet` to move memory from one place to another or to fill memory with a specific value.

In most situations, the proper way to free memory is by calling one of the `MemPtrFree` or `MemHandleFree` functions.

Important For important cautions and practical advice regarding the proper use of memory on Palm OS devices, be sure to read "Writing Robust Code," on page 37 of Part I, *Interface Management*.

Storage Heap Sizes and Memory Management Schemes

In Palm OS version 1.0, individual storage heaps were limited to a maximum size of 64 KB each and the memory manager moved objects automatically among multiple storage heaps to prevent any of them from becoming too full. This strategy tended to decrease the availability of contiguous space for large objects. The version 2.0 memory manager abandoned this approach, increasing the availability of contiguous heap space; however, it still limited the maximum size of individual heaps to 64 KB each. Palm OS version 3.0 removes the 64 KB maximum size restriction on individual heaps and creates just two heaps: one 96K dynamic heap and one storage heap that is the size of all remaining RAM on the card.

Optimizing Memory Manager Performance

Because Palm OS applications must perform well in a RAM-constrained environment, proper code segmentation is critical to achieving optimum performance.

If your application segments are too large, your application may not perform well (or to run at all) when large contiguous blocks of memory are not available. Conversely, if your application segments are too small, performance may be hindered by the overhead required to find and load resources too frequently.

Unfortunately, it impossible to specify a single size for memory chunks that will perform optimally for all applications.You will need to experiment with segmenting your code in different ways while measuring your application's performance in order to discover the size and arrangement of resource chunks that will optimize your particular application's responsiveness and overall performance. Both the Palm OS Debugger and the Simulator provide tools for examining the internal structure of heaps, viewing the amount of free space available, manipulating blocks, and so on.

Memory Manager Function Summary

The following functions are available for application use:

- `MemCardInfo`
- `MemChunkFree`
- `MemDebugMode`
- `MemHandleDataStorage`
- `MemHandleCardNo`
- `MemHandleFree`
- `MemHandleHeapID`
- `MemHandleLock`
- `MemHandleNew`
- `MemHandleResize`
- `MemHandleSize`
- `MemHandleToLocalID`
- `MemHandleUnlock`
- `MemHeapCheck`
- `MemHeapCompact`
- `MemHeapDynamic`
- `MemHeapFlags`
- `MemHeapFreeBytes`
- `MemHeapID`
- `MemHeapScramble`
- `MemHeapSize`
- `MemLocalIDKind`
- `MemLocalIDToGlobal`
- `MemLocalIDToLockedPtr`
- `MemLocalIDToPtr`
- `MemMove`

- `MemNumCards`
- `MemNumHeaps`
- `MemNumRAMHeaps`
- `MemPtrCardNo`
- `MemPtrDataStorage`
- `MemPtrFree`
- `MemPtrHeapID`
- `MemPtrToLocalID`
- `MemPtrNew`
- `MemPtrRecoverHandle`
- `MemPtrResize`
- `MemSet`
- `MemSetDebugMode`
- `MemPtrSize`
- `MemPtrUnlock`
- `MemStoreInfo`

The Data Manager

A traditional file system first reads all or a portion of a file into a memory buffer from disk, using and/or updating the information in the memory buffer, and then writes the updated memory buffer back to disk. Because Palm OS devices have limited amounts of dynamic RAM and use nonvolatile RAM instead of disk storage, a traditional file system is not optimal for storing and retrieving Palm OS user data.

Palm OS accesses and updates all information in place. This works well because it reduces dynamic memory requirements and eliminates the overhead of transferring the data to and from another memory buffer involved in a file system.

As a further enhancement, data in the Palm OS device is broken down into multiple, finite-size **records** that can be left scattered throughout the memory space; thus, adding, deleting, or resizing a record does not require moving other records around in memory. Each record in a database is in fact a memory manager chunk. The data manager uses memory manager functions to allocate, delete, and resize database records.

This section explains how to use the database manager by discussing these topics:

- Records and Databases
- Structure of a Database Header
- Using the Data Manager

Records and Databases

Databases organize related records; every record belongs to one and only one database. A database may be a collection of all address book entries, all datebook entries, and so on. A Palm OS application can create, delete, open, and close databases as necessary, just as a traditional file system can create, delete, open, and close a traditional file. There is no restriction on where the records for a particular database reside as long as they all reside on the same memory card. The records from one database can be interspersed with the records from one or more other databases in memory.

Storing data by database fits nicely with the Palm OS memory manager design. All heaps except for the dynamic heap(s) are nonvolatile, so database records can be stored in any heap except the dynamic heap(s) (see Heap Overview). Because records can be stored anywhere on the memory card, databases can be distributed over multiple discontiguous areas of physical RAM.

Accessing Data With Local IDs

A database maintains a list of all records that belong to it by storing the local ID of each record in the database header. Because local IDs are used, the memory card can be placed into any memory slot of a Palm OS device. An application finds a particular record in a database by index. When an application requests a particular record, the data manager fetches the local ID of the record from the database header by index, converts the local ID to a handle using the card number that contains the database header, and returns the handle to the record.

Structure of a Database Header

A database header consists of some basic database information and a list of records in the database. Each record entry in the header has the local ID of the record, 8 attribute bits, and a 3-byte unique ID for the record.

This section provides information about database headers, discussing these topics:

- Database Header Fields
- Structure of a Record Entry in a Database Header.

WARNING: Expect the database header structure to change in the future. Use the API to work with database structures.

Database Header Fields

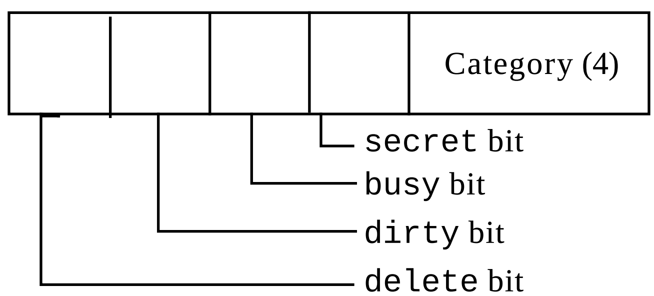
The database header has the following fields:

- The `name` field holds the name of the database.
- The `attributes` field has flags for the database.
- The `version` field holds an application-specific version number for that database.
- The `modificationNumber` is incremented every time a record in the database is deleted, added, or modified. Thus applications can quickly determine if a shared database has been modified by another process.
- The `appInfoID` is an optional field that an application can use to store application-specific information about the database. For example, it might be used to store user display preferences for a particular database.
- The `sortInfoID` is another optional field an application can use for storing the local ID of a sort table for the database.
- The `type` and `creator` fields are each 4 bytes and hold the database type and creator. The system uses these fields to distinguish application databases from data databases and to associate data databases with the appropriate application. See “The System Manager” in Chapter 6, “Using Palm OS Managers,” of “Developing Palm OS Applications, Part I” for more information.

- The `numRecords` field holds the number of record entries stored in the database header itself. If all the record entries cannot fit in the header, then `nextRecordList` has the local ID of a `recordList` that contains the next set of records.

  Each record entry stored in a record list has three fields and is 8 bytes in length. Each entry has the local ID of the record which takes up 4 bytes: 1 byte of attributes and a 3-byte unique ID for the record. The `attribute` field, shown in Figure 1.1, is 8 bits long and contains 4 flags and a 4-bit category number. The category number is used to place records into user-defined categories like "business" or "personal."

Figure 1.1 Record Attributes



Structure of a Record Entry in a Database Header

Each record entry has the local ID of the record, 8 attribute bits, and a 3-byte unique ID for the record.

- Local IDs make the database slot-independent. Since all records for a database reside on the same memory card as the header, the handle of any record in the database can be quickly calculated. When an application requests a specific record from a database, the data manager returns a handle to the record that it determines from the stored local ID.

  A special situation occurs with ROM-based databases. Because ROM-based heaps use nonmovable chunks exclusively, the local IDs to records in a ROM-based database are local IDs of pointers, not handles. So, when an application opens a ROM-based database, the data manager allocates and initializes a fake handle for each record and returns the appropriate fake handle when the application requests a record. Because of this, applications can use handles to access both RAM- and ROM-based database records.

- The unique ID must be unique for each record within a database. It remains the same for a particular record no matter how many times the record is modified. It is used during synchronization with the desktop to track records on the Palm OS device with the same records on the desktop system.

When the user deletes or archives a record on Palm OS:

- The `delete` bit is set in the `attributes` flags, but its entry in the database header remains until the next synchronization with the PC.
- The `dirty` bit is set whenever a record is updated.
- The `busy` bit is set when an application currently has a record locked for reading or writing.
- The `secret` bit is set for records that should not be displayed before the user password has been entered on the device.

When a user “deletes” a record on the Palm OS device, the record’s data chunk is freed, the local ID stored in the record entry is set to 0, and the `delete` bit is set in the attributes. When the user archives a record, the deleted bit is also set but the chunk is not freed and the local ID is preserved. This way, the next time the user synchronizes with the desktop system, the desktop can quickly determine which records to delete (since their record entries are still around on the Palm OS device). In the case of archived records, the desktop can save the record data on the PC before it permanently removes the record entry and data from the Palm OS device. For deleted records, the PC just has to delete the same record from the PC before permanently removing the record entry from the Palm OS device.

Using the Data Manager

Using the data manager is similar to using a traditional file manager, except that the data is broken down into multiple records instead of being stored in one contiguous chunk. To create or delete a database, call `DmCreateDatabase` and `DmDeleteDatabase`.

Each memory card is akin to a disk drive and can contain multiple databases. To open a database for reading or writing, you must first get the database ID, which is simply the local ID of the database header. Calling DmFindDatabase searches a particular memory card for a database by name and returns the local ID of the database header. Alternatively, calling DmGetDatabase returns the database ID for each database on a card by index.

After determining the database ID, you can open the database for read-only or read / write access. When you open a database, the system locks down the database header and returns a reference to a database access structure, which tracks information about the open database and caches certain information for optimum performance. The database access structure is a relatively small structure (less than 100 bytes) allocated in the dynamic heap that is disposed of when the database is closed.

Call DmDatabaseInfo, DmSetDatabaseInfo, and DmDatabaseSize to query or set information about a database, such as its name, size, creation and modification dates, attributes, type, and creator.

Call DmGetRecord, DmQueryRecord, and DmReleaseRecord when viewing or updating a database.

- DmGetRecord takes a record index as a parameter, marks the record busy, and returns a handle to the record. If a record is already busy when DmGetRecord is called, an error is returned.
- DmQueryRecord is faster if the application only needs to view the record; it doesn't check or set the busy bit, so it's not necessary to call `DmReleaseRecord` when finished viewing the record.
- DmReleaseRecord clears the `busy` bit, and updates the modification number of the database and marks the record dirty if the `dirty` parameter is true.

To resize a record to grow or shrink its contents, call DmResizeRecord. This routine automatically reallocates the record in another heap of the same card if the current heap does not have enough space for it. Note that if the data manager needs to move the record into another heap to resize it, the handle to the record changes. DmResizeRecord returns the new handle to the record.

To add a new record to a database, call DmNewRecord. This routine can insert the new record at any index position, append it to the end, or replace an existing record by index. It returns a handle to the new record.

There are three methods for removing a record: `DmRemoveRecord`, `DmDeleteRecord`, and `DmArchiveRecord`.

- `DmRemoveRecord` removes the record's entry from the database header and disposes of the record data.
- `DmDeleteRecord` also disposes of the record data, but instead of removing the record's entry from the database header, it sets the deleted bit in the record entry attributes field and clears the local chunk ID.
- `DmArchiveRecord` does not dispose of the record's data; it just sets the deleted bit in the record entry.

Both `DmDeleteRecord` and `DmArchiveRecord` are useful for synchronizing information with a desktop PC. Since the unique ID of the deleted or archived record is still kept in the database header, the desktop PC can perform the necessary operations on its own copy of the database before permanently removing the record from the Palm OS database.

Call `DmRecordInfo` and `DmSetRecordInfo` to retrieve or set the record information stored in the database header, such as the attributes, unique ID, and local ID of the record. Typically, these routines are used to set or retrieve the category of a record that is stored in the lower four bits of the record's attribute field.

To move records from one index to another or from one database to another, call `DmMoveRecord`, `DmAttachRecord`, and `DmDetachRecord`. `DmDetachRecord` removes a record entry from the database header and returns the record handle. Given the handle of a new record, `DmAttachRecord` inserts or appends that new record to a database or replaces an existing record with the new record. `DmMoveRecord` is an optimized way to move a record from one index to another in the same database.

Data Manager Function Summary

- `DmAttachRecord`
- `DmArchiveRecord`
- `DmCloseDatabase`
- `DmCreateDatabase`

- DmCreateDatabaseFromImage
- DmDatabaseInfo
- DmDatabaseSize
- DmDeleteDatabase
- DmDeleteRecord
- DmDetachRecord
- DmFindDatabase
- DmFindRecordByID
- DmFindSortPositionV10
- DmGetAppInfoID
- DmGetDatabase
- DmGetLastErr
- DmGetNextDatabaseByTypeCreator
- DmGetRecord
- DmInsertionSort
- DmMoveCategory
- DmMoveRecord
- DmNewHandle
- DmNewRecord
- DmNextOpenDatabase
- DmNumDatabases
- DmNumRecords
- DmNumRecordsInCategory
- DmOpenDatabase
- DmOpenDatabaseInfo
- DmOpenDatabaseByTypeCreator
- DmPositionInCategory
- DmQueryNextInCategory
- DmQueryRecord
- DmQuickSort
- DmRecordInfo
- DmReleaseRecord
- DmRemoveRecord

- DmRemoveSecretRecords
- DmResetRecordStates
- DmResizeRecord
- DmSearchRecord
- DmSeekRecordInCategory
- DmSet
- DmSetDatabaseInfo
- DmSetRecordInfo
- DmStrCopy
- DmWrite
- DmWriteCheck

The Resource Manager

Applications can use the resource manager much like the data manager to retrieve and save chunks of data conveniently. The resource manager has the added capability of tagging each chunk of data with a unique resource type and resource ID. These tagged data chunks, called **resources**, are stored in resource databases. Resource databases are almost identical in structure to normal databases except for a slight amount of increased storage overhead per resource record (two extra bytes). In fact, the resource manager is nothing more than a subset of routines in the data manager that are broken out here for conceptual reasons only.

Resources are typically used to store the user interface elements of an application, such as images, fonts, dialog layouts, and so forth. Part of building an application involves creating these resources and merging them with the actual executable code. In the Palm OS environment, an application is, in fact, simply a resource database with the executable code stored as one or more code resources and the graphics elements and other miscellaneous data stored in the same database as other resource types.

Applications may also find the resource manager useful for storing and retrieving application preferences, saved window positions, state information, and so forth. These preferences settings can be stored in a separate resource database.

This section explains how to work with the resource manager and discusses these topics:

- Structure of a Resource Database Header
- Using the Resource Manager
- Resource Manager Functions

Structure of a Resource Database Header

A resource database header consists of some general database information followed by a list of resources in the database. The first portion of the header is identical in structure to a normal database header. Resource database headers are distinguished from normal database headers by the `dmHdrAttrResDB` bit in the `attributes` field.

WARNING: Expect the resource database header structure to change in the future. Use the API to work with resource database structures.

- The `name` field holds the name of the resource database.
- The `attributes` field has flags for the database and always has the `dmHdrAttrResDB` bit set.
- The `modificationNumber` is incremented every time a resource in the database is deleted, added, or modified. Thus, applications can quickly determine if a shared resource database has been modified by another process.
- The `appInfoID` and `sortInfoID` fields are not normally needed for a resource database but are included to match the structure of a regular database. An application may optionally use these fields for its own purposes.
- The `type` and `creator` fields hold 4-byte signatures of the database `type` and `creator` as defined by the application that created the database.
- The `numResources` field holds the number of resource info entries that are stored in the header itself. In most cases, this is the total number of resources. If all the resource info entries cannot fit in the header, however, then `nextResourceList` has the `chunkID` of a `resourceList` that contains the next set of resource info entries.

Each 10-byte resource info entry in the header has the resource type, the resource ID, and the local ID of the memory manager chunk that contains the resource data.

Using the Resource Manager

You can create, delete, open, and close resource databases with the routines used to create normal record-based databases (see Using the Data Manager). This includes all database-level (not record-level) routines in the data manager such as `DmCreateDatabase`, `DmDeleteDatabase`, `DmDatabaseInfo`, and so on.

When you create a new database using `DmCreateDatabase`, the type of database created (record or resource) depends on the value of the `resDB` parameter. If set, a resource database is created and the `dmHdrAttrResDB` bit is set in the `attributes` field of the database header. Given a database header ID, an application can determine which type of database it is by calling `DmDatabaseInfo` and examining the `dmHdrAttrResDB` bit in the returned `attributes` field.

Once a resource database has been opened, an application can read and manipulate its resources by using the resource-based access routines of the resource manager. Generally, applications use the `DmGetResource` and `DmReleaseResource` routines.

`DmGetResource` returns a handle to a resource, given the type and ID. This routine searches all open resource databases for a resource of the given type and ID, and returns a handle to it. The search starts with the most recently opened database. To search only the most recently opened resource database for a resource instead of all open resource databases, call `DmGet1Resource`.

`DmReleaseResource` should be called as soon as an application finishes reading or writing the resource data. To resize a resource, call `DmResizeResource`, which accepts a handle to a resource and reallocates the resource in another heap of the same card if necessary. It returns the handle of the resource, which might have been changed if the resource had to be moved to another heap to be resized.

The remaining resource manager routines are usually not required for most applications. These include functions to get and set resource attributes, move resources from one database to another, get resources by index, and create new resources. Most of these functions reference resources by index to optimize performance. When referencing a resource by index, the DmOpenRef of the open resource database that the resource belongs to must also be specified. Call DmSearchResource to find a resource by type and ID or by pointer by searching in all open resource databases.

To get the DmOpenRef of the topmost open resource database, call DmNextOpenResDatabase and pass nil as the current DmOpenRef. To find out the DmOpenRef of each successive database, call DmNextOpenResDatabase repeatedly with each successive DmOpenRef.

Given the access pointer of a specific open resource database, DmFindResource can be used to return the index of a resource, given its type and ID. DmFindResourceType can be used to get the index of every resource of a given type. To get a resource handle by index, call DmGetResourceIndex.

To determine how many resources are in a given database, call DmNumResources. To get and set attributes of a resource including its type and ID, call DmResourceInfo and DmSetResourceInfo. To attach an existing data chunk to a resource database as a new resource, call DmAttachResource. To detach a resource from a database, call DmDetachResource.

To create a new resource, call DmNewResource and pass the desired size, type, and ID of the new resource. To delete a resource, call DmRemoveResource. Removing a resource disposes of its data chunk and removes its entry from the database header.

Resource Manager Functions

To work with resources, you can use the functions listed in Data Manager Function Summary as well as these functions:

- DmAttachResource
- DmDatabaseProtect
- DmDetachResource
- DmDeleteCategory
- DmFindResource

- DmFindResourceType
- DmFindSortPosition
- DmGetResource
- DmGetResourceIndex
- DmGet1Resource
- DmNewResource
- DmNextOpenResDatabase
- DmNumResources
- DmReleaseResource
- DmRemoveResource
- DmResizeResource
- DmSearchResource
- DmSetResourceInfo

2

Memory Management Functions

Memory Manager Functions

MemCardInfo

Purpose Return information about a memory card.

Prototype

```
Err MemCardInfo (UInt cardNo,
                 CharPtr cardNameP,
                 CharPtr manufNamP,
                 UIntPtr versionP,
                 ULongPtr crDateP,
                 ULongPtr romSizeP,
                 ULongPTr ramSizeP,
                 ULongPtr freeBytesP)
```

Parameters

cardNo	Card number.
cardNameP	Pointer to character array (32 bytes), or 0.
manufNameP	Pointer to character array (32 bytes), or 0.
versionP	Pointer to version variable, or 0.
crDateP	Pointer to creation date variable, or 0.
romSizeP	Pointer to ROM size variable, or 0.
ramSizeP	Pointer to RAM size variable, or 0.
freeBytesP	Pointer to free byte-count variable, or 0.

Result Returns 0 if no error.

Comments Pass 0 for those variables that you don’t want returned.

MemCmp

Purpose Compare two blocks of memory.

Prototype

```
Int MemCmp (VoidPtr s1,
            VoidPtr s2,
            ULong numBytes)
```

Parameters

`s1, s2`	Pointers to block of memory.
`numBytes`	Number of bytes to compare.

Result Zero if they match, non-zero if not.

+ if s1 > s2

- if s1 < s2

MemDebugMode

Purpose Return the current debugging mode of the memory manager.

Prototype

```
Word MemDebugMode (void)
```

Parameters No parameters.

Result Returns debug flags as described for `MemSetDebugMode`.

MemHandleCardNo

Purpose Return the card number a chunk resides in.

Prototype `UInt MemHandleCardNo (VoidHand h)`

Parameters -> h Chunk handle.

Result Returns the card number.

Comments Call this routine to retrieve the card number (0 or 1) a movable chunk resides on.

See Also MemPtrCardNo

MemHandleDataStorage

Purpose Return TRUE if the given handle is part of a data storage heap. If not, it's a handle in the dynamic heap.

Prototype `Boolean MemHandleDataStorage (VoidHand h)`

Parameters -> h Chunk handle.

Result Returns TRUE if the handle is part of a data storage heap.

Comments Called by Fields package routines to determine if they need to worry about data storage write-protection when editing a text field.

See Also MemPtrDataStorage

MemHandleFree

Purpose Dispose of a movable chunk.

Prototype `Err MemHandleFree (VoidHand h)`

Parameters -> `h` Chunk handle.

Result: Returns 0 if no error, or `memErrInvalidParam` if an error occurs.

Comments Call this routine to dispose of a movable chunk.

See Also `MemHandleNew`

MemHandleHeapID

Purpose Return the heap ID of a chunk.

Prototype `UInt MemHandleHeapID (VoidHand h)`

Parameters -> `h` Chunk handle.

Result Returns the heap ID of a chunk.

Comments Call this routine to get the heap ID of the heap a chunk resides in.

See Also `MemPtrHeapID`

MemHandleLock

Purpose Lock a chunk and obtain a pointer to the chunk's data.

Prototype `VoidPtr MemHandleLock (VoidHand h)`

Parameters -> `h` Chunk handle.

Result Returns a pointer to the chunk.

Comments Call this routine to lock a chunk and obtain a pointer to the chunk.

`MemHandleLock` and `MemHandleUnlock` should be used in pairs.

See Also `MemHandleNew`, `MemHandleUnlock`

MemHandleNew

Purpose Allocate a new movable chunk in the dynamic heap and returns a handle to it.

Prototype `VoidHand MemHandleNew (ULong size)`

Parameters -> `size` The desired size of the chunk.

Result Returns a handle to the new chunk, or 0 if unsuccessful.

Comments Use this call to allocate dynamic memory. Before you can write data to the memory chunk that `MemHandleNew` allocates, you must call `MemHandleLock` to lock the chunk and get a pointer to it.

See Also `MemPtrFree`, `MemPtrNew`, `MemHandleFree`, `MemHandleLock`

MemHandleResize

Purpose Resize a chunk.

Prototype `Err MemHandleResize (VoidHandle h, ULong newSize)`

Parameters

-> `h`	Chunk handle.
-> `newSize`	The new desired size.

Result

0	No error.
`memErrInvalidParam`	Invalid parameter passed.
`memErrNotEnoughSpace`	Not enough free space in heap to grow chunk.
`memErrChunkLocked`	Can't grow chunk because it's locked.

Comments Call this routine to resize a chunk. This routine is always successful when shrinking the size of a chunk, even if the chunk is locked. When growing a chunk, it first attempts to grab free space immediately following the chunk so that the chunk does not have to move. If the chunk has to move to another free area of the heap to grow, it must be movable and have a lock count of 0.

On devices running version 2.0 or earlier of Palm OS, the `MemHandleResize` function tries to resize the chunk only within the same heap, whereas `DmResizeRecord` will look for space in other data heaps if it can't find enough space in the original heap.

See Also `MemHandleNew`, `MemHandleSize`

MemHandleSize

Purpose Return the requested size of a chunk.

Prototype `ULong MemHandleSize (VoidHand h)`

Parameters -> h Chunk handle.

Result Returns the requested size of the chunk.

Comments Call this routine to get the size originally requested for a chunk.

See Also `MemHandleResize`

MemHandleToLocalID

Purpose Convert a handle into a local chunk ID which is card relative.

Prototype `LocalID MemHandleToLocalID (VoidHand h)`

Parameters -> h Chunk handle.

Result Returns local ID, or nil (0) if unsuccessful.

Comments Call this routine to convert a chunk handle to a local ID.

See Also `MemLocalIDToGlobal`, `MemLocalIDToLockedPtr`

MemHandleUnlock

Purpose Unlock a chunk given a chunk handle.

Prototype `Err MemHandleUnlock (VoidHand h)`

Parameters `-> h` The chunk handle.

Result

0	No error.
`memErrInvalidParam`	Invalid parameter passed.

Comments Call this routine to decrement the lock count for a chunk.

`MemHandleLock` and `MemHandleUnlock` should be used in pairs.

See Also `MemHandleLock`

MemHeapCheck

Purpose Check validity of a given heap.

Prototype `Err MemHeapCheck (UInt heapID)`

Parameters `heapID` ID of heap to check.

Result Returns 0 if no error.

See Also `MemDebugMode`, `MemSetDebugMode`

MemHeapCompact

Purpose Compact a heap.

Prototype `Err MemHeapCompact (UInt heapID)`

Parameters -> `heapID` ID of the heap to compact.

Result Always returns 0.

Comments Most applications never need to call this function explicitly. The system software calls this function at various times; for example, during memory allocation (if sufficient free space is not available) and during system reboot.

Call this routine to compact a heap and merge all free space. This routine attempts to move all movable chunks to the start of the heap and merge all free space in the center of the heap.

MemHeapDynamic

Purpose Return TRUE if the given heap is a dynamic heap.

Prototype `Boolean MemHeapDynamic (UInt heapID)`

Parameters `heapID` ID of the heap to be tested.

Result Returns TRUE if dynamic, FALSE if not.

Comments Dynamic heaps are used for volatile storage, application stacks, globals, and dynamically allocated memory.

See Also `MemNumHeaps`, `MemHeapID`

MemHeapFlags

Purpose Return the heap flags for a heap.

Prototype

```
UInt MemHeapFlags (UInt heapID)
```

Parameters

-> `heapID`	ID of heap.

Result Returns the heap flags.

Comments Call this routine to retrieve the heap flags for a heap. The flags can be examined to determine if the heap is ROM based or not. ROM-based heaps have the `memHeapFlagReadOnly` bit set.

See Also `MemNumHeaps`, `MemHeapID`

MemHeapFreeBytes

Purpose Return the total number of free bytes in a heap and the size of the largest free chunk in the heap.

Prototype

```
Err MemHeapFreeBytes ( UInt heapID,
                       ULongPtr freeP,
                       ULongPtr maxP)
```

Parameters

-> `heapID`	ID of heap.
<-> `freeP`	Pointer to a variable of type `ULong` for free bytes.
<-> `maxP`	Pointer to a variable of type `ULong` for max free chunk size.

Result Always returns 0.

Comments This routine doesn't compact the heap but may be used to determine in advance whether an allocation request will succeed. Before allocating memory, call this function and test the return value of `maxP` to determine whether enough free space to fulfill your allocation request exists. If not, you may make more space available by calling the `MemHeapCompact` function. An alternative approach is to just call the `MemHeapCompact` function as necessary when an error is returned by the `MemPtrNew` or `MemHandleNew` functions.

See Also `MemHeapSize`, `MemHeapID`, `MemHeapCompact`

MemHeapID

Purpose Return the heap ID for a heap, given its index and the card number.

Prototype `UInt MemHeapID (UInt cardNo, UInt heapIndex)`

Parameters

-> `cardNo`	The card number, either 0 or 1.
-> `heapIndex`	The heap index, anywhere from 0 to `MemNumHeaps` - 1.

Result Returns the heap ID.

Comments Call this routine to retrieve the heap ID of a heap, given the heap index and the card number. A heap ID must be used to obtain information on a heap such as its size, free bytes, etc., and is also passed to any routines which manipulate heaps.

See Also `MemNumHeaps`

MemHeapScramble

Purpose Scramble the specified heap.

Prototype `Err MemHeapScramble (UInt heapID)`

Parameters `heapID` ID of heap to scramble.

Comments The system attempts to move each movable chunk.

Useful for debugging.

Result Always returns 0.

See Also `MemDebugMode`, `MemSetDebugMode`

MemHeapSize

Purpose Return the total size of a heap including the heap header.

Prototype `ULong MemHeapSize (UInt heapID)`

Parameters -> `heapID` ID of heap.

Result Returns the total size of the heap.

See Also `MemHeapFreeBytes`, `MemHeapID`

MemLocalIDKind

Purpose Return whether or not a local ID references a handle or a pointer.

Prototype `LocalIDKind MemLocalIDKind (LocalID local)`

Parameters -> `local` Local ID to query

Result Returns `LocalIDKind`, or a `memIDHandle` or `memIDPtr` (see MemoryMgr.h).

Comments This routine determines if the given local ID is to a nonmovable (`memIDPtr`) or movable (`memIDHandle`) chunk.

MemLocalIDToGlobal

Purpose Convert a local ID, which is card relative, into a global pointer in the designated card.

Prototype

```
VoidPtr MemLocalIDToGlobal ( LocalID local,
                             UInt cardNo)
```

Parameters

-> `local` The local ID to convert.

-> `cardNo` Memory card the chunk resides in.

Result Returns pointer or handle to chunk.

See Also `MemLocalIDKind`, `MemLocalIDToLockedPtr`

MemLocalIDToLockedPtr

Purpose Return a pointer to a chunk given its local ID and card number.

Note: If the local ID references a movable chunk handle, this routine automatically locks the chunk before returning.

Prototype

```
VoidPtr MemLocalIDToLockedPtr(LocalID local,
                              UInt cardNo)
```

Parameters

`local`	Local chunk ID.
`cardNo`	Card number.

Result Returns pointer to chunk, or 0 if an error occurs.

See Also `MemLocalIDToGlobal`, `MemLocalIDToPtr`, `MemLocalIDKind`, `MemPtrToLocalID`, `MemHandleToLocalID`

MemLocalIDToPtr

Purpose Return pointer to chunk, given the local ID and card number.

Prototype

```
VoidPtr MemLocalIDToPtr( LocalID local,
                         UInt cardNo)
```

Parameters

-> `local`	Local ID to query.
-> `cardNo`	Card number the chunk resides in.

Result Returns a pointer to the chunk, or 0 if error.

Comments If the local ID references a movable chunk and that chunk is **not** locked, this function returns 0 to indicate an error.

See Also `MemLocalIDToGlobal`, `MemLocalIDToLockedPtr`

MemMove

Purpose Move a range of memory to another range.

Prototype

```
Err MemMove(VoidPtr dstP,
            VoidPtr srcP,
            ULong numBytes)
```

Parameters

`dstP`	Pointer to destination.
`srcP`	Pointer to source.
`numBytes`	Number of bytes to move.

Result Always returns 0.

Comments Handles overlapping ranges.

For operations where the destination is in a data heap, see `DmSet`, `DmWrite`, and related functions.

MemNumCards

Purpose Return the number of memory card slots in the system. Not all slots need to be populated.

Prototype

```
UInt MemNumCards (void)
```

Parameters None.

Result Returns number of slots in the system.

MemNumHeaps

Purpose Return the number of heaps available on a particular card.

Prototype `UInt MemNumHeaps (UInt cardNo)`

Parameters -> `cardNo` The card number; either 0 or 1.

Result Number of heaps available, including ROM- and RAM-based heaps.

Comments Call this routine to retrieve the total number of heaps on a memory card. The information can be obtained by calling `MemHeapSize`, `MemHeapFreeBytes`, and `MemHeapFlags` on each heap using its heap ID. The heap ID is obtained by calling `MemHeapID` with the card number and the heap index, which can be any value from 0 to `MemNumHeaps`.

MemNumRAMHeaps

Purpose Return the number of RAM heaps in the given card.

Prototype `UInt MemNumRAMHeaps (UInt cardNo)`

Parameters `cardNo` The card number.

Result Returns the number of RAM heaps.

See Also `MemNumCards`

MemPtrCardNo

Purpose Return the card number (0 or 1) a nonmovable chunk resides on.

Prototype `UInt MemPtrCardNo (VoidPtr chunkP)`

Parameters -> `chunkP` Pointer to the chunk.

Result Returns the card number.

See Also `MemHandleCardNo`

MemPtrDataStorage

Purpose Return TRUE if the given pointer is part of a data storage heap; if not, it is a pointer in the dynamic heap.

Prototype `Boolean MemPtrDataStorage (VoidPtr p)`

Parameters `p` Pointer to a chunk.

Result Returns TRUE if the chunk is part of a data storage heap.

Comments Called by Fields package to determine if it needs to worry about data storage write-protection when editing a text field.

See Also `MemHeapDynamic`

MemPtrFree

Purpose Macro to dispose of a chunk.

Prototype `Err MemPtrFree (VoidPtr p)`

Parameters -> `p` Pointer to a chunk.

Result 0 If no error or `memErrInvalidParam` (invalid parameter).

Comments Call this routine to dispose of a nonmovable chunk.

MemPtrHeapID

Purpose Return the heap ID of a chunk.

Prototype `UInt MemPtrHeapID (VoidPtr p)`

Parameters -> `p` Pointer to the chunk.

Result Returns the heap ID of a chunk.

Comments Call this routine to get the heap ID of the heap a chunk resides in.

MemPtrNew

Purpose Allocate a new nonmovable chunk in the dynamic heap.

Prototype `VoidPtr MemPtrNew (ULong size)`

Parameters -> `size` The desired size of the chunk.

Result Returns pointer to the new chunk, or 0 if unsuccessful.

Comments This routine allocates a nonmovable chunk in the dynamic heap and returns a pointer to the chunk. Applications can use it when allocating dynamic memory.

MemPtrRecoverHandle

Purpose Recover the handle of a movable chunk, given a pointer to its data.

Prototype `VoidHand MemPtrRecoverHandle (VoidPtr p)`

Parameters -> `p` Pointer to the chunk.

Result Returns the handle of the chunk, or 0 if unsuccessful.

Comments Don't call this function for pointers in ROM or nonmovable data chunks.

MemPtrResize

Purpose Resize a chunk.

Prototype `Err MemPtrResize (VoidPtr p, ULong newSize)`

Parameters -> `p` Pointer to the chunk.

-> `newSize` The new desired size.

Result Returns 0 if no error, or `memErrNotEnoughSpace` `memErrInvalidParam`, or `memErrChunkLocked` if an error occurs.

Comments Call this routine to resize a locked chunk. This routine is always successful when shrinking the size of a chunk. When growing a chunk, it attempts to use free space immediately following the chunk.

See Also `MemPtrSize`, `MemHandleResize`

MemPtrSize

Purpose Return the size of a chunk.

Prototype `ULong MemPtrSize (VoidPtr p)`

Parameters -> `p` Pointer to the chunk.

Result The requested size of the chunk.

Comments Call this routine to get the original requested size of a chunk.

MemPtrToLocalID

Purpose Convert a pointer into a card-relative local chunk ID.

Prototype `LocalID MemPtrToLocalID (VoidPtr chunkP)`

Parameters -> `chunkP` Pointer to a chunk.

Result Returns the local ID of the chunk.

Comments Call this routine to convert a chunk pointer to a local ID.

See Also `MemLocalIDToPtr`

MemPtrUnlock

Purpose Unlock a chunk, given a pointer to the chunk.

Prototype `Err MemPtrUnlock (VoidPtr p)`

Parameters `p` Pointer to a chunk.

Result 0 if no error, or `memErrInvalidParam` if an error occurs.

Comments A chunk must **not** be unlocked more times than it was locked.

See Also `MemHandleLock`

MemSet

Purpose Set a memory range in a dynamic heap to a specific value.

Prototype

```
Err MemSet (VoidPtr dstP,
            ULong numBytes,
            Byte value)
```

Parameters

`dstP`	Pointer to the destination.
`numBytes`	Number of bytes to set.
`value`	Value to set.

Result Always returns 0.

Comments For operations where the destination is in a data heap, see `DmSet`, `DmWrite`, and related functions.

MemSetDebugMode

Purpose Set the debugging mode of the memory manager.

Prototype

```
Err MemSetDebugMode (Word flags)
```

Parameters

`flags`	Debug flags.

Comments Use the logical OR operator (|) to provide any combination of one, more, or none of the following flags:

`memDebugModeCheckOnChange`

`memDebugModeCheckOnAll`

`memDebugModeScrambleOnChange`

`memDebugModeScrambleOnAll`

`memDebugModeFillFree`

`memDebugModeAllHeaps`

`memDebugModeRecordMinDynHeapFree`

Result Returns 0 if no error, or -1 if an error occurs.

MemStoreInfo

Purpose Return information on either the RAM store or the ROM store for a memory card.

Prototype

```
Err MemStoreInfo ( UInt cardNo,
                   UInt storeNumber,
                   UIntPtr versionP,
                   UIntPtr flagsP,
                   CharPtr nameP,
                   ULongPtr crDateP,
                   ULongPtr bckUpDateP,
                   ULongPtr heapListOffsetP,
                   ULongPtr initCodeOffset1P,
                   ULongPtr initCodeOffset2P,
                   LocalID* databaseDirIDP)
```

Parameters

-> `cardNo`	Card number, either 0 or 1.
-> `storeNumber`	Store number; 0 for ROM, 1 for RAM.
<-> `versionP`	Pointer to version variable, or 0.
<-> `flagsP`	Pointer to flags variable, or 0.
<-> `nameP`	Pointer to character array (32 bytes), or 0.
<-> `crDateP`	Pointer to creation date variable, or 0.
<-> `bckUpDateP`	Pointer to backup date variable, or 0.
<-> `heapListOffsetP`	Pointer to `heapListOffset` variable, or 0.
<-> `initCodeOffset1P`	Pointer to `initCodeOffset1` variable, or 0.
<-> `initCodeOffset2P`	Pointer to `initCodeOffset2` variable, or 0.

<-> `databaseDirIDP`
Pointer to database directory chunk ID variable, or 0.

Result Returns 0 if no error, or `memErrCardNoPresent`, `memErrRAMOnlyCard`, or `memErrInvalidStoreHeader` if an error occurs.

Comments Call this routine to retrieve any or all information on either the RAM store or the ROM store for a card. Pass 0 for variables that you don't wish returned.

Functions for System Use Only

MemCardFormat

Prototype

```
Err MemCardFormat (UInt cardNo,
                   CharPtr cardNameP,
                   CharPtr manufNameP,
                   CharPtr ramStoreNameP)
```

WARNING: This function for use by system software only.

MemChunkFree

Prototype

```
Err MemChunkFree (VoidPtr chunkDataP)
```

WARNING: This function for use by system software only.

MemChunkNew

Prototype

```
VoidPtr MemChunkNew ( UInt heapID,
                      ULong size,
                      UInt attributes)
```

WARNING: This function for use by system software only.

MemHandleFlags

Prototype

```
UInt MemHandleFlags (VoidHand h)
```

WARNING: This function for use by system software only.

MemHandleLockCount

Prototype

```
UInt MemHandleLockCount (VoidHand h)
```

WARNING: This function for use by system software only.

MemHandleOwner

Prototype

```
UInt MemHandleOwner (VoidHand h)
```

WARNING: This function for use by system software only.

MemHandleResetLock

Prototype

```
Err MemHandleResetLock (VoidHand h)
```

WARNING: This function for use by system software only.

MemHandleSetOwner

Prototype

```
Err MemHandleSetOwner (VoidHand h, UInt owner)
```

WARNING: This function for use by system software only.

MemHeapFreeByOwnerID

Prototype

```
Err MemHeapFreeByOwnerID ( UInt heapID,
                           UInt ownerID)
```

WARNING: This function for use by system software only.

MemHeapInit

Prototype

```
Err MemHeapInit( UInt heapID,
                 Int numHandles,
                 Boolean initContents)
```

WARNING: This function for use by system software only.

MemInit

Prototype

```
Err MemInit (void)
```

WARNING: This function for use by system software only.

MemInitHeapTable

Prototype

```
Err MemInitHeapTable (UInt cardNo)
```

WARNING: This function for use by system software only.

MemKernelInit

Prototype

```
Err MemKernelInit(void)
```

WARNING: This function for use by system software only.

MemPtrFlags

Prototype

```
UInt MemPtrFlags (VoidPtr chunkDataP)
```

WARNING: This function for use by system software only.

MemPtrOwner

Prototype `UInt MemPtrOwner (VoidPtr chunkDataP)`

WARNING: This function for use by system software only.

MemPtrResetLock

Prototype `Err MemPtrResetLock (VoidPtr chunkP)`

WARNING: This function for use by system software only.

MemPtrSetOwner

Prototype `Err MemPtrSetOwner (VoidPtr chunkP, UInt owner)`

WARNING: This function for use by system software only.

MemSemaphoreRelease

Prototype `Err MemSemaphoreRelease (Boolean writeAccess)`

WARNING: This function for use by system software only.

MemSemaphoreReserve

Prototype `Err MemSemaphoreReserve (Boolean writeAccess)`

WARMING: This function for use by system software only.

MemStoreSetInfo

Prototype

```
Err MemStoreSetInfo (UInt cardNo,
                     UInt storeNumber,
                     UIntPtr versionP,
                     UIntPtr flagsP,
                     CharPtr nameP,
                     ULongPtr crDateP,
                     ULongPtr bckUpDateP,
                     ULongPtr heapListOffsetP,
                     ULongPtr initCodeOffset1P,
                     ULongPtr initCodeOffset2P,
                     LocalID* databaseDirIDP)
```

WARNING: This function for use by system software only.

3

Data and Resource Manager Functions

Data Manager Functions

DmArchiveRecord

Purpose Mark a record as archived by leaving the record's chunk around and setting the delete bit for the next sync.

Prototype `Err DmArchiveRecord (DmOpenRef dbR, UInt index)`

Parameters

-> `dbR`	`DmOpenRef` to open database.
-> `index`	Which record to archive.

Result Returns 0 if no error or `dmErrIndexOutOfRange` or `dmErrReadOnly` if an error occurs.

Comments Marks the delete bit in the database header for the record but does not dispose of the record's data chunk.

See Also `DmRemoveRecord`, `DmDetachRecord`, `DmNewRecord`, `DmDeleteRecord`

DmAttachRecord

Purpose Attach an existing chunk ID handle to a database as a record.

Prototype

```
Err DmAttachRecord (DmOpenRef dbR,
                    UIntPtr atP,
                    Handle newH,
                    Handle* oldHP)
```

Parameters

-> `dbR`	`DmOpenRef` to open database.
<-> `atP`	Pointer to index where new record should be placed.
-> `newH`	Handle of new record.
<-> `oldHP`	Pointer to return old handle if replacing existing record.

Result Returns 0 if no error, or `dmErrIndexOutOfRange`, `dmErrMemError`, `dmErrReadOnly`, `dmErrRecordInWrongCard`, `memErrChunkLocked`, `memErrInvalidParam`, or `memErrNotEnoughSpace` if an error occurs.

Comments Given the handle of an existing chunk, this routine makes that chunk a new record in a database and sets the dirty bit. The parameter `atP` points to an index variable. If `oldHP` is `NIL`, the new record is inserted at index `*atP` and all record indices that follow are shifted down. If `*atP` is greater than the number of records currently in the database, the new record is appended to the end and its index is returned in `*atP`. If `oldHP` is not `NIL`, the new record replaces an existing record at index `*atP` and the handle of the old record is returned in `*oldHP` so that the application can free it or attach it to another database.

Useful for cutting and pasting between databases.

See Also `DmDetachRecord`, `DmNewRecord`, `DmNewHandle`

DmAttachResource

Purpose Attach an existing chunk ID to a resource database as a new resource.

Prototype

```
Err DmAttachResource (DmOpenRef dbR,
                      VoidHand newH,
                      ULong resType,
                      Int resID)
```

Parameters

-> `dbR`	`DmOpenRef` to open database.
-> `newH`	Handle of new resource's data.
-> `resType`	Type of the new resource.
-> `resID`	ID of the new resource.

Result Returns 0 if no error, or `dmErrIndexOutOfRange`, `dmErrMemError`, `dmErrReadOnly`, `dmErrRecordInWrongCard`, `memErrChunkLocked`, `memErrInvalidParam`, or `memErrNotEnoughSpace` if an error occurs.

Comments Given the handle of an existing chunk with resource data in it, this routine makes that chunk a new resource in a resource database. The new resource will have the given type and ID.

See Also `DmDetachResource`, `DmRemoveResource`, `DmNewHandle`, `DmNewResource`

DmCloseDatabase

Purpose Close a database.

Prototype `Err DmCloseDatabase (DmOpenRef dbR)`

Parameters

`dbR`	Database access pointer.

Result Returns 0 if no error or `dmErrInvalidParam` if an error occurs.

Comments This routine doesn't unlock any records in the database which have been left locked, so the application should be careful not to leave records locked. When performance is not an issue, call `DmResetRecordStates` before closing the database in order to unlock all records and clear the busy bits.

See Also `DmOpenDatabase`, `DmDeleteDatabase`, `DmOpenDatabaseByTypeCreator`

DmCreateDatabase

Purpose Create a new database on the specified card with the given name, creator, and type.

Prototype

```
Err DmCreateDatabase (UInt cardNo,
                      CharPtr nameP,
                      ULong creator,
                      ULong type,
                      Boolean resDB)
```

Parameters

-> `cardNo`	The card number to create the database on.
-> `nameP`	Name of new database, up to 31 ASCII bytes long.
-> `creator`	Creator of the database.

-> `type`	Type of the database.
-> `resDB`	If `TRUE`, create a resource database.

Result Returns 0 if no error, or `dmErrInvalidDatabaseName`, `dmErrAlreadyExists`, `memErrCardNotPresent`, `dmErrMemError`, `memErrChunkLocked`, `memErrInvalidParam`, `memErrInvalidStoreHeader`, `memErrNotEnoughSpace`, or `memErrRAMOnlyCard` if an error occurs.

Comments Call this routine to create a new database on a specific card. If another database with the same name already exists in RAM store, this routine returns a `dmErrAlreadyExists` error code. Once created, the database ID can be retrieved by calling `DmFindDatabase` and the database opened using the database ID. To create a resource database instead of a record-based database, set the `resDB` Boolean to `TRUE`.

See Also `DmCreateDatabaseFromImage`, `DmOpenDatabase`, `DmDeleteDatabase`

DmCreateDatabaseFromImage

Purpose Call to create an entire database from a single resource that contains an image of the database; usually, make this call from an application's reset action code during boot.

Prototype

```
Err DMCreateDatabaseFromImage (Ptr bufferP)
```

Parameters

`bufferP`	Pointer to locked resource containing database image.

Result Returns 0 if no error

Comments Use this function to create the default database for an application.

See Also `DmCreateDatabase`, `DmOpenDatabase`

DmDatabaseInfo

Purpose Retrieve information about a database.

Prototype

```
Err DmDatabaseInfo (
          UInt cardNo, LocalID dbID,
          CharPtr nameP, UIntPtr attributesP,
          UIntPtr versionP, ULongPtr crDateP,
          ULongPtr modDateP, ULongPtr bckUpDateP,
          ULongPtr modNumP, LocalID* appInfoIDP,
          LocalID* sortInfoIDP, ULongPtr typeP,
          ULongPtr creatorP)
```

Parameters

-> `cardNo`	Number of card database resides on.
-> `dbID`	Database ID of the database.
<-> `nameP`	Pointer to 32-byte character array for returning the name, or `NIL`.
<-> `attributesP`	Pointer to return attributes variable, or `NIL`.
`versionP`	Pointer to new version, or `NIL`.
<-> `crDateP`	Pointer to return creation date variable, or `NIL`.
<-> `modDateP`	Pointer to return modification date variable, or `NIL`.
<-> `bckUpDateP`	Pointer to return backup date variable, or `NIL`.
<-> `modNumP`	Pointer to return modification number variable, or `NIL`.
<-> `appInfoIDP`	Pointer to return `appInfoID` variable, or `NIL`.
<-> `sortInfoIDP`	Pointer to return `sortInfoID` variable, or `NIL`.
<-> `typeP`	Pointer to return type variable, or `NIL`.
<-> `creatorP`	Pointer to return creator variable, or `NIL`.

Result Returns 0 if no error, or `dmErrInvalidParam` if an error occurs.

Comments Call this routine to retrieve any or all information about a database. This routine accepts `NIL` for any return variable parameter pointer you don't want returned.

See Also DmSetDatabaseInfo, DmDatabaseSize, DmOpenDatabaseInfo, DmFindDatabase, DmGetNextDatabaseByTypeCreator

DmDatabaseProtect

Purpose

This routine can be used to prevent a database from being deleted (by passing TRUE for 'protect'). It increments the protect count if protect is TRUE and decrements it if protect is FALSE.

Use this function if you want to keep a particular record or resource in a database locked down but don't want to keep the database open. This information is kept in the dynamic heap so all databases are "unprotected" at system reset.

Prototype

```
Err DmDatabaseProtect (UInt cardNo,
                       LocalID dbID,
                       Boolean protect)
```

Parameters

cardNo	Card number of database to protect/unprotect.
dbID	Local ID of database to protect/unprotect.
protect	If TRUE, protect count will be incremented. If FALSE, protect count will be decremented.

Result Zero if successful.

DmDatabaseSize

Purpose Retrieve size information on a database.

Prototype

```
Err DmDatabaseSize (UInt cardNo,
                    ChunkID dbID,
                    ULongPtr numRecordsP,
                    ULongPtr totalBytesP,
                    ULongPtr dataBytesP)
```

Parameters

-> `cardNo`	Card number database resides on.
-> `dbID`	Database ID of the database.
<-> `numRecordsP`	Pointer to return `numRecords` variable, or `NIL`.
<-> `totalBytesP`	Pointer to return `totalBytes` variable, or `NIL`.
<-> `dataBytesP`	Pointer to return `dataBytes` variable, or `NIL`.

Result Returns 0 if no error, or `dmErrMemError` if an error occurs.

Comments Call this routine to retrieve the size of a database. Any of the return data variable pointers can be `NIL`.

- The total number of records is returned in `*numRecordsP`.
- The total number of bytes used by the database including the overhead is returned in `*totalBytesP`.
- The total number of bytes used to store just each record's data, not including overhead, is returned in `*dataBytesP`.

See Also `DmDatabaseInfo`, `DmOpenDatabaseInfo`, `DmFindDatabase`, `DmGetNextDatabaseByTypeCreator`

DmDeleteCategory

Purpose Delete all records in a category. The category name is not changed.

Prototype

```
Err DmDeleteCategory (DmOpenRef dbR,
                      UInt categoryNum)
```

Parameters

`dbR`	Database access pointer.
`categoryNum`	Category of records to delete.

Result Zero if there is no error, an error code otherwise.

DmDeleteDatabase

Purpose Delete a database and all its records.

Prototype

```
Err DmDeleteDatabase (UInt cardNo, LocalID dbID)
```

Parameters

--> `cardNo`	Card number the database resides on.
--> `dbID`	Database ID.

Result Returns 0 if no error, or `dmErrCantFind`, `dmErrCantOpen`, `memErrChunkLocked`, `dmErrDatabaseOpen`, `dmErrROMBased`, `memErrInvalidParam`, or `memErrNotEnoughSpace` if an error occurs.

Comments Call this routine to delete a database. This routine accepts a database ID as a parameter. To determine the database ID, call either `DmFindDatabase` or `DmGetDatabase` with a database index.

See Also `DmDeleteRecord`, `DmRemoveRecord`, `DmRemoveResource`, `DmCreateDatabase`, `DmGetNextDatabaseByTypeCreator`, `DmFindDatabase`

DmDeleteRecord

Purpose Delete a record's chunk from a database but leave the record entry in the header and set the `delete` bit for the next sync.

Prototype `Err DmDeleteRecord (DmOpenRef dbR, UInt index)`

Parameters

-> `dbR`	`DmOpenRef` to open database.
-> `index`	Which record to delete.

Result Returns 0 if no error, or `dmErrIndexOutOfRange`, `dmErrReadOnly`, or `memErrInvalidParam` if an error occurs.

Comments Marks the `delete` bit in the database header for the record and disposes of the record's data chunk. Does not remove the record entry from the database header, but simply sets the `localChunkID` of the record entry to `NIL`.

See Also `DmDetachRecord`, `DmRemoveRecord`, `DmArchiveRecord`, `DmNewRecord`

DmDetachRecord

Purpose

Detach and orphan a record from a database but don't delete the record's chunk.

Prototype

```
Err DmDetachRecord ( DmOpenRef dbR,
                     UInt index,
                     Handle* oldHP)
```

Parameters

-> `dbR`	`DmOpenRef` to open.
-> `index`	Index of the record to detach.
<-> `oldHP`	Pointer to return handle of the detached record.

Result

Returns 0 if no error or `dmErrReadOnly` (database is marked read only), `dmErrIndexOutOfRange` (index out of range), `memErrChunkLocked`, `memErrInvalidParam`, or `memErrNotEnoughSpace` if an error occurs.

Comments

This routine detaches a record from a database by removing its entry from the database header and returns the handle of the record's data chunk in `*oldHP`. Unlike `DmDeleteRecord`, this routine removes any traces of the record, including its entry in the database header.

See Also

`DmAttachRecord`, `DmRemoveRecord`, `DmArchiveRecord`, `DmDeleteRecord`

DmDetachResource

Purpose Detach a resource from a database and return the handle of the resource's data.

Prototype

```
Err DmDetachResource ( DmOpenRef dbR,
                       Int index,
                       VoidHand* oldHP)
```

Parameters

-> dbR	DmOpenRef to open database.
-> index	Index of resource to detach.
<-> oldHP	Pointer to return handle of the detached record.

Result Returns 0 if no error, or dmErrCorruptDatabase, dmErrIndexOutOfRange, dmErrReadOnly, memErrChunkLocked, memErrInvalidParam, or memErrNotEnoughSpace if an error occurs.

Comments This routine detaches a resource from a database by removing its entry from the database header and returns the handle of the resource's data chunk in *oldHP.

See Also DmAttachResource, DmRemoveResource

DmFindDatabase

Purpose Return the database ID of a database by card number and name.

Prototype

```
LocalID DmFindDatabase ( UInt cardNo,
                         CharPtr nameP)
```

Parameters

-> cardNo	Number of card to search.
-> nameP	Name of the database to look for.

Result Returns the database ID, or 0 if not found.

See Also DmGetNextDatabaseByTypeCreator, DmDatabaseInfo, DmOpenDatabase

DmFindRecordByID

Purpose Return the index of the record with the given unique ID.

Prototype

```
Err DmFindRecordByID ( DmOpenRef dbR,
                       ULong uniqueID,
                       UIntPtr indexP)
```

Parameters

dbR	Database access pointer.
uniqueID	Unique ID to search for.
indexP	Return index.

Result Returns 0 if found, otherwise `dmErrUniqueIDNotFound`.

See Also DmQueryRecord, DmGetRecord, DmRecordInfo

DmFindResource

Purpose Search the given database for a resource by type and ID, or by pointer if it is non-`NIL`.

Prototype

```
Int DmFindResource (DmOpenRef dbR,
                    ULong resType,
                    Int resID,
                    VoidHand findResH)
```

Parameters

-> `dbR`	Open resource database access pointer.
-> `resType`	Type of resource to search for.
-> `resID`	ID of resource to search for.
->`findResH`	Pointer to locked resource, or `NIL`.

Result Returns index of resource in resource database, or -1 if not found.

Comments Use this routine to find a resource in a particular resource database by type and ID or by pointer. It is particularly useful when you want to search only one database for a resource and that database is not the topmost one.

If `findResH` is `NIL`, the resource is searched for by type and ID.

If `findResH` is not `NIL`, `resType` and `resID` are ignored and the index of the given locked resource is returned.

Once the index of a resource is determined, it can be locked down and accessed by calling `DmGetResourceIndex`.

See Also `DmGetResource`, `DmSearchResource`, `DmResourceInfo`, `DmGetResourceIndex`, `DmFindResourceType`

DmFindResourceType

Purpose Search the given database for a resource by type and type index.

Prototype

```
Int DmFindResourceType (DmOpenRef dbR,
                        ULong resType,
                        Int typeIndex)
```

Parameters

-> `dbR`	Open resource database access pointer.
-> `resType`	Type of resource to search for.
-> `typeIndex`	Index of given resource type.

Result Index of resource in resource database, or -1 if not found.

Comments Use this routine to retrieve all the resources of a given type in a resource database. By starting at `typeIndex` 0 and incrementing until an error is returned, the total number of resources of a given type and the index of each of these resources can be determined. Once the index of a resource is determined, it can be locked down and accessed by calling `DmGetResourceIndex`.

See Also `DmGetResource`, `DmSearchResource`, `DmResourceInfo`, `DmGetResourceIndex`, `DmFindResource`

DmFindSortPosition

Purpose Return to where a record is or should be. Useful to find where to insert a record. Uses a binary search.

Prototype

```
UInt DmFindSortPosition (DmOpenRef dbR,
                 VoidPtr newRecord,
                 SortRecordInfoPtr newRecordInfo,
                 DmComparF *compar,
                 Int other)
```

Parameters

`dbR`	Database access pointer.
`newRecord`	Pointer to the new record.
`newRecordInfo`	Information about the new record.
`compar`	Pointer to comparison.
`other`	Other info for comparison.

Result The position where the record should be inserted.

The position should be viewed as between the record returned and the record before it. Note that the return value may be one greater than the number of records.

Caveat If there are deleted records in the database, `DmFindSortPosition` only works if those records are at the end of the database. `DmFindSortPosition` always assumes that a deleted record is greater than or equal to any other record.

See Also `DmFindSortPositionV10`

DmFindSortPositionV10

Purpose

Return to where a record is or should be.

Useful to find an existing record or find where to insert a record. Uses a binary search.

Prototype

```
UInt DmFindSortPositionV10 (DmOpenRef dbR,
                            VoidPtr newRecord,
                            DmComparF *compar,
                            Int other)
```

Parameters

`dbR`	Database access pointer.
`newRecord`	Pointer to the new record.
`compar`	Comparison function (see Comments).
`other`	Any value the application wants to pass to the comparison function.

Result

Returns the position where the record should be inserted. The position should be viewed as between the record returned and the record before it. Note that the return value may be one greater than the number of records.

Comments

The comparison function, `compar`, accepts two arguments, `elem1` and `elem2`, each a pointer to an entry in the table. The comparison function compares each of the pointed-to items (`*elem1` and `*elem2`), and returns an integer based on the result of the comparison.

If the items...	`compar` returns...
`*elem1 < *elem2`	an integer < 0
`*elem1 == *elem2`	0
`*elem1 > *elem2`	an integer > 0

2.0 Note

DmComparF has changed; it previously had three parameters but now has six. DmComparF is the typedef of a callback used by SysInsertionSort, DmInsertionSort, and FindInsertPosition.

The new compar parameters allow a Palm OS application to pass more information to the system than before, most noticeably the record (and all associated information) which allows sorting by unique ID, so that the Palm OS device and the desktop always match.

The revised callback is used by new sorting routines (and can be used the same way by your application):

```
typedef Int DmComparF (void *,
  void *,
  Int other,
  SortRecordInfoPtr,
  SortRecordInfoPtr,
  VoidHand appInfoH);
```

As a rule, the change in the number of arguments from three to six doesn't cause problems when a 1.0 application is run on a 2.0 device, because the system only pulls the arguments from the stack that are there.

Keep in mind, however, that some optimized applications built with tools other than Metrowerks CodeWarrior for Palm OS may have problems as a result of the change in arguments when running on a 2.0 or later device.

See Also DmFindSortPosition, DmQuickSort, DmInsertionSort

DmGetAppInfoID

Purpose Return the local ID of the application info block.

Prototype `LocalID DmGetAppInfoID (DmOpenRef dbR).`

Parameters

`dbR`	Database access pointer.

Result Returns local ID of the application info block

See Also `DmDatabaseInfo`, `DmOpenDatabase`

DmGetDatabase

Purpose Return the database header ID of a database by index and card number.

Prototype `LocalID DmGetDatabase (UInt cardNo, UInt index)`

Parameters

-> `cardNo`	Card number of database.
-> `index`	Index of database.

Result Returns the database ID, or 0 if an invalid parameter passed.

Comments Call this routine to retrieve the database ID of a database by index. The index should range from 0 to `DmNumDatabases`-1. This routine is useful for getting a directory of all databases on a card.

See Also `DmOpenDatabase`, `DmNumDatabases`, `DmDatabaseInfo`, `DmDatabaseSize`

DmGetLastErr

Purpose Return error code from last data manager call.

Prototype `Err DmGetLastErr (void)`

Parameters None.

Result Error code from last unsuccessful data manager call.

Comments Use this routine to determine why a data manager call failed. In particular, calls like `DmGetRecord` return 0 only if unsuccessful, so calling `DmGetLastErr` is the only way to determine why they failed.

Note that `DmGetLastErr` does not always reflect the error status of the last data manager call. Rather, it reflects the error status of data manager calls that don't return an error code. For some of those calls, the saved error code value is not set to 0 when the call is successful.

For example, if a call to `DmOpenDatabaseByTypeCreator` returns `NULL` for database reference (that is, it fails), `DmGetLastErr` returns something meaningful; otherwise, it returns the error value of some previous data manager call.

Only the following data manager functions currently affect the value returned by `DmGetLastErr`:

`DmFindDatabase`	`DmOpenDatabaseByTypeCreator`
`DmOpenDatabase`	`DmNewRecord`
`DmQueryRecord`	`DmGetRecord`
`DmQueryNextInCategory`	`DmPositionInCategory`
`DmSeekRecordInCategory`	`DmResizeRecord`
`DmGetResource`	`DmGet1Resource`
`DmNewResource`	`DmGetResourceIndex.`

DmGetNextDatabaseByTypeCreator

Purpose Return a database header ID and card number given the type and/or creator. This routine searches all memory cards for a match.

Prototype

```
Err DmGetNextDatabaseByTypeCreator (
             Boolean newSearch,
             DmSearchStatePtr stateInfoP,
             ULong type,
             ULong creator,
             Boolean onlyLatestVers,
             UIntPtr cardNoP,
             LocalID* dbIDP)
```

Parameters

-> `newSearch`	`TRUE` if starting a new search.
-> `stateInfoP`	If `newSearch` is `FALSE`, this must point to the same data used for the previous invocation.
-> `type`	Type of database to search for, pass 0 as a wildcard.
-> `creator`	Creator of database to search for, pass 0 as a wildcard.
-> `onlyLatestVers`	If `TRUE`, only latest version of each database with a given type and creator is returned.
<- `cardNoP`	On exit, the card number of the found database.
<- `dbIDP`	Database local ID of the found database.

Result

0	No error.
`dmErrCantFind`	No matches found.

Comments To start the search, pass `TRUE` for `newSearch`. To continue a search where the previous one left off, pass `FALSE` for `newSearch`. When continuing a search, `stateInfoP` must point to the same structure passed during the previous call to this function.

The `type` and `creator` parameters specify search criteria which a database must meet in order to be included in this function's result.

You may need to call this function successively to discover all databases having a specified type / creator pair.

You can pass `NIL` as a wildcard operator for the type or creator parameters to conduct searches of wider scope. If the `type` parameter is `NIL`, this routine can be called successively to return all databases of the given creator. If the `creator` parameter is `NIL`, this routine can be called successively to return all databases of the given type. You can also pass `NIL` as the value for both of these parameters to return all available databases without regard to type or creator.

Because databases are scattered freely throughout memory space, they are not returned in any particular order—any database matching the specified type / creator criteria can be returned.Thus, if the value of the `onlyLatestVers` parameter is `FALSE`, this function may return a database which is not the most recent version matching the specified type / creator pair. To obtain only the latest version of each database matching the search criteria, set the value of the `onlyLatestVers` parameter to `TRUE`,.

See Also `DmGetDatabase`, `DmFindDatabase`, `DmDatabaseInfo`, `DmOpenDatabaseByTypeCreator`, `DmDatabaseSize`

DmGetRecord

Purpose

Return a handle to a record by index and mark the record busy.

Prototype

```
VoidHand DmGetRecord ( DmOpenRef dbR,
                       UInt index)
```

Parameters

-> `dbR`	`DmOpenRef` to open database.
-> `index`	Which record to retrieve.

Result

Returns handle to record data.

Comments

Returns handle to given record and sets the `busy` bit for the record. If another call to `DmGetRecord` for the same record is attempted before the record is released, an error is returned.

If the record is ROM-based (pointer accessed), this routine makes a fake handle to it and store this handle in the `DmAccessType` structure.

`DmReleaseRecord` should be called as soon as the caller finishes viewing or editing the record.

See Also

`DmSearchRecord`, `DmFindRecordByID`, `DmRecordInfo`, `DmReleaseRecord`, `DmQueryRecord`

DmGetResource

Purpose Search all open resource databases and return a handle to a resource, given the resource type and ID.

Prototype `VoidHand DmGetResource (ULong type, Int ID)`

Parameters

-> `type`	The resource type.
->`ID`	The resource ID.

Result Returns pointer to resource data, or `NIL` if unsuccessful.

Comments Searches all open resource databases starting with the most recently opened one for a resource of the given type and ID. If found, the resource handle is returned. The application should call `DmReleaseRecord` as soon as it finishes accessing the resource data to avoid fragmenting the heap.

See Also `DmGet1Resource`, `DmReleaseResource`

DmGetResourceIndex

Purpose Return a handle to a resource by index.

Prototype

```
VoidHand DmGetResourceIndex (DmOpenRef dbR,
                             Int index)
```

Parameters

-> `dbR`	Access pointer to open database.
-> `index`	Index of resource to lock down.

Result Handle to resource data, or `NIL` if unsuccessful.

See Also `DmFindResource`, `DmFindResourceType`, `DmSearchResource`

DmGet1Resource

Purpose Search the most recently opened resource database and return a handle to a resource given the resource type and ID.

Prototype

```
VoidHand DmGet1Resource (ULong type, Int ID)
```

Parameters

-> `type`	The resource type.
-> `ID`	The resource ID.

Result Returns a pointer to resource data, or `NIL` if unsuccessful.

Comments Searches the most recently opened resource database for a resource of the given type and ID. If found, the resource handle is returned. The application should call `DmReleaseRecord` as soon as it finishes accessing the resource data in order to avoid fragmenting the heap.

See Also `DmGetResource`, `DmReleaseResource`

DmInsertionSort

Purpose Sort records in a database.

Prototype

```
Err DmInsertionSort (DmOpenRef dbR,
                     DmComparF *compar,
                     Int other)
```

Parameters

`dbR`	Database access pointer.
`compar`	Comparison function (see below).
`other`	Any value the application wants to pass to the comparison function.

Result Returns 0 if no error, or `dmErrReadOnly` if read-only database. Returns `dmErrInvalidParam` for an invalid parameter.

Comments

Deleted records are placed last in any order. All others are sorted according to the passed comparison function. Only records which are out of order move. Moved records are moved to the end of the range of equal records. If a large number of records are being sorted, try to use the quick sort.

The following insertion-sort algorithm is used: Starting with the second record, each record is compared to the preceding record. Each record not greater than the last is inserted into sorted position within those already sorted. A binary insertion is performed. A moved record is inserted after any other equal records.

The comparison function, `compar`, accepts two arguments, `*elem1` and `*elem2`, each a pointer to an entry in the table. The comparison function compares each of the pointed-to items (`*elem1` and `*elem2`), and returns an integer based on the `result*` of the comparison.

If the items...	`compar` returns...
`*elem1 < *elem2`	an integer < 0
`*elem1 == *elem2`	0
`*elem1 > *elem2`	an integer > 0

`DmInsertionSort` is also called by `SysAppLaunch` (see Part 1) to move an application database it is launching out of the system list and into the application's list.

2.0 Note

`DmComparF` has changed; it previously had 3 parameters and now has 6. `DmComparF` is the typedef of a callback used by `SysInsertionSort`, `DmInsertionSort`, and `FindInsertPosition`.

The new parameters allow a Palm OS application to pass more information to the system than before, most noticeably the record (and all associated information) which allows sorting by unique ID, so that the Palm OS device and the desktop always match.

The revised callback is used by new sorting routines (and can be used the same way by your application):

```
typedef Int DmComparF (void *,
   void *,
   Int other,
   SortRecordInfoPtr,
   SortRecordInfoPtr,
   VoidHand appInfoH);
```

As a rule, this change in the number of arguments doesn't cause problems when a 1.0 application is run on a 2.0 device, because the system only pulls the arguments from the stack that are there.

Keep in mind, however, that some optimized applications built with tools other than Metrowerks CodeWarrior for Palm OS may have problems as a result of the change in arguments when running on a 2.0 or later device.

See Also `DmQuickSort`

DmMoveCategory

Purpose Move all records in a category to another category.

Prototype

```
Err DmMoveCategory (DmOpenRef dbR,
                    UInt toCategory,
                    UInt fromCategory,
                    Boolean dirty)
```

Parameters

-> `dbR`	`DmOpenRef` to open database.
<- `toCategory`	Category to which to retrieve records.
-> `fromCategory`	Category from which to retrieve records.
-> `dirty`	If `TRUE`, set the `dirty` bit.

Result Returns 0 if successful, or `dmErrReadOnly` if read-only database.

Comments If `dirty` is `TRUE`, the moved records are marked as dirty.

DmMoveRecord

Purpose Move a record from one index to another.

Prototype

```
Err DmMoveRecord ( DmOpenRef dbR,
                   UInt from,
                   UInt to)
```

Parameters

-> `dbR`	`DmOpenRef` to open database.
-> `from`	Index of record to move.
-> `to`	Where to move the record.

Result Returns 0 if no error or one of `dmErrIndexOutOfRange`, `dmErrReadOnly`, `memErrChunkLocked`, `memErrInvalidParam`, or `memErrNotEnoughSpace` if an error occurs.

Comments Insert the record at the `to` index and move other records down. The `to` position should be viewed as an insertion position. This value may be one greater than the index of the last record in the database.

DmNewHandle

Purpose Attempt to allocate a new chunk in the same data heap or card as the database header of the passed database access pointer. If there is not enough space in that data heap, try other heaps.

Prototype `VoidHand DmNewHandle (DmOpenRef dbR, ULong size)`

Parameters

-> `dbR`	`DmOpenRef` to open database.
-> `size`	Size of new handle.

Result Returns the `chunkID` of new chunk, or 0 if not enough space.

Comments Allocates a new handle of the given size. Ensures that the new handle is in the same memory card as the given database. This guarantees that you can attach the handle to the database as a record to obtain and save its `LocalID` in the `appInfoID` or `sortInfoID` fields of the header.

DmNewRecord

Purpose Return a handle to a new record in the database and mark the record busy.

Prototype

```
VoidHand DmNewRecord ( DmOpenRef dbR,
                       UIntPtr atP,
                       ULong size)
```

Parameters

-> dbR	`DmOpenRef` to open database.
<-> atP	Pointer to index where new record should be placed.
-> size	Size of new record.

Result Pointer to record data, or 0 if error.

Comments Allocates a new record of the given size, and returns a handle to the record data. The parameter `atP` points to an index variable. The new record is inserted at index `*atP` and all record indices that follow are shifted down. If `*atP` is greater than the number of records currently in the database, the new record is appended to the end and its index is returned in `*atP`.

Both the `busy` and `dirty` bits are set for the new record and a unique ID is automatically created.

See Also `DmAttachRecord`, `DmRemoveRecord`, `DmDeleteRecord`

DmNewResource

Purpose Allocate and add a new resource to a resource database.

Prototype

```
VoidHand DmNewResource (DmOpenRef dbR,
                        ULong resType,
                        Int resID,
                        ULong size)
```

Parameters

-> `dbR`	`DmOpenRef` to open database.
-> `resType`	Type of the new resource.
-> `resID`	ID of the new resource.
-> `size`	Desired size of the new resource.

Result Returns a handle to new resource, or `NIL` if unsuccessful.

Comments Allocates a memory chunk for a new resource and adds it to the given resource database. The new resource has the given type and ID. If successful, the application should call `DmReleaseResource` as soon as it finishes initializing the resource.

See Also `DmAttachResource`, `DmRemoveResource`

DmNextOpenDatabase

Purpose Return `DmOpenRef` to next open database for the current task.

Prototype `DmOpenRef DmNextOpenDatabase (DmOpenRef currentP)`

Parameters -> `currentP` Current database access pointer or `NIL`.

Result `DmOpenRef` to next open database, or `NIL` if there are no more.

Comments Call this routine successively to get the `DmOpenRefs` of all open databases. Pass `NIL` for `currentP` to get the first one. Applications don't usually call this function, but is useful for system information.

See Also `DmOpenDatabaseInfo`, `DmDatabaseInfo`

DmNextOpenResDatabase

Purpose Return access pointer to next open resource database in the search chain.

Prototype `DmOpenRef DmNextOpenResDatabase (DmOpenRef dbR)`

Parameters `dbR` Database reference, or 0 to start search from the top.

Result Pointer to next open resource database.

Comments Returns pointer to next open resource database. To get a pointer to the first one in the search chain, pass `NIL` for `dbR`. This first database is the first and only one searched when `DmGet1Resource` is called.

DmNumDatabases

Purpose Determine how many databases reside on a memory card.

Prototype `UInt DmNumDatabases (UInt cardNo)`

Parameters -> `cardNo` Number of the card to check.

Result Returns the number of databases found.

Comments This routine is helpful for getting a directory of all databases on a card. The routine `DmGetDatabase` accepts an index from 0 to `DmNumDatabases` -1 and returns a database ID by index.

See Also `DmGetDatabase`

DmNumRecords

Purpose Return the number of records in a database.

Prototype `UInt DmNumRecords (DmOpenRef dbR)`

Parameters -> `dbR` `DmOpenRef` to open database.

Result Returns the number of records in a database.

See Also `DmNumRecordsInCategory`, `DmRecordInfo`, `DmSetRecordInfo`

DmNumRecordsInCategory

Purpose Return the number of records of a specified category in a database.

Prototype

```
UInt DmNumRecordsInCategory (DmOpenRef dbR,
                            UInt category)
```

Parameters

`dbr`	`DmOpenRef` to open database.
`category`	Category.

Result Returns the number of records.

Comments Because this function must examine all records in the database, it can be slow to return, especially when called on a large database.

See Also `DmNumRecords`, `DmQueryNextInCategory`, `DmPositionInCategory`, `DmSeekRecordInCategory`, `DmMoveCategory`

DmNumResources

Purpose Return the total number of resources in a given resource database.

Prototype

```
UInt DmNumResources (DmOpenRef dbR)
```

Parameters

-> `dbR`	`DmOpenRef` to open database.

Result Returns the total number of resources in the given database.

DmOpenDatabase

Purpose Open a database and return a reference to it.

Prototype

```
DmOpenRef DmOpenDatabase ( UInt cardNo,
                           LocalID dbID,
                           UInt mode)
```

Parameters

-> `cardNo`	Card number database resides on.
-> `dbID`	The database ID of the database.
-> `mode`	Which mode to open database in (see below).

Result Returns `DmOpenRef` to open database, or 0 if unsuccessful.

Comments Call this routine to open a database for reading or writing. The `mode` parameter can be one or more of the following constants ORed together:

`dmModeReadWrite`	Read-write access.
`dmModeReadOnly`	Read-only access.
`dmModeLeaveOpen`	Leave database open even after application quits.
`dmModeExclusive`	Don’t let anyone else open this database.

This routine returns a `DmOpenRef` which must be used to access particular records in a database. If unsuccessful, 0 is returned and the cause of the error can be determined by calling `DmGetLastErr`.

See Also `DmCloseDatabase`, `DmCreateDatabase`, `DmFindDatabase`, `DmOpenDatabaseByTypeCreator`, `DmDeleteDatabase`

DmOpenDatabaseByTypeCreator

Purpose Open the most recent revision of a database with the given type and creator.

Prototype

```
DmOpenRef DmOpenDatabaseByTypeCreator (
                          ULong type,
                          ULong creator,
                          UInt mode)
```

Parameters

`type`	Type of database.
`creator`	Creator of database.
`mode`	Open mode; see Comments for `DmOpenDatabase`.

Result `DmOpenRef` to open database, or 0 if unsuccessful.

See Also `DmCreateDatabase`, `DmOpenDatabase`, `DmOpenDatabaseInfo`, `DmCloseDatabase`

DmOpenDatabaseInfo

Purpose Retrieve information about an open database.

Prototype

```
Err DmOpenDatabaseInfo ( DmOpenRef dbR,
                         LocalIDPtr dbIDP,
                         UIntPtr openCountP,
                         UIntPtr modeP,
                         UIntPtr cardNoP,
                         BooleanPtr resDBP)
```

Parameters

-> `dbR`	`DmOpenRef` to open database.
<-> `dbIDP`	Pointer to return `dbID` variable, or `NIL`.
<-> `openCountP`	Pointer to return `openCount` variable, or `NIL`.
<-> `modeP`	Pointer to return `mode` variable, or `NIL`.
<-> `cardNoP`	Pointer to return card number, or `NIL`.
<-> `resDBP`	Pointer to return `resDB` Boolean, or `NIL`.

Result

0	No error.
`dmErrInvalidParam`	Invalid parameter passed.

Comments This routine retrieves information about an open database. Any `NIL` return parameter pointers are ignored.

See Also `DmDatabaseInfo`

DmPositionInCategory

Purpose Return a position of a record within the specified category.

Prototype

```
UInt DmPositionInCategory ( DmOpenRef dbR,
                            UInt index,
                            UInt category)
```

Parameters

`dbR`	`DmOpenRef` to open database.
`index`	Index of the record.
`category`	Category to search.

Result Returns the position (zero-based).

Comments Because this function must examine all records up to the current record, it can be slow to return, especially when called on a large database.

If the record is ROM-based (pointer accessed) this routine makes a fake handle to it and stores this handle in the `DmAccessType` structure.

See Also `DmQueryNextInCategory`, `DmSeekRecordInCategory`, `DmMoveCategory`

DmQueryNextInCategory

Purpose Return a handle to the next record in the specified category for reading only (does not set the `busy` bit).

Prototype
```
VoidHand DmQueryNextInCategory (DmOpenRef dbR,
                                UIntPtr indexP,
                                UInt category)
```

Parameters

`dbR`	`DmOpenRef` to open database.
`indexP`	Index of a known record (often retrieved with `DmPositionInCategory`).
`category`	Which category to query.

Result Returns a handle to the record following a known record.

See Also `DmNumRecordsInCategory`, `DmPositionInCategory`, `DmSeekRecordInCategory`

DmQueryRecord

Purpose Return a handle to a record for reading only (does not set the `busy` bit).

Prototype
```
VoidHand DmQueryRecord (DmOpenRef dbR, UInt index)
```

Parameters

-> `dbR`	`DmOpenRef` to open database.
-> `index`	Which record to retrieve.

Result Returns record handle, or 0 if record is out of range or deleted.

Comments Returns handle to given record. Use this routine only when viewing the record. This routine successfully returns a handle to the record even if the record is busy.

If the record is ROM-based (pointer accessed) this routine returns the fake handle to it.

DmQuickSort

Purpose Sort records in a database.

Prototype

```
Err DmQuickSort (const DmOpenRef dbR,
                 DmComparF *compar,
                 Int other)
```

Parameters

`dbR`	Database access pointer.
`compar`	Comparison function (see Comments).
`other`	Any value the application wants to pass to the comparison function.

Result Returns 0 if no error or `DmErrReadOnly` if an error occurred.

Comments Deleted records are placed last in any order. All others are sorted according to the passed comparison function.

The comparison function, `compar`, accepts two arguments, `elem1` and `elem2`, each a pointer to an entry in the table. The comparison function compares each of the pointed-to items (`*elem1` and `*elem2`), and returns an integer based on the result of the comparison.

If the items...	`compar` returns...
`*elem1 < *elem2`	an integer < 0
`*elem1 == *elem2`	0
`*elem1 > *elem2`	an integer > 0

See Also `DmFindSortPositionV10`, `DmInsertionSort`

DmRecordInfo

Purpose Retrieve the record information as stored in the database header.

Prototype

```
Err DmRecordInfo (DmOpenRef dbR,
                  UInt index,
                  UBytePtr attrP,
                  ULongPtr uniqueIDP,
                  LocalID* chunkIDP)
```

Parameters

-> `dbR`	`DmOpenRef` to open database.
-> `index`	Index of record.
<-> `attrP`	Pointer to return attribute variable, or `NIL`.
<-> `uniqueIDP`	Pointer to return unique ID variable, or `NIL`.
<-> `chunkIDP`	Pointer to return Local ID variable, or `NIL`.

Result Returns 0 if no error or `dmErrIndexOutOfRange` if an error occurred.

Comments Retrieves information about a record. Any of the return variable pointers can be `NIL`.

See Also `DmNumRecords`, `DmSetRecordInfo`, `DmQueryNextInCategory`

DmReleaseRecord

Purpose Clear the `busy` bit for the given record and set the `dirty` bit if dirty is `TRUE`.

Prototype

```
Err DmReleaseRecord (DmOpenRef dbR,
                     UInt index,
                     Boolean dirty)
```

Parameters

-> `dbR`	`DmOpenRef` to open database.
-> `index`	The record to unlock.
-> `dirty`	If `TRUE`, set the `dirty` bit.

Result Returns 0 if no error or `dmErrIndexOutOfRange` if an error occurred.

Comments Call this routine when you finish modifying or reading a record that you've called `DmGetRecord` on.

See Also `DmGetRecord`

DmReleaseResource

Purpose Release a resource acquired with `DmGetResource`.

Prototype

```
Err DmReleaseResource (VoidHand resourceH)
```

Parameters

-> `resourceH`	Handle to resource.

Result Returns 0 if no error.

Comments Marks a resource as being no longer needed by the application.

See Also `DmGet1Resource`, `DmGetResource`

DmRemoveRecord

Purpose Remove a record from a database and dispose of its data chunk.

Prototype

```
Err DmRemoveRecord ( DmOpenRef dbR,
                     UInt index)
```

Parameters

-> `dbR`	`DmOpenRef` to open database.
-> `index`	Index of the record to remove.

Result Returns 0 if no error, or `dmErrCorruptDatabase`, `dmErrIndexOutOfRange`, `dmErrReadOnly`, `memErrChunkLocked`, `memErrInvalidParam`, or `memErrNotEnoughSpace` if an error occurs.

Comments Disposes of a the record’s data chunk and removes the record’s entry from the database header.

See Also `DmDetachRecord`, `DmDeleteRecord`, `DmArchiveRecord`, `DmNewRecord`

DmRemoveResource

Purpose Delete a resource from a resource database.

Prototype `Err DmRemoveResource (DmOpenRef dbR, Int index)`

Parameters

-> `dbR`	`DmOpenRef` to open database.
-> `index`	Index of resource to delete.

Result Returns 0 if no error or `dmErrCorruptDatabase`, `dmErrIndexOutOfRange`, `dmErrReadOnly`, `memErrChunkLocked`, `memErrInvalidParam`, or `memErrNotEnoughSpace` if an error occurs.

Comments This routine disposes of the memory manager chunk that holds the given resource and removes its entry from the database header.

See Also `DmDetachResource`, `DmRemoveResource`, `DmAttachResource`

DmRemoveSecretRecords

Purpose Remove all secret records.

Prototype `Err DmRemoveSecretRecords (DmOpenRef dbR)`

Parameters

`dbR`	`DmOpenRef` to open database.

Result Returns 0 if no error or `dmErrReadOnly` (read-only database) if an error occurred.

See Also `DmRemoveRecord`, `DmRecordInfo`, `DmSetRecordInfo`

DmResetRecordStates

Purpose Unlock all records in a database and clear all `busy` bits.

Prototype

```
Err DmResetRecordStates (DmOpenRef dbR)
```

Parameters

-> `dbR`	`DmOpenRef` to open database.

Result Returns 0 if no error or `dmErrROMBased` if an error occurred.

Comments This routine unlocks all records in a database and clears all `busy` bits. It can optionally be called before closing a database to ensure that the records are all unlocked. For performance reasons, the data manager does not call `DmResetRecordStates` automatically when closing a database.

This routine automatically allocates the record in another data heap if the current heap is too full.

DmResizeRecord

Purpose Resize a record by index.

Prototype

```
VoidHand DmResizeRecord (DmOpenRef dbR,
                         UInt index,
                         ULong newSize)
```

Parameters

-> `dbR`	`DmOpenRef` to open database.
-> `index`	Which record to retrieve.
-> `newSize`	New size of record.

Result Pointer to resized record, or `NIL` if unsuccessful.

Comments This routine reallocates the record in another heap of the same memory card if the current heap is not big enough. If this happens, the handle changes, so be sure to use the returned handle to access the resized resource.

DmResizeResource

Purpose Resize a resource and return the new handle.

Prototype

```
VoidHand DmResizeResource (VoidHand resourceH,
                           ULong newSize)
```

Parameters

-> `resourceH`	Handle to resource.
-> `newSize`	Desired new size of resource.

Result Returns a handle to newly sized resource or `NIL` if unsuccessful.

Comments Resizes the resource and returns new handle. If necessary in order to grow the resource, this routine will reallocate it in another heap on the same memory card that it is currently in.

The handle may change if the resource had to be reallocated in a different data heap because there was not enough space in its present data heap.

DmResourceInfo

Purpose Retrieve information on a given resource.

Prototype

```
Err DmResourceInfo (DmOpenRef dbR,
                    Int index,
                    ULongPtr resTypeP,
                    IntPtr resIDP,
                    LocalID* chunkLocalIDP)
```

Parameters

-> `dbR`	`DmOpenRef` to open database.
-> `index`	Index of resource to get info on.
<-> `resTypeP`	Pointer to return `resType` variable, or `NIL`.
<-> `resIDP`	Pointer to return `resID` variable, or `NIL`.
<-> `chunkLocalIDP`	Pointer to return `chunkID` variable, or `NIL`.

Result Returns 0 if no error or `dmErrIndexOutOfRange` if an error occurred.

Comments Use this routine to retrieve all or a portion of the information on a particular resource. Any or all of the return variable pointers can be `NIL`. The type and ID of the resource are returned in `*resTypeP` and `*resIDP`. The memory manager local ID of the resource data is returned in `*chunkLocalIDP`.

See Also `DmGetResource`, `DmGet1Resource`, `DmSetResourceInfo`, `DmFindResource`, `DmFindResourceType`

DmSearchRecord

Purpose Search all open record databases for a record with the handle passed.

Prototype

```
Int DmSearchRecord (VoidHand recH,
                    DmOpenRef* dbRP)
```

Parameters

`recH`	Record handle.
`dbRP`	Pointer to return variable of type `DmOpenRef`.

Result Returns the index of the record and database access pointer; if not found, index will be -1 and `*dbRP` will be 0.

See Also `DmGetRecord`, `DmFindRecordByID`, `DmRecordInfo`

DmSearchResource

Purpose Search all open resource databases for a resource by type and ID, or by pointer if it is non-`NIL`.

Prototype

```
Int DmSearchResource (ULong resType,
                      Int resID,
                      VoidHand resH,
                      DmOpenRef* dbRP)
```

Parameters

-> `resType`	Type of resource to search for.
-> `resID`	ID of resource to search for.
-> `resH`	Pointer to locked resource, or `NIL`.
-> `dbRP`	Pointer to return variable of type `DmOpenRef`.

Result Returns the index of the resource, stores `DmOpenRef` in `dbRP`.

Comments This routine can be used to find a resource in all open resource databases by type and ID or by pointer. If `resH` is `NIL`, the resource is searched for by type and ID. If `resH` is not `NIL`, `resType` and `resID` is ignored and the index of the resource handle is returned. On return `*dbRP` contains the access pointer of the resource database that the resource was eventually found in. Once the index of a resource is determined, it can be locked down and accessed by calling `DmGetResourceByIndex`.

See Also `DmGetResource`, `DmFindResourceType`, `DmResourceInfo`, `DmGetResourceIndex`, `DmFindResource`

DmSeekRecordInCategory

Purpose Return the index of the record at the offset from the passed record index. (The `offset` parameter indicates the number of records to move forward or backward; the value for backward is negative.)

Prototype

```
Err DmSeekRecordInCategory (DmOpenRef dbR,
                            UIntPtr indexP,
                            Int offset,
                            Int direction,
                            UInt category)
```

Parameters

`dbR`	`DmOpenRef` to open database.
`index`	Pointer to the returned index.
`offset`	Offset of the passed record index.
`direction`	`dmSeekForward` or `dmSeekBackward`.
`category`	Category ID.

Result Returns 0 if no error; returns `dmErrIndexOutOfRange` or `dmErrSeekFailed` if an error occurred.

See Also `DmNumRecordsInCategory`, `DmQueryNextInCategory`, `DmPositionInCategory`, `DmMoveCategory`

DmSet

Purpose Write a specified value into a section of a record. This function also checks the validity of the pointer for the record and makes sure the writing of the record information doesn't exceed the bounds of the record.

Prototype

```
Err DmSet (VoidPtr recordP,
           ULong offset,
           ULong bytes,
           Byte value)
```

Parameters

`recordP`	Pointer to locked data record (chunk pointer).
`offset`	Offset within record to start writing.
`bytes`	Number of bytes to write.
`value`	Byte value to write.

Result Returns 0 if no error; returns `dmErrNotValidRecord` or `dmErrWriteOutOfBounds` if an error occurred.

Comments Must be used to write to data manager records because the data storage area is write-protected.

See Also `DmWrite`

DmSetDatabaseInfo

Purpose Set information about a database.

Prototype

```
Err DmSetDatabaseInfo (UInt cardNo,
        LocalID dbID, CharPtr nameP,
        UIntPtr attributesP, UIntPtr versionP
        ULongPtr crDateP, ULongPtr modDateP,
        ULongPtr bckUpDateP, ULongPtr modNumP,
        LocalID* appInfoIDP, LocalID* sortInfoIDP,
        ULongPtr typeP, ULongPtr creatorP)
```

Parameters

-> cardNo	Card number the database resides on.
-> dbID	Database ID of the database.
-> nameP	Pointer to 32-byte character array for new name, or NIL.
-> attributesP	Pointer to new attributes variable, or NIL.
versionP	Pointer to new version, or NIL.
-> crDateP	Pointer to new creation date variable, or NIL.
-> modDateP	Pointer to new modification date variable, or NIL.
-> bckUpDateP	Pointer to new backup date variable, or NIL.
-> modNumP	Pointer to new modification number variable, or NIL.
-> appInfoIDP	Pointer to new appInfoID variable, or NIL.
-> sortInfoIDP	Pointer to new sortInfoID variable, or NIL.
-> typeP	Pointer to new type variable, or NIL.
-> creatorP	Pointer to new creator variable, or NIL.

Result Returns 0 if no error or dmErrInvalidParam if an error occurred.

Comments When this call changes appInfoID or sortInfoID, the old chunk ID (if any) is marked as an orphan chunk and the new chunk ID is unorphaned. Consequently, you shouldn't replace an existing appInfoID or sortInfoID if that chunk has already been attached to another database.

Call this routine to set any or all information about a database except for the card number and database ID. This routine sets the new value for any non-NIL parameter.

See Also DmDatabaseInfo, DmOpenDatabaseInfo, DmFindDatabase, DmGetNextDatabaseByTypeCreator

DmSetRecordInfo

Purpose Set record information stored in the database header.

Prototype

```
Err DmSetRecordInfo ( DmOpenRef dbR,
                      UInt index,
                      UBytePtr attrP,
                      ULongPtr uniqueIDP)
```

Parameters

-> `dbR`	`DmOpenRef` to open database.
-> `index`	Index of record.
-> `attrP`	Pointer to new attribute variable, or `NIL`.
-> `uniqueIDP`	Pointer to new unique ID variable, or `NIL`.

Result Returns 0 if no error; returns `dmErrIndexOutOfRange` or `dmErrReadOnly` if an error occurred.

Comments Sets information about a record.

See Also `DmNumRecords`, `DmRecordInfo`

DmSetResourceInfo

Purpose Set information on a given resource.

Prototype

```
Err DmSetResourceInfo (DmOpenRef dbR,
                       Int index,
                       ULongPtr resTypeP,
                       IntPtr resIDP)
```

Parameters

-> dbR	DmOpenRef to open database.
-> index	Index of resource to set info for.
<-> resTypeP	Pointer to new resType, or NIL.
<-> resIDP	Pointer to new resID, or NIL.

Result Returns 0 if no error; returns dmErrIndexOutOfRange or dmErrReadOnly if an error occurred.

Comments Use this routine to set all or a portion of the information on a particular resource. Any or all of the new info pointers can be NIL. If not NIL, the type and ID of the resource are changed to *resTypeP and *resIDP.

Normally, the unique ID for a record is automatically created by the data manager when a record is created using DmNewRecord, so an application would not typically change the unique ID.

DmStrCopy

Purpose Check the validity of the chunk pointer for the record and make sure that writing the record will not exceed the chunk bounds.

Prototype

```
Err DmStrCopy (VoidPtr recordP,
               ULong offset,
               CharPtr srcP)
```

Parameters

`recordP`	Pointer to data record (chunk pointer).
`offset`	Offset within record to start writing.
`srcP`	Pointer to 0-terminated string.

Result Returns 0 if no error; returns `dmErrNotValidRecord` or `dmErrWriteOutOfBounds` if an error occurred.

See Also `DmWrite`, `DmSet`

DmWrite

Purpose Must be used to write to data manager records because the data storage area is write-protected. This routine checks the validity of the chunk pointer for the record and makes sure that the write will not exceed the chunk bounds.

Prototype

```
Err DmWrite (VoidPtr recordP,
             ULong offset,
             VoidPtr srcP,
             ULong bytes)
```

Parameters

`recordP`	Pointer to locked data record (chunk pointer).
`offset`	Offset within record to start writing.
`srcP`	Pointer to data to copy into record.
`bytes`	Number of bytes to write.

Result Returns 0 if no error; returns `dmErrNotValidRecord` or `dmErrWriteOutOfBounds` if an error occurred.

See Also `DmSet`

DmWriteCheck

Purpose

Check the parameters of a write operation to a data storage chunk before actually performing the write.

Prototype

```
Err DmWriteCheck ( VoidPtr recordP,
                   ULong offset,
                   ULong bytes)
```

Parameters

`recordP`	Locked pointer to `recordH`.
`offset`	Offset into record to start writing.
`bytes`	Number of bytes to write.

Result

Returns 0 if no error; returns `dmErrNotValidRecord` or `dmErrWriteOutOfBounds` if an error occurred.

Functions for System Use Only

DmMoveOpenDBContext

Prototype

```
Err DmMoveOpenDBContext (DmOpenRef* dstHeadP,
                         DmOpenRef dbR)
```

WARNING: System Use Only!

4

Palm OS Communications

The Palm OS communications software provides high-performance serial communications capabilities, including byte-level serial I/O, best-effort packet-based I/O with CRC-16, reliable data transport with retries and acknowledgments, connection management, and modem dialing capabilities.

This chapter helps you understand the different parts of the communications software and explains how to use them, discussing these topics:

- Byte Ordering briefly explains the byte order used for all data.
- Communications Architecture Hierarchy provides an overview of the hierarchy, including an illustration.
- The Serial Manager is responsible for byte-level serial I/O and control of the RS232 signals.
- The Serial Link Protocol provides an efficient mechanism for sending and receiving packets.
- The Serial Link Manager is the Palm OS implementation of the serial link protocol.

Byte Ordering

By convention, all data coming from and going to the Palm OS device use Motorola byte ordering. That is, data of compound types such as Word (2 bytes) and DWord (4 bytes), as well as their integral counterparts, are packaged with the most-significant byte at the lowest address. This contrasts with Intel byte ordering.

Communications Architecture Hierarchy

The communications software has multiple layers. Higher layers depend on more primitive functionality provided by lower layers. Applications can use functionality of all layers. The software consists of the following layers, described in more detail below:

- The serial manager, at the lowest layer, deals with the Palm OS serial port and control of the RS232 signals, providing byte-level serial I/O. See The Serial Manager.
- The modem manager provides modem dialing capabilities.
- The Serial Link Protocol (SLP) provides best-effort packet send and receive capabilities with CRC-16. Packet delivery is left to the higher-level protocols; SLP does not guarantee it. See The Serial Link Protocol.
- The Packet Assembly / Disassembly Protocol (PADP) sends and receives buffered data. PADP is an efficient protocol featuring variable-size block transfers with robust error checking and automatic retries. Applications don't need access to that part of the system.
- The Connection Management Protocol (CMP) provides connection-establishment capabilities featuring baud rate arbitration and exchange of communications software version numbers.
- The Desktop Link Protocol (DLP) provides remote access to Palm OS data storage and other subsystems.

  DLP facilitates efficient data synchronization between desktop (PC, Macintosh, etc.) and Palm OS applications, database backup, installation of code patches, extensions, applications, and other databases, as well as Remote Interapplication Communication (RIAC) and Remote Procedure Calls (RPC).

Figure 4.1 illustrates the communications layers.

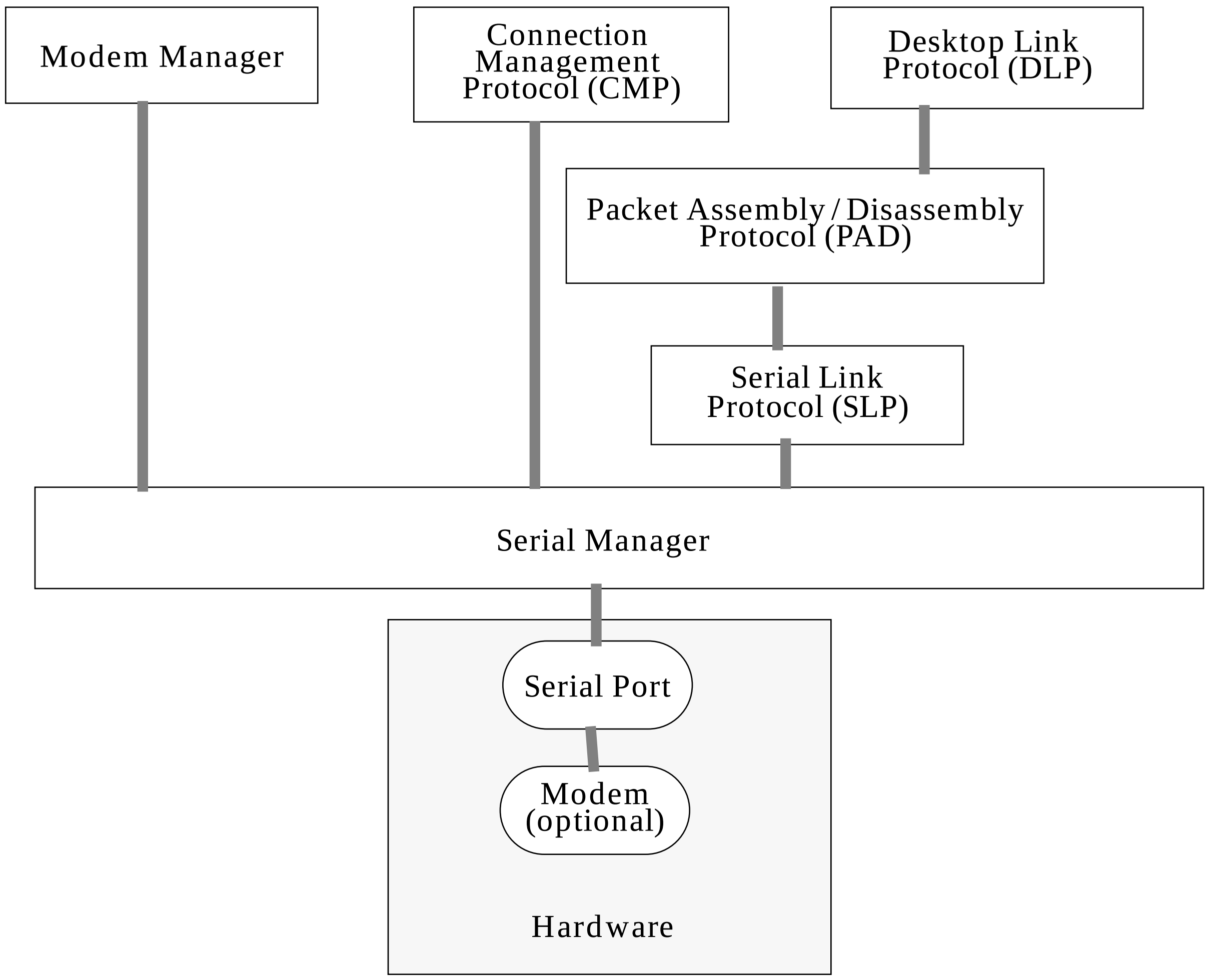


Figure 4.1 Palm OS Communications Architecture

The Serial Manager

The Palm OS serial manager is responsible for byte-level serial I/O and control of the RS232 signals.

In order to prolong battery life, the serial manager must be very efficient in its use of processing power. To reach this goal, the serial manager receiver is interrupt-driven. In the present implementation, the serial manager uses the polling mode to send data.

Using the Serial Manager

Before using the serial manager, call `SysLibFind`, passing `Serial Library` for the library name to get the serial library reference number. This reference number is used with all subsequent serial manager calls. To obtain the number, call `SysLibFind` with “Serial Library” as the library name. The system software automatically installs the serial library during system initialization.

To open the serial port, call `SerOpen`, passing the serial library reference number (returned by `SysLibFind`), 0 (zero) for the port number, and the desired baud rate. An error code of 0 (zero) or `serErrAlreadyOpen` indicates that the port was successfully opened.

If the serial port is already open when `SerOpen` is called, the port’s open count is incremented and an error code of `serErrAlreadyOpen` is returned. This ability to open the serial port multiple times allows cooperating tasks to share the serial port.

All other applications must refrain from sharing the serial port and close it by calling `SerClose` when `serErrAlreadyOpen` is returned. Error codes other than 0 (zero) or `serErrAlreadyOpen` indicate failure. The application must open the serial port before making other serial manager calls.

To close the serial port, call `SerClose`. Every successful call to `SerOpen` must eventually be paired with a call to `SerClose`. Because an open serial port consumes more energy from the device’s batteries, it is essential not to keep the port open any longer than necessary.

To change serial port settings, such as the baud rate, CTS timeout, number of data and stop bits, parity options, and handshaking options, call `SerSetSettings`. For baud rates above 19200, use of hardware handshaking is advised.

To retrieve the current serial port settings, call `SerGetStatus`.

To retrieve the current line error status, call `SerGetStatus`, which returns the cumulative status of all line errors being monitored. This includes parity, hardware and software overrun, framing, break detection, and handshake errors.

To reset the serial port error status, call `SerClearErr`, which resets the serial port's line error status. Other serial manager functions, such as `SerReceive`, immediately return with the error code `serErrLineErr` if any line errors are pending. Applications should therefore check the result of serial manager function calls and call `SerClearErr` if line error(s) occurred.

To send a stream of bytes, call `SerSend`. In the present implementation, `SerSend` blocks until all data are transferred to the UART or a timeout error (if CTS handshaking is enabled) occurs. If your software needs to detect when all data has been transmitted, consider calling `SerSendWait`.

2.0 Note Both `SerSend` and `SerReceive` have been enhanced in version 2.0 of the system. See the function descriptions for more information.

To wait until all data queued up for transmission has been transmitted, call `SerSendWait`. `SerSendWait` blocks until all pending data is transmitted or a CTS timeout error occurs (if CTS handshaking is enabled).

To flush all bytes from the transmission queue, call `SerSendWait`. This routine discards any data not yet transferred to the UART for transmission.

To receive a stream of bytes from the serial port, call SerReceive, specifying a buffer, the number of bytes desired, and the interbyte time out. This call blocks until all the requested data have been received or an error occurs.

To read bytes already in the receive queue, call SerReceiveCheck (see below) to get the number of bytes presently in the receive queue and then call SerReceive, specifying the number of bytes desired. Because SerReceive returns immediately without any data if line errors are pending, it is important to acknowledge the detection of line errors by calling SerClearErr.

To wait for a specific number of bytes to be queued up in the receive queue, call SerReceiveWait, passing the desired number of bytes and an interbyte timeout. This call blocks until the desired number of bytes have accumulated in the receive queue or an error occurs. The desired number of bytes must be less than the current receive queue size. The default queue size is 512 bytes. Because this call returns immediately if line errors are pending, applications have to call SerClearErr to detect any line errors. See also SerReceiveCheck and SerSetReceiveBuffer.

To check how many bytes are presently in the receive queue, call SerReceiveCheck.

To discard all data presently in the receive queue and to flush bytes coming into the serial port, call SerReceiveFlush, specifying the interbyte timeout. This call blocks until a time out occurs waiting for the next byte to arrive.

To replace the default receive queue, call SerSetReceiveBuffer, specifying the pointer to the buffer to be used for the receive queue and its size. The default receive queue must be restored before the serial port is closed. To restore the default receive queue, call SerSetReceiveBuffer, passing 0 (zero) for the buffer size. The serial manager does not free the custom receive queue.

To avoid having the system go to sleep while it's waiting to receive data, an application should call EvtResetAutoOffTimer periodically. For example, the serial link manager automatically calls EvtResetAutoOffTimer each time a new packet is received. Note that this facility is not part of the serial manager but part of the event manager. See Chapter 12, “System Manager Functions,” of “Developing Palm OS Applications, Part II.”

To perform a control function, applications can call `SerControl`. This Palm OS function performs one of the control operations specified by `SerCtlEnum`, which has the following elements:

Element	Description
`serCtlFirstReserved = 0`	Reserve 0
`serCtlStartBreak`	Turn RS232 break signal on. Applications have to make sure that the break is set long enough to generate a value BREAK! `valueP = 0; valueLenP = 0`
`serCtlStopBreak`	Turn RS232 break signal off: `valueP = 0; valueLenP = 0`
`serCtlBreakStatus`	Get RS232 break signal status (on or off): `valueP` = ptr to Word for returning status (0 = off, !0 = on) `*valueLenP = sizeof(`Word`)`
`serCtlStartLocalLoopback`	Start local loopback test; `valueP = 0, valueLenP = 0`
`serCtlStopLocalLoopback`	Stop local loopback test `valueP = 0, valueLenP = 0`
`serCtlMaxBaud`	`valueP = ptr` to `DWord` for returned baud `*valueLenP = sizeof(DWord)`
`serCtlHandshakeThreshold`	Retrieve HW handshake threshold; this is the maximum baud rate that does not require hardware handshaking `valueP` = `ptr` to DWord for returned baud `*valueLenP = sizeof(DWord)`

Element	Description
`serCtlEmuSetBlockingHook`	Set a blocking hook routine. WARNING: For use with the Simulator on Mac OS only: NOT SUPPORTED ON THE PALM DEVICE. `valueP =` ptr to SerCallbackEntryType `*valueLenP=sizeof(SerCallbackEntryType)` Returns the old settings in the first argument.
`serCtlLAST`	Add new address entries before this one.

Calling `serControl` with `serCtlEmuSetBlockingHook` replaces the mandatory need to define a `YieldTime` function. If the application never sets the blocking hook, then no blocking hook calls will be made.

The prototype for the blocking hook callback function is `SerBlockingHookHandler` which is defined and described in detail in `SerialMgr.h`.

Palm OS 1.0 developers that relied on the static `YieldTime` function for periodic processing such as draining the event queue and checking for user cancel action, have to add a parameter to their `YieldTime` function and call `serCtlEmuSetBlockingHook` to set their `YieldTime` function as the blocking hook callback function.

When applications no longer want the callback function to be called, they should call `serControl` with `serCtlEmuSetBlockingHook`, passing `NULL` for `funcP` in the `SerCallbackEntryType` structure.

Serial Manager Function Summary

The following functions are available for application use:

- SerClearErr
- SerClose
- SerControl
- SerGetSettings
- SerGetStatus
- SerOpen
- SerReceive
- SerReceiveCheck
- SerReceiveFlush
- SerReceiveWait
- SerSend
- SerSendWait
- SerSetReceiveBuffer
- SerSetSettings

The Serial Link Protocol

The Serial Link Protocol (SLP) provides an efficient packet send and receive mechanism. SLP provides robust error detection with CRC-16. SLP is a best-effort protocol; it does not guarantee packet delivery (packet delivery is left to the higher-level protocols). For enhanced error detection and implementation convenience of higher-level protocols, SLP specifies packet type, source, destination, and transaction ID information as an integral part of its data packet structure.

SLP Packet Structures

The following sections describe:

- SLP Packet Format
- Packet Type Assignment
- Socket ID Assignment
- Transaction ID Assignment

SLP Packet Format

Each SLP packet consists of a packet header, client data of variable size, and a packet footer, as shown in Figure 4.2.

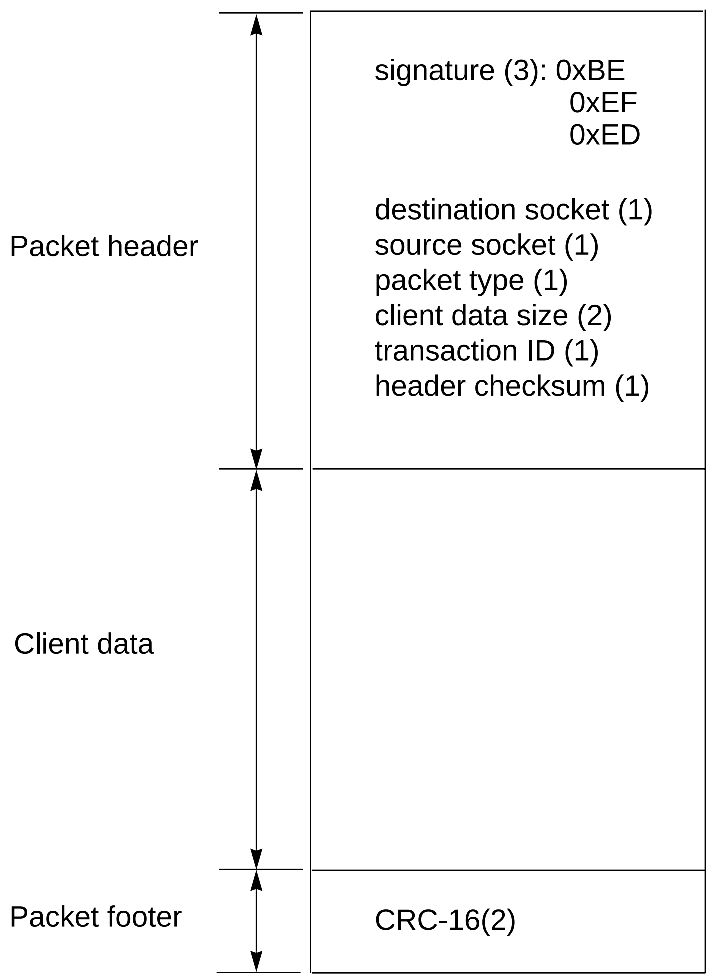


Figure 4.2 Structure of a Serial Link Packet

- The **packet header** contains the packet signature, the destination socket ID, the source socket ID, packet type, client data size, transaction ID, and header checksum. The packet signature is composed of the three bytes 0xBE, 0xEF, 0xED, in that order. The header checksum is an 8-bit arithmetic checksum of the entire packet header, not including the checksum field itself.

- The **client data** is a variable-size block of binary data specified by the user and is not interpreted by the Serial Link Protocol.
- The **packet footer** consists of the CRC-16 value computed over the packet header and client data.

Packet Type Assignment

Packet type values in the range of 0x00 through 0x7F are reserved for use by the system software. The following packet type assignments are currently implemented:

0x00	Remote Debugger, Remote Console, and System Remote Procedure Call packets.
0x02	PADP packets.
0x03	Loop-back test packets.

Socket ID Assignment

Socket IDs are divided into two categories: static and dynamic. The static socket IDs are “well-known” socket ID values that are reserved by the components of the system software. The dynamic socket IDs are assigned at runtime when requested by clients of SLP. Static socket ID values in the ranges 0x00 through 0x03 and 0xE0 through 0xFF are reserved for use by the system software. The following static socket IDs are currently implemented or reserved:

0x00	Remote Debugger socket.
0x01	Remote Console socket.
0x02	Remote UI socket.
0x03	Desktop Link Server socket.

0x04 -0xCF	Reserved for dynamic assignment.
0xD0 - 0xDF	Reserved for testing.

Transaction ID Assignment

Transaction ID values are not interpreted by the Serial Link Protocol and are for the sole benefit of the higher-level protocols. The following transaction ID values are currently reserved:

0x00 and 0xFF	Reserved for use by the system software.
0x00	Reserved by the Palm OS implementation of SLP to request automatic transaction ID generation.
0xFF	Reserved for the connection manager's WakeUp packets.

Transmitting an SLP Packet

This section provides an overview of the steps involved in transmitting an SLP packet. The next section describes the implementation.

Transmission of an SLP packet consists of these steps:

1. Fill in the packet header and compute its checksum.
2. Compute the CRC-16 of the packet header and client data.
3. Transmit the packet header, client data, and packet footer.
4. Return an error code to the client.

Receiving an SLP Packet

Receiving an SLP packet consists of these steps:

1. Scan the serial input until the packet header signature is matched.
2. Read in the rest of the packet header and validate its checksum.

3. Read in the client data.
4. Read in the packet footer and validate the packet CRC.
5. Dispatch / return an error code and the packet (if successful) to the client.

The Serial Link Manager

The serial link manager is the Palm OS implementation of the Palm OS Serial Link Protocol.

Serial link manager provides the mechanisms for managing multiple client sockets, sending packets, and receiving packets both synchronously and asynchronously. It also provides support for the Remote Debugger and Remote Procedure Calls (RPC).

Using the Serial Link Manager

Before an application can use the services of the serial link manager, the application must open the manager by calling `SlkOpen`. Success is indicated by error codes of 0 (zero) or `slkErrAlreadyOpen`. The return value `slkErrAlreadyOpen` indicates that the serial link manager has already been opened (most likely by another task). Other error codes indicate failure.

When you finish using the serial link manager, call `SlkClose`. `SlkClose` may be called only if `SlkOpen` returned 0 (zero) or `slkErrAlreadyOpen`. When open count reaches zero, `SlkClose` frees resources allocated by `SlkOpen`.

To use the serial link manager socket services, open a Serial Link socket by calling `SlkOpenSocket`. Pass a reference number of an opened and initialized communications library (see `SlkClose`), a pointer to a memory location for returning the socket ID, and a Boolean indicating whether the socket is static or dynamic. If a static socket is being opened, the memory location for the socket ID must contain the desired socket number. If opening a dynamic socket, the new socket ID is returned in the passed memory location. Sharing of sockets is not supported. Success is indicated by an error code of 0 (zero). For information about static and dynamic socket IDs, see Socket ID Assignment.

When you have finished using a Serial Link socket, close it by calling `SlkCloseSocket`. This releases system resources allocated for this socket by the serial link manager.

To obtain the communications library reference number for a particular socket, call `SlkSocketRefNum`. The socket must already be open.

To set the interbyte packet receive timeout for a particular socket, call `SlkSocketSetTimeout`.

To flush the receive stream for a particular socket, call `SlkFlushSocket`, passing the socket number and the interbyte timeout.

To register a socket listener for a particular socket, call `SlkSetSocketListener`, passing the socket number of an open socket and a pointer to the `SlkSocketListenType` structure. Because the serial link manager does not make a copy of the `SlkSocketListenType` structure but instead saves the pointer passed to it, the structure may not be an automatic variable (that is, allocated on the stack). The `SlkSocketListenType` structure may be a global variable in an application or a locked chunk allocated from the dynamic heap. The `SlkSocketListenType` structure specifies pointers to the socket listener procedure and the data buffers for dispatching packets destined for this socket. Pointers to two buffers must be specified:

- Packet header buffer (size of `SlkPktHeaderType`).
- Packet body buffer, which must be large enough for the largest expected client data size.

Both buffers can be application global variables or locked chunks allocated from the dynamic heap.

The socket listener procedure is called when a valid packet is received for the socket. Pointers to the packet header buffer and the packet body buffer are passed as parameters to the socket listener procedure. The serial link manager does not free the

`SlkSocketListenType` structure or the buffers when the socket is closed; freeing them is the responsibility of the application. For this mechanism to function, some task needs to assume the responsibility to "drive" the serial link manager receiver by periodically calling `SlkReceivePacket`.

To send a packet, call `SlkSendPacket`, passing a pointer to the packet header (`SlkPktHeaderType`) and a pointer to an array of `SlkWriteDataType` structures. `SlkSendPacket` stuffs the signature, client data size, and the checksum fields of the packet header. The caller must fill in all other packet header fields. If the transaction ID field is set to 0 (zero), the serial link manager automatically generates and stuffs a new non-zero transaction ID. The array of `SlkWriteDataType` structures enables the caller to specify the client data part of the packet as a list of noncontiguous blocks. The end of list is indicated by an array element with the `size` field set to 0 (zero). Listing 3.1 incorporates the processes described in this section.

Listing 4.1 Sending a Serial Link Packet

```
Err                err;
SlkPktHeaderType   sendHdr;
                //serial link packet header
SlkWriteDataType   writeList[2];
                //serial link write data segments
Byte        body[20];
                //packet body(example packet body)

    // Initialize packet body
    ...

// Compose the packet header
sendHdr.dest = slkSocketDLP;
sendHdr.src = slkSocketDLP;
sendHdr.type = slkPktTypeSystem;
sendHdr.transId = 0;
        // let Serial Link Manager set the transId
// Specify packet body
writeList[0].size = sizeof(body);
```

```
        // first data block size
writeList[0].dataP = body;
        // first data block pointer
writeList[1].size = 0;
        // no more data blocks

// Send the packet
err = SlkSendPacket( &sendHdr, writeList );
  ...
}
```

Listing 4.2 Generating a New Transaction ID

```
//
// Example: Generating a new transaction ID given the previous
// transaction ID. Can start with any seed value.
//

Byte NextTransactionID (Byte previousTransactionID)
{
  Byte nextTransactionID;

  // Generate a new transaction id, avoid the
  // reserved values (0x00 and 0xFF)
  if ( previousTransactionID >= (Byte)0xFE )
    nextTransactionID = 1;          // wrap around
  else
    nextTransactionID = previousTransactionID + 1;
                                    // increment

  return nextTransactionID;
}
```

To receive a packet, call `SlkReceivePacket`. You may request a packet for the passed socket ID only or for any open socket that does not have a socket listener. The parameters also specify buffers for the packet header and client data, and a timeout. The timeout indicates how long the receiver should wait for a packet to begin arriving before timing out. A timeout value of (-1) means “wait forever." If a packet is received for a socket with a registered socket listener, the packet is dispatched via its socket listener procedure.

Serial Link Manager Function Summary

The following functions are available for application use:

- `SlkClose`
- `SlkCloseSocket`
- `SlkFlushSocket`
- `SlkOpen`
- `SlkOpenSocket`
- `SlkReceivePacket`
- `SlkSendPacket`
- `SlkSetSocketListener`
- `SlkSocketRefNum`
- `SlkSocketSetTimeout`

5

Communications Functions

Serial Manager Functions

SerClearErr

Purpose Reset the serial port's line error status.

Prototype `Err SerClearErr (UInt refNum)`

Parameters -> `refNum` The serial library reference number.

Result 0 No error.

Caveats Call `SerClearErr` only after a serial manager function (`SerReceive`, `SerReceiveCheck`, `SerSend`, etc.) returns with the error code `serErrLineErr`.

The reason for this is that `SerClearErr` resets the serial port. So, if `SerClearErr` is called unconditionally while a byte is coming into the serial port, that byte is guaranteed to become corrupted.

The right strategy is to always check the error code returned by a serial manager function. If it 's `serErrLineErr`, call `SerClearErr` immediately. However, don't make unsolicited calls to `SerClearErr`.

When you get `serErrLineErr`, consider flushing the receive queue for a fraction of a second by calling `SerReceiveFlush`. `SerReceiveFlush` calls `SerClearErr` for you.

SerClose

Purpose Release the serial port previously acquired by `SerOpen`.

Prototype `Err SerClose (UInt refNum)`

Parameters

-> `refNum`	Serial library reference number.

Result

0	No error.
`serErrNotOpen`	Port wasn't open.
`serErrStillOpen`	Port still held open by another process.

Comments Releases the serial port and shuts down serial port hardware if the open count has reached 0. Open serial ports consume more energy from the device's batteries; it's therefore essential to keep a port open only as long as necessary.

Caveat Don't call `SerClose` unless the return value from `SerOpen` was 0 (zero) or `serErrAlreadyOpen`.

See Also `SerOpen`

SerControl

Purpose Perform a control function.

Prototype

```
Err SerControl (UInt refNum,
                Word op,
                VoidPtr valueP,
                WordPtr valueLenP)
```

Parameters

-> `refNum`	Reference number of library.
-> `op`	Control operation to perform(`SerCtlEnum`).
<-> `valueP`	Pointer to value for operation.
<-> `valueLenP`	Pointer to size of value.

Result

0	No error.
`serErrBadParam`	Invalid parameter (unknown).
`serErrNotOpen`	Library not open.

Comments This function provides extensible control features for the serial manager. You can

- Turn on / off the RS232 break signal and check its status.
- Perform a local loopback test.
- Get the maximum supported baud rate.
- Get the hardware handshake threshold baud rate.

There is one emulator-only control, `serCtlEmuSetBlockingHook`. See Using the Serial Manager for more information.

SerGetSettings

Purpose Fill in `SerSettingsType` structure with current serial port attributes.

Prototype

```
Err SerGetSettings (UInt refNum,
                    SerSettingsPtr settingsP)
```

Parameters

-> `refNum`	Serial library reference number.
<-> `settingsP`	Pointer to `SerSettingsType` structure to be filled in.

Result

0	No error.
`serErrNotOpen`	The port wasn't open.

Comments The information returned by this call includes the current baud rate, CTS timeout, handshaking options, and data format options.

See the `SerSettingsType` structure for more details.

See Also `SerSend`

SerGetStatus

Purpose

Return the pending line error status for errors that have been detected since the last time `SerClearErr` was called.

Prototype

```
Word SerGetStatus (UInt refNum,
                   BooleanPtr ctsOnP,
                   BooleanPtr dsrOnP)
```

Parameters

-> `refNum`	Serial library reference number.
-> `ctsOnP`	Pointer to location for storing a Boolean value.
-> `dsrOnP`	Pointer to location for storing a Boolean value.

Result

Returns any combination of the following constants, bitwise ORed together:

`serLineErrorParity`	Parity error.
`serLineErrorHWOverrun`	Hardware overrun.
`serLineErrorFraming`	Framing error.
`serLineErrorBreak`	Break signal detected.
`serLineErrorHShake`	Line handshake error.
`serLineErrorSWOverrun`	Software overrun.

Comments

When another serial manager function returns an error code of `serErrLineErr`, `SerGetStatus` can be used to find out the specific nature of the line error(s).

The values returned via `ctsOnP` and `dsrOnP` are not meaningful in the present version of the software

See Also

`SerClearErr`

SerOpen

Purpose Acquire and open a serial port with given baud rate and default settings.

Prototype `Err SerOpen (UInt refNum, UInt port, ULong baud)`

Parameters

-> `refNum`	Serial library reference number.
-> `port`	Port number.
-> `baud`	Baud rate.

Result

0	No error.
`serErrAlreadyOpen`	Port was open. Enables port sharing by "friendly" clients (not recommended).
`serErrBadParam`	Invalid parameter.
`memErrNotEnoughSpace`	Insufficient memory.

Comments Acquires the serial port, powers it up, and prepares it for operation. To obtain the serial library reference number, call `SysLibFind` with "Serial Library" as the library name. This reference number must be passed as a parameter to all serial manager functions. The device currently contains only one serial port with port number 0 (zero).

The baud rate is an integral baud value (for example - 300, 1200, 2400, 9600, 19200, 38400, 57600, etc.). The Palm OS device has been tested at the standard baud rates in the range of 300 - 57600 baud. Baud rates through 1 Mbit are theoretically possible. Use CTS handshaking at baud rates above 19200 (see `SerSetSettings`).

An error code of 0 (zero) or `serErrAlreadyOpen` indicates that the port was successfully opened. If the port is already open when `SerOpen` is called, the port's open count is incremented and an error code of `serErrAlreadyOpen` is returned. This ability to open the serial port multiple times allows cooperating tasks to share the serial port. Other tasks must refrain from using the port if `serErrAlreadyOpen` is returned and close it by calling `SerClose`.

SerReceive

Purpose Receives `size` bytes worth of data or returns with error if a line error or timeout is encountered.

Prototype

```
ULong SerReceive (UInt refNum,
                  VoidPtr rcvBufP,
                  ULong count,
                  Long timeout,
                  Err* errP)
```

Parameters

`refNum`	Serial library reference number.
<-> `rcvBufP`	Buffer for receiving data.
-> `count`	Number of bytes to receive.
-> `timeout`	Interbyte timeout in ticks, 0 for none, -1 forever.

Result Number of bytes received:

`*errP = 0`	No error.
`serErrLineErr`	RS232 line error.
`serErrTimeOut`	Interbyte timeout.

See Also `SerReceive10`

SerReceive10

Purpose Receive a stream of bytes.

Prototype

```
Err SerReceive (UInt refNum,
                VoidPtr bufP,
                ULong bytes,
                Long timeout)
```

Parameters

-> `refNum`	The serial library reference number.
-> `bufP`	Pointer to the buffer for receiving data.
-> `bytes`	Number of bytes desired.
-> `timeout`	Interbyte time out in system ticks (-1 = forever).

Result

0	No error. Requested number of bytes was received.
`serErrTimeOut`	Interbyte time out exceeded while waiting for the next byte to arrive.
`serErrLineErr`	Line error occurred (see `SerClearErr` and `SerGetStatus`).

Comments `SerReceive` blocks until all the requested data has been received or an error occurs. Because this call returns immediately without any data if line errors are pending, it is important to acknowledge the detection of line errors by calling `SerClearErr`. If you just need to retrieve all or some of the bytes which are already in the receive queue, call `SerReceiveCheck` first to get the count of bytes presently in the receive queue.

SerReceiveCheck

Purpose Return the count of bytes presently in the receive queue.

Prototype

```
Err SerReceiveCheck (UInt refNum,
                     ULongPtr numBytesP)
```

Parameters

-> `refNum`	Serial library reference number.
<-> `numBytesP`	Pointer to location for returning the byte count.

Result

0	No error.
`serErrLineErr`	Line error pending (see `SerClearErr` and `SerGetStatus`).

Comments Because this call does not return the byte count if line errors are pending, it is important to acknowledge the detection of line errors by calling `SerClearErr`.

See also `SerReceiveWait`

SerReceiveFlush

Purpose Discard all data presently in the receive queue and flush bytes coming into the serial port. Clear the saved error status.

Prototype

```
void SerReceiveFlush (UInt refNum, Long timeout)
```

Parameters

-> `refNum`	Serial library reference number.
-> `timeout`	Interbyte time out in system ticks (-1 = forever).

Result Returns nothing.

Comments `SerReceiveFlush` blocks until a timeout occurs while waiting for the next byte to arrive.

SerReceiveWait

Purpose Wait for at least `bytes` bytes of data to accumulate in the receive queue.

Prototype

```
Err SerReceiveWait (UInt refNum,
                    ULong bytes,
                    Long timeout)
```

Parameters

-> `refNum`	Serial library reference number.
-> `bytes`	Number of bytes desired.
-> `timeout`	Interbyte timeout in system ticks (-1 = forever).

Result

0	No error.
`serErrTimeOut`	Interbyte timeout exceeded while waiting for next byte to arrive.
`serErrLineErr`	Line error occurred (see `SerClearErr` and `SerGetStatus`).

Comments This is the preferred method of waiting for serial input, since it blocks the current task and allows switching the processor into a more energy-efficient state.

`SerReceiveWait` blocks until the desired number of bytes accumulate in the receive queue or an error occurs. The desired number of bytes must be less than the current receive queue size. The default queue size is 512 bytes. Because this call returns immediately if line errors are pending, it is important to acknowledge the detection of line errors by calling `SerClearErr`.

See also `SerReceiveCheck`, `SerSetReceiveBuffer`

SerSend

Purpose

Send one or more bytes of data over the serial port.

Prototype

```
ULong SerSend (UInt refNum,
               VoidPtr bufP,
               ULong count,
               Err* errP
```

Parameters

-> `refNum`	Serial library reference number.
-> `bufP`	Pointer to data to send.
-> `count`	Number of bytes to send.
<-> `errP`	For returning error code.

Result

Returns the number of bytes transferred.

Stores in `errP`:

0	No error.
`serErrTimeOut`	Handshake timeout.

NOTE: The old versions of `SerSend` and `SerReceive` are still available as `SerSend10` and `SerReceive10` (not V10).

The old calls worked, but they did not return enough info when they failed. The new calls (available in Palm OS devices >= v2.0) add more parameters to solve this problem and make serial communications programming simpler.

Don’t call the new functions when running on Palm OS 1.0.

SerSend10

Purpose Send a stream of bytes to the serial port.

Prototype

```
Err SerSend10 (UInt refNum,
               VoidPtr bufP,
               ULong size)
```

Parameters

-> `refNum`	Serial library reference number.
-> `bufP`	Pointer to the data to send.
-> `size`	Size (in number of bytes) of the data to send.

Result

0	No error.
`serErrTimeOut`	Handshake timeout (such as waiting for CTS to become asserted).

Comments In the present implementation, `SerSend` blocks until all data is transferred to the UART or a timeout error (if CTS handshaking is enabled) occurs. Future implementations may queue up the request and return immediately, performing transmission in the background. If your software needs to detect when all data has been transmitted, see `SerSendWait`.

This routine observes the current CTS time out setting if CTS handshaking is enabled (see `SerGetSettings` and `SerSend`).

SerSendWait

Purpose Wait until the serial transmit buffer empties.

Prototype `Err SerSendWait (UInt refNum, Long timeout)`

Parameters

-> `refNum`	Serial library reference number.
-> `timeout`	Reserved for future enhancements. Set to (-1) for compatibility.

Result

0	No error.
`serErrTimeOut`	Handshake timeout (such as waiting for CTS to become asserted).

Comments `SerSendWait` blocks until all data is transferred or a timeout error (if CTS handshaking is enabled) occurs. This routine observes the current CTS timeout setting if CTS handshaking is enabled (see `SerGetSettings` and `SerSend`).

SerSetReceiveBuffer

Purpose Replace the default receive queue. To restore the original buffer, pass `bufSize` = 0.

Prototype

```
Err SerSetReceiveBuffer (UInt refNum,
                         VoidPtr bufP,
                         UInt bufSize)
```

Parameters

-> `refNum`	Serial library reference number.
-> `bufP`	Pointer to buffer to be used as the new receive queue.
-> `bufSize`	Size of buffer, or 0 to restore the default receive queue.

Result Returns 0 if successful.

Comments The specified buffer needs to contain 32 extra bytes for serial manager overhead (its size should be your application's requirement plus 32 bytes). The default receive queue must be restored before the serial port is closed. To restore the default receive queue, call `SerSetReceiveBuffer` passing 0 (zero) for the buffer size. The serial manager does not free the custom receive queue.

SerSetSettings

Purpose Set the serial port settings; that is, change its attributes.

Prototype

```
Err SerSetSettings (UInt refNum,
                    SerSettingsPtr settingsP)
```

Parameters

-> `refNum`	Serial library reference number.
<-> `settingsP`	Pointer to the filled in `SerSettingsType` structure.

Result

0	No error.
`serErrNotOpen`	The port wasn't open.
`serErrBadParam`	Invalid parameter.

Comments The attributes set by this call include the current baud rate, CTS timeout, handshaking options, and data format options. See the definition of the `SerSettingsType` structure for more details.

To do 7E1 transmission, `OR` together:

```
serSettingsFlagBitsPerChar7 |
serSettingsFlagParityOnM |
serSettingsFlagParityEvenM |
serSettingsFlagStopBits1
```

If you're trying to communicate at speeds greater than 19.2 KbPS, you need to use hardware handshaking: `serSettingsFlagRTSAutoM` | `serSettingsFlagCTSAutoM`.

See Also `SerGetSettings`

Functions Used Only by System Software

These routines are for use by the system software only and should not be called by the applications under any circumstances.

SerReceiveISP

WARNING: This function for use by system software only.

SerReceiveWindowClose

WARNING: This function for System use only.

SerReceiveWindowOpen

WARNING: This function for System use only.

SerSetWakeupHandler

WARNING: This function for System use only.

SerSleep

WARNING: This function for use by system software only.

SerWake

WARNING: This function for use by system software only.

Serial Link Manager Functions

SlkClose

Purpose Close down the serial link manager.

Prototype `Err SlkClose (void)`

Parameters None.

Result

0	No error.
`slkErrNotOpen`	The serial link manager was not open.

Comments When the open count reaches zero, this routine frees resources allocated by serial link manager.

SlkCloseSocket

Purpose Closes a socket previously opened with `SlkOpenSocket`.

WARNING: The caller is responsible for closing the communications library used by this socket, if necessary.

Prototype `Err SlkCloseSocket (UInt socket)`

Parameters

`socket`	The socket ID to close.

Result

0	No error.
`slkErrSocketNotOpen`	The socket was not open.

Comments `SlkCloseSocket` frees system resources the serial link manager allocated for the socket. It does not free resources allocated and passed by the client, such as the buffers passed to `SlkSetSocketListener`; this is the client's responsibility. The caller is also responsible for closing the communications library used by this socket.

See Also `SlkOpenSocket`, `SlkSocketRefNum`

SlkFlushSocket

Purpose Flush the receive queue of the communications library associated with the given socket.

Prototype `ErrSlkFlushSocket (UInt socket, Long timeout)`

Parameters

-> `socket`	Socket ID.
-> `timeout`	Interbyte timeout in system ticks.

Result

0	No error.
`slkErrSocketNotOpen`	The socket wasn't open.

SlkOpen

Purpose Initialize the serial link manager.

Prototype `Err SlkOpen (void)`

Parameters None.

Result

0	No error.
`slkErrAlreadyOpen`	No error.

Comments Initializes the serial link manager, allocating necessary resources. Return codes of 0 (zero) and `slkErrAlreadyOpen` both indicate success. Any other return code indicates failure. The `slkErrAlreadyOpen` function informs the client that someone else is also using the serial link manager. If the serial link manager was successfully opened by the client, the client needs to call `SlkClose` when it finishes using the serial link manager.

SlkOpenSocket

Purpose Open a serial link socket and associate it with a communications library. The socket may be a known static socket or a dynamically assigned socket.

Prototype

```
Err SlkOpenSocket (UInt libRefNum,
                   UIntPtr socketP,
                   Boolean staticSocket)
```

Parameters

`libRefNum`	Comm library reference number for socket.
`socketP`	Pointer to location for returning the socket ID.
`staticSocket`	If TRUE, `*socketP` contains the desired static socket number to open. If FALSE, any free socket number is assigned dynamically and opened.

Result

0	No error.
`slkErrOutOfSockets`	No more sockets can be opened.

Comments The communications library must already be initialized and opened (see `SerOpen`). When finished using the socket, the caller must call `SlkCloseSocket` to free system resources allocated for the socket. For information about well-known static socket IDs, see The Serial Link Protocol.

SlkReceivePacket

Purpose Receive and validate a packet for a particular socket or for any socket. Check for format and checksum errors.

Prototype

```
Err SlkReceivePacket (UInt socket,
                      Boolean andOtherSockets,
                      SlkPktHeaderPtr headerP,
                      void* bodyP,
                      UInt bodySize,
                      Long timeout)
```

Parameters

-> `socket`	The socket ID.
-> `andOtherSockets`	If `TRUE`, ignore destination in packet header.
<-> `headerP`	Pointer to the packet header buffer (size of `SlkPktHeaderType`).
<-> `bodyP`	Pointer to the packet client data buffer.
-> `bodySize`	Size of the client data buffer (maximum client data size which can be accommodated).
-> `timeout`	Maximum number of system ticks to wait for beginning of a packet; -1 means wait forever.

Result

0	No error.
`slkErrSocketNotOpen`	The socket was not open.
`slkErrTimeOut`	Timed out waiting for a packet.
`slkErrWrongDestSocket`	The packet being received had an unexpected destination.
`slkErrChecksum`	Invalid header checksum or packet CRC-16.
`slkErrBuffer`	Client data buffer was too small for packet's client data.

If `andOtherSockets` is `FALSE`, this routine returns with an error code unless it gets a packet for the specific socket.

If `andOtherSockets` is `TRUE`, this routine returns successfully if it sees any incoming packet from the communications library used by `socket`.

Comments You may request to receive a packet for the passed socket ID only, or for any open socket which does not have a socket listener. The parameters also specify buffers for the packet header and client data, and a timeout. The timeout indicates how long the receiver should wait for a packet to begin arriving before timing out. If a packet is received for a socket with a registered socket listener, it will be dispatched via its socket listener procedure. On success, the packet header buffer and packet client data buffer is filled in with the actual size of the packet's client data in the packet header's `bodySize` field.

SlkSendPacket

Purpose Send a serial link packet via the serial output driver.

Prototype

```
Err SlkSendPacket (SlkPktHeaderPtr headerP,
                   SlkWriteDataPtr writeList)
```

Parameters

<-> `headerP`	Pointer to the packet header structure with client information filled in (see Comments).
-> `writeList`	List of packet client data blocks (see Comments).

Result

0	No error.
`slkErrSocketNotOpen`	The socket was not open.
`slkErrTimeOut`	Handshake timeout.

Comments `SlkSendPacket` stuffs the signature, client data size, and the checksum fields of the packet header. The caller must fill in all other packet header fields. If the transaction ID field is set to 0 (zero), the serial link manager automatically generates and stuffs a new nonzero transaction ID. The array of `SlkWriteDataType` structures enables the caller to specify the client data part of the packet as a list of noncontiguous blocks. The end of list is indicated by an array element with the `size` field set to 0 (zero). This call blocks until the entire packet is sent out or until an error occurs.

SlkSetSocketListener

Purpose Register a socket listener for a particular socket.

Prototype

```
Err SlkSetSocketListener (UInt socket,
                          SlkSocketListenPtr socketP)
```

Parameters

-> `socket`	Socket ID.
-> `socketP`	Pointer to a `SlkSocketListenType` structure.

Result

0	No error.
`slkErrBadParam`	Invalid parameter.
`slkErrSocketNotOpen`	The socket was not open.

Comments Called by applications to set up a socket listener.

Since the serial link manager does not make a copy of the `SlkSocketListenType` structure, but instead saves the passed pointer to it, the structure

- must **not** be an automatic variable (that is, local variable allocated on the stack)
- may be a global variable in an application
- may be a locked chunk allocated from the dynamic heap

The `SlkSocketListenType` structure specifies pointers to the socket listener procedure and the data buffers for dispatching packets destined for this socket. Pointers to two buffers must be specified: the packet header buffer (size of `SlkPktHeaderType`), and the packet body (client data) buffer. The packet body buffer must be large enough for the largest expected client data size. Both buffers may be application global variables or locked chunks allocated from the dynamic heap.

The socket listener procedure is called when a valid packet is received for the socket. Pointers to the packet header buffer and the packet body buffer are passed as parameters to the socket listener procedure.

Note: The application is responsible for freeing the `SlkSocketListenType` structure or the allocated buffers when the socket is closed. The serial link manager doesn't do it.

SlkSocketRefNum

Purpose Get the reference number of the communications library associated with a particular socket.

Prototype

```
ErrSlkSocketRefNum (UInt socket, UIntPtr refNumP)
```

Parameters

-> `socket`	The socket ID.
<-> `refNumP`	Pointer to location for returning the communications library reference number.

Result

0	No error.
`slkErrSocketNotOpen`	The socket was not open.

SlkSocketSetTimeout

Purpose Set the interbyte packet receive-timeout for a particular socket.

Prototype

```
Err SlkSocketSetTimeout (UInt socket,
                         Long timeout)
```

Parameters

-> `socket`	Socket ID.
-> `timeout`	Interbyte packet receive-timeout in system ticks.

Result

0	No error.
`slkErrSocketNotOpen`	The socket was not open.

Functions for Use By System Software Only

SlkSysPktDefaultResponse

Prototype

```
Err SlkSysPktDefaultResponse (
                    SlkPktHeaderPtr headerP,
                    void* bodyP)
```

WARNING: This function for use by system software only.

SlkProcessRPC

Prototype

```
Err SlkProcessRPC (SlkPktHeaderPtr headerP,
                   void* bodyP)
```

WARNING: This function for use by system software only.

Miscellaneous Communications Functions

Crc16CalcBlock

Purpose Calculate the 16-bit CRC of a data block using the table lookup method.

Prototype

```
Word Crc16CalcBlock (VoidPtr bufP,
                     UInt count,
                     Word crc)
```

Parameters

`bufP`	Pointer to the data buffer.
`count`	Number of bytes in the buffer.
`crc`	Seed CRC value.

Result A 16-bit CRC for the data buffer.

6

Palm OS Net Library

The Palm OS **net library** provides basic network services to applications. Using the net library, a Palm OS application can easily establish a connection with any other machine on the Internet and transfer data to and from that machine using the standard TCP/IP protocols.

The basic network services provided by the net library include:

- Stream-based, guaranteed delivery of data using TCP (Transmission Control Protocol).
- Datagram-based, best-effort delivery of data using UDP (User Datagram Protocol).

All higher-level Internet-based services (file transfer, e-mail, web browsing, etc.) can be implemented by applications on top of these basic data delivery services.

The application programming interface (API) for the net library is designed to be general enough to support almost any network protocol including Novell IPX, AppleTalk. Note, however, that currently only the TCP/IP protocols are implemented.

The API maps almost directly to the Berkeley UNIX sockets API, the de facto standard API for Internet applications. By including the appropriate header files, an application written to use the Berkeley sockets API can be compiled for the Palm OS with only slight (if any) changes to the source code.

Overview

This overview of the net library discusses the following topics:

- Structure
- System Requirements
- Constraints

Structure

The net library is implemented as a system library. System libraries are dynamically installed at runtime and don't always have to be present in the system. Since it is unclear whether all future platforms will need or want network support (especially devices with very limited amounts of memory), network support is an optional part of the operating system. As a result, systems which do not require network support will not pay any RAM penalty (for added entries in the system dispatch table, etc.).

The net library consists of two parts: a netlib interface and a net protocol stack. Neither part is actually linked in with an application. As a result, developers can update them as necessary in the future without having to recompile the applications that use them.

The **netlib interface** is the set of routines that an application calls directly when it makes a net library call. These routines execute in the caller's task like subroutines of the application. They are not linked in with the application, however, but are called through the library dispatch mechanism.

The **net protocol stack** runs as a separate task in the operating system. Inside this task, the TCP/IP protocol stack runs, and received packets are processed from the network device drivers. The netlib interface communicates with the net protocol stack through an operating system mailbox queue. It posts requests from applications into the queue and blocks until the net protocol stack processes the requests.

Having the net protocol stack run as a separate task has two big advantages:

- The operating system can switch in the net protocol stack to process incoming packets from the network even if one or more applications are currently busy.
- Even if an application is blocked waiting for some data to arrive off the network, the net protocol stack can continue to process requests for other applications.

System Requirements

The net library requires Palm OS 2.0 or better.

When the net library itself is opened, it requires an estimated additional 32 KB of RAM. This in effect doubles the overall system RAM requirements, currently 32 KB without the net library. It's therefore not practical to run the net library on any platform that has 128 KB or less of total RAM available since the system itself will consume 64 KB of RAM (leaving only 64 KB for user storage in a 128 KB system).

Constraints

Developers must keep in mind that Palm OS is designed for small devices with limited amounts of memory and other hardware resources. All applications written for Palm OS must pay special attention to memory and CPU usage. Devices that have the net library installed will most likely have only 64 KB of RAM available for system and applications. This does not include user storage RAM. When the net library is opened and initialized, the total remaining amount of RAM available to an application is approximately 14 KB.

The net library is built to allow a maximum of four open sockets at one time to keep the memory requirements of the net library to a minimum. Network applications have to be designed with this constraint in mind.

Network applications should also be careful about the amount of data they try to send to a remote host at the same time. When using TCP, the data that an application writes to a remote host is buffered in the dynamic heap so that control can be returned to the caller before the data is actually transmitted out over the network. Obviously, sending a 16 KB block of data to a remote host will severely tax the small dynamic memory space available to a Palm OS application. When an application tries to send a large block of data, the net library's send routines automatically buffer only a portion of the block of data, return the size of that portion to the caller, and rely on the caller to issue additional send calls to finish the transmission.

If an application expects to also receive data during a large transmission, it should therefore send a smaller block, then read back whatever is available to read before sending the next block. In this way, the amount of memory in the dynamic heap that must be used to buffer data waiting to send out and data waiting to be read back in by the application is kept to a minimum.

The Programmer's Interface

The net library API was designed in such a way that a program written to use the Berkeley sockets API can be compiled to use the net library API simply by including the appropriate header files. Little or no source code modification should be required. The `sys/socket.h` header file provided with the Palm OS SDK includes a set of macros that map Berkeley sockets calls directly to net library calls. That information is also included with the reference page for each function (See Chapter 7, "Net Library Functions,")

Net Library and Berkeley Sockets API: Differences

There are four main reasons why the net library API is slightly different from the sockets API.

- **Error Codes.** The sockets API by convention returns error codes in the application's global variable `errno`. The net API doesn't rely on any application global variables. This allows system code (which cannot have global variables) to use the net library API.
- **RefNum.** All library calls in the Palm OS must have the library reference number (`refnum`) as their first parameter.
- **Timeouts.** In a consumer system such as the Palm OS device, infinite timeouts don't work well because the end user can't "kill" a process that's stuck. A timeout parameter was therefore added to the API to allow the application to gracefully recover from hung connections.
- **Naming Conventions.** The naming conventions in the sockets API don't match the naming conventions of the Palm OS.

The main differences between the net library API and the Berkeley sockets API is that most net library API calls accept additional parameters for:

- A timeout
- The `refNum` of the net library
- The address for the return error code

The design of the Palm OS library manager requires that all library calls have the library `refNum` as the first parameter.

The macros in `sys/socket.h` do the following:

For...	The macros pass...
`refNum`	`AppNetRefnum` (application global variable).
timeout	`AppNetTimeout` (application global variable).
return error code	Address of the application global `errno`.

Example

The following example illustrates how the API mapping works for the Berkeley sockets call `socket()`, which has the calling convention:

```
int socket(int domain, int type, int protocol);
```

The equivalent net library call is `NetLibSocketOpen`, which has the calling convention:

```
NetSocketRef NetLibSocketOpen(
                      Word libRefnum,
                      NetSocketAddrEnum domain,
                      NetSocketTypeEnum type,
                      SWord protocol,
                      SDWord timeout,
                      Err* errP)
```

The macro for `socket` is:

```
#define socket(domain,type,protocol)\
NetLibSocketOpen(AppNetRefnum,
              domain,
              type,
              protocol,
              AppNetTimeout,
              &errno)
```

The macro in `sys/socket.h` for the `socket()` call passes:

- The application global `AppNetRefnum` as the `libRefnum.`
- The address of the application global `errno` for `errP.`

- A timeout value from the application global `AppNetTimeout`.

All other parameters are passed as is. Consequently, there is no extra layer of glue code penalty for using the sockets API instead of the net library API directly. Of course, an application that uses the sockets API with the Palm OS must declare and initialize the global variables `AppNetTimeout`, `AppNetRefnum`, and `errno` somewhere in its source code.

Using the Net Library

The net library can be thought of as having two groups of API calls: setup and configuration calls, and runtime calls. Normally, applications only use the runtime calls and leave all setup and configuration up to the net library preference panel.

Applications that need to use the net library should assume that all setup and configuration has occurred and focus on using the runtime calls.

An exception to this rule is applications that allow the user to select a particular "service" before trying to establish a connection. These kinds of applications present a pick list of service names and allow the user to select a service name. This functionality is provided via the net library preference panel. The panel provides action codes that allow an application to present a list of possible service names to let the end user pick one. The preference panel then makes the necessary net library setup and configuration calls to set up for that particular service.

This section first discusses Setup and Configuration Calls, then provides some detail on Runtime Calls.

Setup and Configuration Calls

The setup and configuration API calls of the net library are normally only used by the net library preference panel. This includes calls to set IP addresses, host name, domain name, login script, interface settings, and so on. Each setup and configuration call saves its settings in the net library preferences database in nonvolatile storage for later retrieval by the runtime calls.

Usually, the setup and configuration calls are made while the library is closed. A subset of the calls can also be issued while the library is open and will have real-time effects on the behavior of the library. Chapter 7, "Net Library Functions," describes the behavior of each call in more detail.

Interface Specific Settings

The net library configuration is structured so that network interface-specific settings can be specified for each network interface independently. These interface specific settings are called IF settings and are set and retrieved through the `NetLibIFSettingGet` and `NetLibIFSettingSet` calls.

- The `NetLibIFSettingGet` call takes a setting ID as a parameter along with a buffer pointer and buffer size for the return value of the setting. Some settings, like login script, are of variable size so the caller must be prepared to allocate a buffer large enough to retrieve the entire setting.
- The `NetLibIFSettingSet` call also takes a setting ID as a parameter along with a pointer to the new setting value and the size of the new setting.

General Settings

In addition to the interface-specific settings, there's a class of settings that don't apply to any one particular interface. These general settings are set and retrieved through the `NetLibSettingGet` and `NetLibSettingSet` calls. These calls take setting ID, buffer pointer, and buffer size parameters.

Settings for Interface Selection

Finally, there is a set of calls for specifying which interface(s) should be used by the net library. The `NetLibIFGet` call can be used to find out which interfaces are currently set up to be used by the library. The `NetLibIFAttach` and `NetLibIFDetach` can be used to attach and detach specific interfaces from the library.

These calls in particular can be called while the library is open or closed. If the library is open, the specific interface is attached or detached in real time. If the library is closed, the information is saved in preferences and used the next time the library is opened.

Summary

In summary, the preference panel needs to

- Set the general settings.
- Attach the appropriate network interfaces.
- Set the network specific settings for each interface.

The order in which this is done is not important since nothing is done with the settings until the library is opened. The API description for each of the configuration calls lists in detail the possible setting values for each call, which are required or optional, and the default values for each setting.

Runtime Calls

Most applications will use only the net library runtime calls. Most of these calls have an equivalent function in the Berkeley sockets API. The `sys/socket.h` header file allows source code written to the Berkeley sockets API to be compiled directly for the Palm OS.

There is, however, some additional setup and shutdown code that every Palm OS application must have in order to use the net library. Because of the limited resources in the Palm OS environment, the net library was designed so that it only takes up extra memory from the system when an application is running that actually needs to use its services. An Internet application must therefore inform the system when it needs to use the net library by opening the net library when it starts up and by closing it when it exits.

Initialization and Shutdown

The following calls are available to open and close the net library:

- Calls Made Before Opening the Net Library
- Opening the Net Library
- Closing the Net Library

Calls Made Before Opening the Net Library

Most net library calls don't work before the library is opened. An exception to this rule are calls that specify which network interface(s) to use, and the calls for setting the net library settings and the settings for the network interfaces. These calls are `NetLibIFGet`, `NetLibIFAttach`, `NetLibIFDetach`, `NetLibIFSettingGet`, `NetLibIFSettingSet`, `NetLibSettingGet`, and `NetLibSettingSet` (see also Setup and Configuration Calls). All of these calls save the settings in the net library Preferences database used by `NetLibOpen` to initialize the library and establish the connection.

It's expected that most applications won't need to use these calls because the network preferences panel is responsible for configuring the net library.

Opening the Net Library

An application can call `NetLibOpen` to open the net library. Before the net library is opened, most calls issued to it fail with a `netErrNotOpen` error code.

If the net library is not already open for another application, `NetLibOpen` starts up the net protocol stack task, allocates memory for internal use by the net library, and brings up the network connection. Most likely, the user has configured the Palm OS device to establish a SLIP or PPP connection through a modem and in this type of setup, `NetLibOpen` dials up the modem and establishes the connection before returning.

If the net library is already open when `NetLibOpen` is called, it simply increments the open count and returns immediately.

Note that the `NetLibOpen` call may bring up UI elements to display connection progress information, depending on which network interfaces it is using. Because of this, the caller must call `NetLibOpen` from the main UI task (that is, the main event loop of an application) and not from a background task.

Closing the Net Library

Before an application quits, or if it no longer needs to do network I/O, it should call `NetLibClose`.

`NetLibClose` decrements the open count. If the open count has reached 0, `NetLibClose` schedules a timer to shut down the net library unless another `NetLibOpen` is issued before the timer expires. The close timer allows the user to quit from one network application and launch another application within a certain time period without having to wait for another network connection establishment.

If `NetLibOpen` is called before the close timer expires, it simply cancels the timer and marks the library as fully open with an open count of 1 before returning. If the timer expires before another `NetLibOpen` is issued, all existing network connections are brought down, the net protocol stack task is terminated, and all memory allocated for internal use by the net library is freed.

Summary of Initialization

In summary, any application that needs to do network I/O should always call `NetLibOpen` first and `NetLibClose` before it quits. The details of whether or not a connection needs to be established or brought down are automatically handled by the library.

Note that all net library calls, including `NetLibOpen` and `NetLibClose` require the `refNum` of the net library as their first parameter. To find this `refNum`, call `SysLibFind`, passing the name of the net library, "`Net.lib`". In addition, if the application is using the sockets API macros, it must save this `refnum` in the application global variable `AppNetRefnum`.

Initialization Example

The following example code fragment illustrates how to find the net library's `refnum` and then open the library. Note that if the net library is not installed on the Palm OS device (on a pre-2.0 ROM, or a 128Kb machine for example), `SysLibFind` returns an error code.

```
#include <sys/socket.h>
....
err = SysLibFind("Net.lib", &AppNetRefnum);
if (err) {/* error handling here */}
err = NetLibOpen(AppNetRefnum, &ifErrs);
if (err || ifErrs) {/* error handling here */}
```

Once the net library has been opened, sockets can be opened and data sent to and received from remote hosts using either the Berkeley sockets API, or the native net library API. The following example code fragment shows how to close down the net library when an application exits or no longer needs network support:

```
err = NetLibClose(AppNetRefnum, false);
```

Version Checking

Besides using `SysLibFind` to determine if the net library is installed, an application can also look for the net library version feature. This feature is only present if the net library is installed. This feature can be used to get the version number of the net library as follows:

```
DWord version;
err = FtrGet(netFtrCreator, netFtrNumVersion,
            &version);
```

If the net library is not installed, `FtrGet` returns a non-zero result code.

The version number is encoded in the format `0xMMmfsbbb`, where:

MM	major version
m	minor version
f	bug fix level
s	stage: 3-release, 2-beta, 1-alpha, 0-development
bbb	build number for non-releases

For example:

V1.1.2b3 would be encoded as 0x01122003

V2.0a2 would be encoded as 0x02001002

V1.0.1 would be encoded as 0x01013000

This document describes version 1.0 of the net library (0x01003000).

Network I/O and Utility Calls

Because of the close correlation with the Berkeley sockets API, the reader is referred to one of the many books written on network communications for an explanation of how to use the remaining calls in the net library. Where applicable, the detailed function explanations in Net Library Functions provide the equivalent sockets API call for each native net library call.

Note that because the Berkeley sockets API requires some application global variables and glue code, an application written for this API must link with the module `"NetSocket.c"`, which is included as part of the Palm OS SDK. The following is a summary of the mappings from the Berkeley sockets API to the native net library API.

Berkeley Sockets API	Net Library
`accept`	`NetLibSocketAccept`
`bcopy`	`MemMove`
`bzero`	`MemSet`
`bcmp`	`MemCmp`
`bind`	`NetLibSocketBind`
`close`	`NetLibSocketClose`
`connect`	`NetLibSocketConnect`
`fcntl`	`NetLibSocketOptionSet/NetLibSocketOptionGet (...,netSocketOptSockNonBlocking,...)`
`getdomainname`	`NetLibSocketOptionGet(..,netSettingDomainName,...)`
`gethostbyaddr`	`NetLibGetHostByAddr`
`gethostbyname`	`NetLibGetHostByName`

Berkeley Sockets API	Net Library
gethostname	NetLibSettingGet(..,netSettingHostName,...)
getpeername	NetLibSocketAddr
getservbyname	NetLibGetServByName
getsockname	NetLibSocketAddr
getsockopt	NetLibSocketOptionGet
gettimeofday	glue code using TimGetSeconds() (see Part II)
htonl	macro
htons	macro
inet_addr	NetLibAddrAToIN
inet_lnaof	glue code
inet_makeaddr	glue code
inet_netof	glue code
inet_network	glue code
inet_ntoa	NetLibAddrINToA
listen	NetLibSocketListen
ntohl	macro
ntohs	macro
read	NetLibReceive
recv	NetLibReceive
recvfrom	NetLibReceive

Berkeley Sockets API	**Net Library**
recvmsg	NetLibReceivePB
send	NetLibSend
sendmsg	NetLibSendPB
sendto	NetLibSend
setsockopt	NetLibSocketOptionSet
shutdown	NetLibSocketShutdown
sleep	SysTaskDelay
socket	NetLibSocketOpen
select	NetLibSelect
setdomainname	NetLibSettingSet(..,netSettingDomainName,...)
sethostname	NetLibSettingSet(..,netSettingHostName,...)
settimeofday	glue code using TimSetSeconds() (see Part II)
write	NetLibSend

7

Net Library Functions

This chapter lists the calls available in the net library and their Berkeley sockets equivalents. Each call has a *purpose* section which gives a short description of what the call does; a *prototype* section identifies the parameters to the call and their types; a *parameters* section lists detailed information about each of the parameters; a *result* section identifies the possible return codes; a *sockets API equivalent* section gives the name of the corresponding sockets API call; and a *comments* section gives a more detailed description of the call.

The functions are grouped as follows:

Library Open and Close

NetLibClose

Purpose Closes the net library.

Prototype `Err NetLibClose (Word libRefnum, Word immediate)`

Parameters

-> `libRefnum`	Reference number of the net library.
-> `immediate`	If `TRUE`, library will shut down immediately. If `FALSE`, library will shut down only if close timer expires before another `NetLibOpen` is issued.

Result Codes

0	No error.
`netErrNotOpen`	Library was not open.
`netErrStillOpen`	Not really an error; returned if library is still in use by another application.

Sockets Equivalent None.

Comments Applications must call this function when they no longer need the net library. If the net library open count is greater than 1 before this call is made, the count is decremented and `netErrStillOpen` is returned. If the open count was 1, the library takes the following action:

- If `immediate` is `TRUE`, the library shuts down immediately. All network interfaces are brought down, the net protocol stack task is terminated, and all memory used by the net library is freed.
- If `immediate` is `FALSE`, a close timer is created and this call returns immediately without actually bringing the net library down. Instead it leaves it up and running but marks it as in the “close-wait” state. It remains in this state until

either the timer expires or another `NetLibOpen` is issued. If the timer expires, the library is shut down. If another `NetLibOpen` call is issued before the timer expires (possibly by another application), the timer is cancelled and the library is marked as fully open.

It is expected that most applications will pass `FALSE` for `immediate`. This allows the user to quit one Internet application and launch another within a short period of time without having to wait through the process of closing down and then re-establishing dial-up network connections.

See Also `NetLibOpen`, `NetLibOpenCount`

NetLibConnectionRefresh

Purpose This routine is a convenience call for applications. It checks the status of all connections and optionally tries to open any that were closed.

Prototype

```
Err NetLibConnectionRefresh (Word refNum,
                             Boolean refresh,
                             BooleanPtr allInterfacesUpP,
                             WordPtr netIFErrP)
```

Parameters

`refnum`	Reference number of the net library.
`refresh`	If `TRUE`, any connections that aren't currently open are opened.
`allInterfacesUpP`	Set to `TRUE` if all connections are open.
`netIFErrP`	First error encountered when reopening connections that were closed.

Result Codes

0	No error.

Sockets Equivalent None.

Comments This function determines whether a connection is up based on the internal status of the TCP/IP stack. To test the presence of a "physical connection" (phone line, modem, serial cable), a command should be sent once it's been determined that the logical connection is up. If the physical connection is broken, nothing returns, and a timeout error eventually occurs.

NetLibFinishCloseWait

Purpose Forces the net library to do a complete close if it's currently in the close-wait state.

Prototype `Err NetLibFinishCloseWait (Word libRefnum)`

Parameters

-> `libRefnum`	Reference number of the net library.

Result Codes

0	No error.

Sockets Equivalent None.

Comments This call checks the current open state of the net library. If it's in the close-wait state (see `NetLibClose`), it forces the library to perform an immediate, complete close operation.

This call will most likely only be used by the preferences panel that configures the net library.

NetLibOpen

Purpose Opens and initializes the net library.

Prototype

```
Err NetLibOpen (Word libRefnum,
                WordPtr netIFErrP)
```

Parameters

-> `libRefnum` Reference number of the net library.

-> `netIFErrP` Pointer to return error code for interfaces.

Result

0 No error.

`netErrAlreadyOpen`
Not really an error; returned if library was already open and the open count was simply incremented.

`netErrOutOfMemory`
Not enough memory available to open the library.

`netErrNoInterfaces`
Incorrect setup.

`netErrPrefNotFound`
Incorrect setup.

Comments Applications must call this function before using the net library. If the net library was already open, `NetLibOpen` increments its open count. Otherwise, it opens the library, initializes it, starts up the net protocol stack component of the library as a separate task, and brings up all attached network interfaces.

`NetLibOpen` uses settings saved in the net library's preferences database during initialization. These settings include the interfaces to attach, the IP addresses, etc. It's assumed that these settings have been previously set up by a preference panel or equivalent so an application doesn't normally have to set them up before calling `NetLibOpen`.

If the end user has configured the Palm OS device to connect through a dialup interface, there's a good chance that the interface will display a progress dialog as it establishes a connection. For this reason, `NetLibOpen` **must** be called from the main UI task (an application's main event loop), and **not** from a separate background task.

If any of the attached interfaces fails to come up, `*netIFErrP` will contain the error number of the first interface that encountered a problem.

It's possible, and quite likely, that the net library will be able to open even though one or more interfaces failed to come up (due to bad modem settings, service down, etc). Some applications may therefore wish to close the net library using `NetLibClose` if `*netIFErrP` is non-zero and display an appropriate message for the user. If an application needs more detailed information, e.g. which interface(s) in particular failed to come up, it can loop through each of the attached interfaces and ask each one if it is up or not. Use the following calls to accomplish this:

- `NetLibIFGet(...)`,
- `NetLibIFSettingGet(..., netIFSettingUp, ...)`
- `NetLibIFSettingGet(..., netIFSettingName,...)`

See Also `SysLibFind`, `NetLibClose`, `NetLibOpenCount`

NetLibOpenCount

Purpose Retrieves the open count of the net library.

Prototype

```
Err NetLibOpenCount (Word libRefnum,
                     WordPtr countP)
```

Parameters

-> `libRefnum`	Reference number of the net library.
<- `countP`	Pointer to return count variable.

Result Codes

0	No error.

Sockets Equivalent None.

Comments This call will most likely only be used by the Network preference panel. Most applications will simply call `NetLibOpen` unconditionally during startup and `NetLibClose` when they exit.

Socket Creation and Deletion

NetLibSocketClose

Purpose

Close a socket.

Prototype

```
SWord NetLibSocketClose (Word libRefnum,
                         NetSocketRef socketRef,
                         SDWord timeout,
                         Err* errP)
```

Parameters

-> libRefNum	Reference number of the net library.
-> socketRef	SocketRef of the open socket.
-> timeout	Maximum timeout in system ticks, -1 means wait forever.
<- errP	Address of variable used to return error code.

Result Codes

0	No error.
-1	Error occurred. Error code in *errP.

Errors

0	No error.
netErrTimeout	Call timed out.
netErrNotOpen	
netErrParamErr	
netErrSocketNotOpen	

Sockets Equivalent

```
int close(int socket);
```

Comments

Closes down a socket and frees all memory associated with it.

See Also

NetLibSocketOpen, NetLibSocketShutdown

NetLibSocketOpen

Purpose Open a new socket.

Prototype

```
NetSocketRef NetLibSocketOpen (Word libRefnum,
                   NetSocketAddrEnum domain,
                   NetSocketTypeEnum type,
                   SWord protocol,
                   Long timeout, Err* errP)
```

Parameters

-> `libRefNum`	Reference number of the net library.
-> `domain`	Address domain. Only `netSocketAddrINET` is currently supported.
-> `type`	Desired type of connection, either `netSocketTypeStream` or `netSocketTypeDatagram`. `netSocketTypeRaw` is **not** currently supported.
-> `protocol`	Protocol to use. Currently ignored for the `netSocketAddrINET` domain.
-> `timeout`	Maximum timeout in system ticks, -1 means wait forever.
<- `errP`	Address of variable used to return error code.

Result Codes

>= 0	Socket `refNum` of open socket.
-1	Error occurred, error code in `*errP`.

Errors

0	No error.
`netErrTimeout`	
`netErrNotOpen`	
`netErrParamErr`	
`netErrNoMoreSockets`	

Sockets Equivalent

```
int socket(int domain, int type, int protocol);
```

Comments

Allocates memory for a new socket and opens it.

Note that only stream-based and datagram-based sockets are supported. Raw sockets, in particular, are **not** currently supported.

See Also

`NetLibSocketClose`

Socket Options

NetLibSocketOptionGet

Purpose Retrieves the current value of a socket option.

Prototype

```
SWord NetLibSocketOptionGet (Word libRefnum,
                             NetSocketRef socket,
                             Word level,
                             Word option,
                             VoidPtr optValueP,
                             WordPtr optValueLenP,
                             SDWord timeout,
                             Err* errP)
```

Parameters

-> `libRefNum`	Reference number of the net library.
-> `socket`	`SocketRef` of the open socket.
-> `level`	Level of the option, one of the `netSocketOptLevelXXX` enum constants.
-> `option`	One of the `netSocketOptXXX` enum constants.
-> `optValueP`	Pointer to variable holding new value of option.
<-> `optValueLenP`	Size of variable pointed to by `optValueP` on entry. Actual size of return value on exit.
-> `timeout`	Maximum timeout in system ticks; -1 means wait forever.
<- `errP`	Address of variable used to return error code.

Result Codes

0	No error.
-1	Error occurred, error code in `*errP`.

Errors

0	No error.
`netErrTimeout`	
`netErrNotOpen`	
`netErrParamErr`	
`netErrSocketNotOpen`	
`netErrUnimplemented`	
`netErrWrongSocketType`	

Sockets Equivalent

```
int getsockopt (int socket, int level,
                int option, const void*
                optValueP, int* optValueLenP);
```

Comments

Returns the current value of a socket option. The caller passes a pointer to a variable to hold the returned value (`in optValueP`) and the size of this variable (`in *optValueLenP`). On exit, `*optValueP` is updated with the actual size of the return value.

For all of the fixed size options (every option except `netSockOptIPOptions`), `*optValueLenP` is unmodified on exit and this call does its best to return the value in the caller's desired type size.

For compatibility with existing Internet applications, this call is quite flexible on the `*optValueLenP` parameter. If the desired type for an option is `FLAG`, this call supports an `*optValueLenP` of 1, 2, or 4. If the desired type for an option is `int`, it supports an `*optValueLenP` of 2 or 4.

See `NetLibSocketOptionSet` for a list of available options.

See Also

`NetLibSocketOptionSet`

NetLibSocketOptionSet

Purpose Set a socket option.

Prototype

```
SWord NetLibSocketOptionSet (Word libRefnum,
                  NetSocketRef socketRef,
                  Word level,
                  Word option,
                  VoidPtr optValueP,
                  Word optValueLen,
                  SDWord timeout,
                  Err* errP)
```

Parameters

-> `libRefNum`	Reference number of the net library.
-> `socketRef`	`SocketRef` of the open socket.
-> `level`	Level of the option, one of the `netSocketOptLevelXXX` enum constants.
-> `option`	One of the `netSocketOptXXX` enum constants.
-> `optValueP`	Pointer to the variable holding the new value of the option.
-> `optValueLen`	Size of variable pointed to by `optValueP.`
-> `timeout`	Maximum timeout in system ticks; -1 means wait forever.
<- `errP`	Address of variable used to return error code.

Result Codes

0	No error.
-1	Error occurred, error code in *`errP`.

Errors

0	No error.
`netErrTimeout`	Call timed out.
`netErrNotOpen`	
`netErrParamErr`	

`netErrSocketNotOpen`

`netErrUnimplemented`

`netErrWrongSocketType`

Sockets Equivalent

```
int setsockopt (int socketRef,
                int level, int option,
                const void* optValueP,
                int optValueLen);
```

Comments

Sets various options associated with a socket. The caller passes a pointer to the new option value in `optValueP` and the size of the option in `optValueLen`.

The following table lists the available options.

- The Level column specifies the option level, which is one of the `netSocketOptLevelXXX` constants.
- The Option column lists the option, which is one of the `netSocketOptXXX` constants.
- The G/S column lists whether this option can be fetched with the `NetLibSocketOptionGet` call (G) and/or set (S) with this call.
- The type column lists the type of the option.
- The I column specifies whether or not this option is currently implemented.

Level	Option	G/S	Type	I	Description
IP	IPOptions	GS	Byte[]	N	Options in IP Header
TCP	TCPNoDelay	GS	FLAG	Y	Don’t delay send to coalesce packets
TCP	TCPMaxSeg	G	int	Y	Get TCP maximum segment size
Socket	SockDebug	GS	FLAG	N	Turn on recording of debug info

Level	Option	G/S	Type	I	Description
Socket	SockAcceptConn	G	FLAG	N	Socket has had listen
Socket	SockReuseAddr	GS	FLAG	N	Allow local address reuse
Socket	SockKeepAlive	GS	FLAG	Y	Keep connections alive
Socket	SockDontRoute	GS	FLAG	N	Just use interface addresses
Socket	SockBroadcast	GS	FLAG	N	Permit sending of broadcast messages
Socket	SockUseLoopback	GS	FLAG	N	Bypass hardware when possible
Socket	SockLinger	GS	NetSocketLinger	Y	Linger on close if data present
Socket	SockOOBInLine	GS	FLAG	N	Leave received OOB data inline
Socket	SockSndBufSize	GS	int	N	Send buffer size
Socket	SockRcvBufSize	GS	int	N	Receive buffer size
Socket	SockSndLowWater	GS	int	N	Send low-water mark
Socket	SockRcvLowWater	GS	int		Receive low-water mark
Socket	SockSndTimeout	GS	int	N	Send timeout
Socket	SockRcvTimeout	GS	int	N	Receive timeout
Socket	SockErrorStatus	G	int	Y	Get error status and clear
Socket	SockSocketType	G	int	Y	Get socket type
Socket	SockNonBlocking	GS	FLAG	Y	Set non-blocking mode on/off

For compatibility with existing Internet applications, this call is quite flexible on the `optValueLen` parameter. If the desired type for an option is `FLAG`, this call accepts an `optValueLen` of 1, 2, or 4. If the desired type for an option is `int`, it accepts an `optValueLen` of 2 or 4.

Except for the `SockNonBlocking` option, all options listed above have equivalents in the sockets API. The `SockNonBlocking` option was added to this call in the net library in order to implement the functionality of the UNIX `fcntl()` control call, which can be used to turn nonblocking mode on and off for sockets.

See Also `NetLibSocketOptionGet`

Socket Connections

NetLibSocketAccept

Purpose Accept a connection from a stream-based socket.

Prototype

```
SWord NetLibSocketAccept(Word libRefnum,
            NetSocketRef socketRef,
            NetSocketAddrType* remAddrP,
            SWord* remAddrLenP,
            Long timeout,
            Err* errP)
```

Parameters

-> `libRefNum`	Reference number of the net library.
-> `socketRef`	`SocketRef` of the open socket.
<- `remAddrP`	Address of remote host is returned here.
<->`remAddrLenP`	On entry, length of `remAddrP` buffer in bytes. On exit, length of returned address stored in `*remAddrP`.
-> `timeout`	Maximum timeout in system ticks, -1 means wait forever.
<- `errP`	Address of variable used to return error code.

Result Codes

>=0	`NetSocketRef` of new socket.
-1	Error occurred, error code in `*errP`.

Errors

0	No error.
`netErrTimeout`	Call timed out.
`netErrNotOpen`	
`netErrParamErr`	
`netErrSocketNotOpen`	
`netErrNotConnected`	
`netErrClosedByRemote`	
`netErrWrongSocketType`	
`netErrSocketNotListening`	

Sockets Equivalent

```
int accept (int socket,
            void* sockAddrP,
            int* addrLenP);
```

Comments

Accepts the next connection request from a remote client. This call is only applicable to stream-based sockets. Before calling `NetLibSocketAccept` on a socket, a server application needs to:

- Open the socket (`NetLibSocketOpen`).
- Bind the socket to a local address (`NetLibSocketBind`).
- Set the maximum pending connection-request queue length (`NetLibSocketListen`).

`NetLibSocketAccept` will block until a successful connection request is obtained from a remote client. After a successful connection is made, this call returns with the address of the remote host in `*remAddrP` and the `socketRef` of a **new** socket as the return value.

See Also

`NetLibSocketBind`, `NetLibSocketListen`

NetLibSocketAddr

Purpose Returns the local and remote addresses currently associated with a socket.

Prototype

```
SWord NetLibSocketAddr (Word libRefnum,
                        NetSocketRef socketRef,
                        NetSocketAddrType* locAddrP,
                        SWord* locAddrLenP,
                        NetSocketAddrType* remAddrP,
                        SWord* remAddrLenP,
                        SDWord timeout,
                        Err* errP)
```

Parameters

-> `libRefNum`	Reference number of the net library.
-> `socketRef`	`SocketRef` of the open socket.
<- `locAddrP`	Local address of socket is returned here.
<->`locAddrLenP`	On entry, length of `locAddrPbuffer` in bytes. On exit, length of returned address stored in *`locAddrP`.
<- `remAddrP`	Address of remote host is returned here.
<->`remAddrLenP`	On entry, length of `remAddrP` buffer in bytes. On exit, length of returned address stored in *`remAddrP`.
-> `timeout`	Maximum timeout in system ticks, -1 means wait forever.
<- `errP`	Address of variable used to return error code.

Result Codes

0	No error.
-1	Error occurred, error code in `*errP.`

Errors

0	No error.
`netErrTimeout`	Call timed out.
`netErrNotOpen`	
`netErrParamErr`	
`netErrSocketNotOpen`	
`netErrClosedByRemote`	

Sockets Equivalent

```
int getpeername (int s,
                 struct sockaddr* name,
                 int* namelen);
int getsockname (int s,
                 struct sockaddr* name,
                 int* namelen);
```

Comments

This call is mainly useful for stream-based sockets. It allows the caller to find out what address was bound to a connected socket and the address of the remote host that it's connected to.

See Also

`NetLibSocketBind`, `NetLibSocketConnect`, `NetLibSocketAccept`

NetLibSocketBind

Purpose Assign a local address to a socket.

Prototype

```
SWord NetLibSocketBind (Word libRefnum,
                  NetSocketRef socketRef,
                  NetSocketAddrType* socketAddrP,
                  SWord addrLen,
                  Long timeout,
                  Err* errP)
```

Parameters

-> `libRefNum`	Reference number of the net library.
-> `socketRef`	`SocketRef` of the open socket.
-> `sockAddrP`	Pointer to address.
-> `addrLen`	Length of address in *`sockAddrP`.
-> `timeout`	Maximum timeout in system ticks; -1 means wait forever.
<- `errP`	Address of variable used to return error code.

Result Codes

0	No error.
-1	Error occurred, error code in `*errP`.

Errors

0	No error.
`netErrTimeout`	Call timed out.
`netErrNotOpen`	
`netErrParamErr`	
`netErrSocketNotOpen`	
`netErrAlreadyConnected`	
`netErrClosedByRemote`	

Sockets Equivalent

```
int bind (int socket,
          const void* sockAddrP,
          int addrLen);
```

Comments Applications that want to wait for an incoming connection request from a remote host must call this function. After calling `NetLibSocketBind`, applications can call `NetLibSocketListen` and then `NetLibSocketAccept` to make the socked ready to accept connection requests.

See Also `NetLibSocketConnect`, `NetLibSocketListen`, `NetLibSocketAccept`

NetLibSocketConnect

Purpose Assign a destination address to a socket and initiate three-way handshake if it's stream based.

Prototype

```
SWord NetLibSocketConnect (Word libRefnum,
                  NetSocketRef socketRef,
                  NetSocketAddrType* socketAddrP,
                  SWord addrLen,
                  Long timeout,
                  Err* errP)
```

Parameters

-> `libRefNum`	Reference number of the net library.
-> `socketRef`	`SocketRef` of the open socket.
-> `sockAddrP`	Pointer to address.
-> `addrLen`	Length of address in `*sockAddrP.`
-> `timeout`	Maximum timeout in system ticks; -1 means wait forever.
<- `errP`	Address of variable used to return error code.

Result Codes

0	No error.
-1	Error occurred, error code in `*errP`.

Errors

0	No error.
`netErrTimeout`	Call timed out.
`netErrNotOpen`	

netErrParamErr

netErrSocketNotOpen

netErrSocketBusy

netErrNoInterfaces

netErrPortInUse

netErrQuietTimeNotElapsed

netErrInternal

netErrAlreadyConnected

netErrClosedByRemote

netErrTooManyTCPConnections

Sockets Equivalent

```
int connect (int socket,
             const void* sockAddrP,
             int addrLen);
```

See Also NetLibSocketBind

NetLibSocketListen

Purpose Put a stream-based socket into passive listen mode.

Prototype

```
SWord NetLibSocketListen(Word libRefnum,
                         NetSocketRef socketRef,
                         Word queueLen,
                         Long timeout,
                         Err* errP)
```

Parameters

-> `libRefNum`	Reference number of the net library.
-> `socketRef`	`SocketRef` of the open socket.
-> `queueLen`	Maximum number of pending connections allowed.
-> `timeout`	Maximum timeout in system ticks, -1 means wait forever.
<- `errP`	Address of variable used to return error code.

Result Codes

0	No error.
-1	Error occurred, error code in *`errP`.

Errors

0	No error.
`netErrTimeout`	`Call timed out.`
`netErrNotOpen`	
`netErrParamErr`	
`netErrOutOfResources`	
`netErrSocketNotOpen`	
`netErrSocketBusy`	
`netErrNoInterfaces`	
`netErrPortInUse`	
`netErrInternal`	
`netErrAlreadyConnected`	
`netErrClosedByRemote`	
`netErrWrongSocketType`	

Sockets Equivalent

```
int listen (int socket, int queueLen);
```

Comments

Sets the maximum allowable length of the queue for pending connections. This call is only applicable to `NetLibSocketAccept` sockets.

After a socket is created and bound to a local address using `NetLibSocketBind`, a server application can call `NetLibSocketListen` and then `NetLibSocketAccept` to accept connections from remote clients.

The `queueLen` is currently quietly limited to 1 (higher values are ignored).

See Also

`NetLibSocketBind`, `NetLibSocketAccept`

NetLibSocketShutdown

Purpose Shut down a socket in one or both directions.

Prototype

```
SWord NetLibSocketShutdown (Word libRefnum,
                    NetSocketRef socketRef,
                    SWord direction,
                    SDWord timeout,
                    Err* errP)
```

Parameters

-> `libRefNum`	Reference number of the net library.
-> `socketRef`	`SocketRef` of the open socket.
-> `direction`	Direction to shut down. One of the `NetSocketDirXXX` enum constants.
-> `timeout`	Maximum timeout in system ticks; -1 means wait forever.
<- `errP`	Address of variable used to return error code.

Result Codes

0	No error.
-1	Error occurred, error code in `*errP`.

Errors

0	No error.
`netErrTimeout`	Call timed out.
`netErrNotOpen`	
`netErrParamErr`	
`netErrSocketNotOpen`	

Sockets Equivalent

```
int shutdown (int socket, int direction);
```

Comments Shuts down communication in one or both directions on a socket. Direction can be `netSocketDirInput`, `netSocketDirOutput`, or `netSocketDirBoth`.

If direction is `netSocketDirInput`, the socket is marked as down in the receive direction and further read operations from it return a `netErrSocketInputShutdown` error.

Send and Receive Routines

NetLibDmReceive

Purpose Receive data from a socket directly into a database record.

Prototype

```
SWord NetLibDmReceive(Word libRefNum,
                      NetSocketRef socket,
                      VoidPtr recordP,
                      ULong recordOffset,
                      Word rcvLen,
                      Word flags,
                      VoidPtr fromAddrP,
                      WordPtr fromLenP,
                      Long timeout,
                      Err* errP)
```

Parameters

-> `libRefNum`	Reference number of the net library.
-> `socket`	`SocketRef` of the open socket.
-> `recordP`	Pointer to beginning of record.
-> `recordOffset`	Offset from beginning of record to read data into.
-> `rcvLen`	Maximum number of bytes to read.
-> `flags`	One or more `netMsgFlagXXX` flag.
-> `fromAddrP`	Pointer to buffer to hold address of sender (`NetSocketAddrType`).
<-> `fromLenP`	On entry, size of `fromAddrP` buffer. On exit, actual size of returned address in `fromAddrP`.

-> `timeout`	Maximum timeout in system ticks, -1 means wait forever.
<- `errP`	Address of variable used to return error code.

Result Codes

0	Socket has been shut down by remote host.
>0	Number of bytes successfully received.
-1	Error occurred, error code in `*errP`.

Errors

0	No error.
`netErrTimeout`	Call timed out.
`netErrNotOpen`	
`netErrParamErr`	
`netErrSocketNotOpen`	
`netErrWouldBlock`	

Comments This call behaves similarly to `NetLibReceive` but reads the data directly into a database record, which is normally write-protected. The caller must pass a pointer to the start of the record and an offset into the record of where to start the read.

NetLibReceive

Purpose Receive data from a socket into a single buffer.

Prototype

```
SWord NetLibReceive (Word libRefNum,
                     NetSocketRef socket,
                     VoidPtr bufP,
                     Word bufLen,
                     Word flags,
                     VoidPtr fromAddrP,
                     WordPtr fromLenP,
                     Long timeout,
                     Err* errP);
```

Parameters

-> `libRefNum`	Reference number of the net library.
-> `socket`	`SocketRef` of the open socket.
-> `bufP`	Pointer to buffer to hold received data.
-> `bufLen`	Length of `bufP` buffer.
-> `flags`	One or more `netMsgFlagXXX` flag.
-> `fromAddrP`	Pointer to buffer to hold address of sender (`NetSocketAddrType`).
<-> `fromLenP`	On entry, size of `fromAddrP` buffer. On exit, actual size of returned address in `fromAddrP`.
-> `timeout`	Maximum timeout in system ticks; -1 means wait forever.
<- `errP`	Address of variable used to return error code.

Result Codes

0	Socket has been shut down by remote host.
>0	Number of bytes successfully received,
-1	Error occurred, error code in `*errP.`

Errors

0	No error.
`netErrTimeout`	Call timed out.

`netErrNotOpen`

`netErrParamErr`

`netErrSocketNotOpen`

`netErrWouldBlock`

Sockets Equivalent

```
int recvfrom ( int socket, const void* bufP,
               int bufLen, int flags,
               const void* fromAddrP,
               int* fromLenP);

int recv     ( int socket, const void* bufP,
               int bufLen, int flags);

int read     ( int socket, const void* bufP,
               int bufLen);
```

Comments

For stream-based sockets, this call reads whatever bytes are available and returns the number of bytes actually read into the caller's buffer. If there is no data available, this call will block until at least 1 byte arrives, until the socket is shut down by the remote host, or until a timeout occurs.

For datagram-based sockets, this call reads a complete datagram and returns the number of bytes in the datagram. If the caller's buffer is not large enough to hold the entire datagram, the end of the datagram is discarded. If a datagram is not available, this call will block until one arrives, or until the call times out.

The data is read into a single buffer pointed to by `bufP`.

See Also

`NetLibReceive`, `NetLibDmReceive`, `NetLibSend`, `NetLibSendPB`

NetLibReceivePB

Purpose Receive data from a socket into a gather-read array.

Prototype

```
SWord NetLibReceivePB (Word libRefnum,
                       NetSocketRef socket,
                       NetIOParamType* pbP,
                       Word flags,
                       Long timeout,
                       Err* errP)
```

Parameters

-> `libRefNum`	Reference number of the net library.
-> `socketRef`	`SocketRef` of the open socket.
-> `pbP`	Pointer to parameter block containing buffer info.
-> `flags`	One or more `netMsgFlagXXX` flag.
-> `timeout`	Maximum timeout in system ticks, -1 means wait forever.
<- `errP`	Address of variable used to return error code.

Result Codes

0	Socket has been shut down by remote host.
>0	Number of bytes successfully received.
-1	Error occurred, error code in *`errP`.

Errors

0	No error.
`netErrTimeout`	Call timed out.
`netErrNotOpen`	
`netErrParamErr`	
`netErrSocketNotOpen`	
`netErrWouldBlock`	

Sockets Equivalent

```
int recvmsg (int socket,
             const struct msghdr* pbP,
             int flags);
```

Comments

For stream-based sockets, this call reads whatever bytes are available and returns the number of bytes actually read into the caller's buffer. If no data is available, this call will block until at least 1 byte arrives, until the socket is shut down by the remote host, or until a timeout occurs.

For datagram-based sockets, this call reads a complete datagram and returns the number of bytes in the datagram. If the caller's buffer is not large enough to hold the entire datagram, the end of the datagram is discarded. If a datagram is not available, this call will block until one arrives, or until the call times out.

The data is read into the gather-read array specified by the `pbP->iov` array.

See Also

`NetLibReceive`, `NetLibDmReceive`, `NetLibSend`, `NetLibSendPB`

NetLibSend

Purpose

Send data to a socket from a single buffer.

Prototype

```
SWord NetLibSend (Word libRefNum,
                  NetSocketRef socket,
                  const VoidPtr bufP,
                  Word bufLen,
                  Word flags,
                  VoidPtr toAddrP,
                  Word toLen,
                  Long timeout,
                  Err* errP)
```

Parameters

-> `libRefNum`	Reference number of the net library.
-> `socket`	`SocketRef` of the open socket.

-> `bufP`	Pointer to data to write.
-> `bufLen`	Length of data to write
-> `flags`	One or more of `netMsgFlagXXX` flags.
-> `toAddrP`	Address to send to (`NetSocketAddrType*`), or 0
-> `toLen`	Size of `addrP` buffer.
-> `timeout`	Maximum timeout in system ticks, -1 means wait forever.
<- `errP`	Address of variable used to return error code.

Result Codes

0	Socket has been shut down by remote host.
>0	Number of bytes successfully sent.
-1	Error occurred, error code in `*errP`.

Errors

0	No error.
`netErrTimeout`	Call timed out.
`netErrNotOpen`	
`netErrParamErr`	
`netErrSocketNotOpen`	
`netErrMessageTooBig`	
`netErrSocketNotConnected`	
`netErrClosedByRemote`	
`netErrIPCantFragment`	
`netErrIPNoRoute`	
`netErrIPNoSrc`	
`netErrIPNoDst`	
`netErrIPPktOverflow`	

Sockets Equivalent

```
int sendto (int socket, const void* bufP,
            int bufLen, int flags,
            const void* toAddrP, int toLen);
```

```
int send   (int socket, const void* bufP,
            int bufLen, int flags);
int write  (int socket, const void* bufP,
            int bufLen,);
```

Comments

This call attempts to write data to the specified socket and returns the number of bytes actually sent, which may be less than or equal to the requested number of bytes. The data is passed in a single buffer that `bufP` points to.

If the socket is a datagram socket and the data is too large to fit in a single UDP packet, no data is sent and -1 is returned.

For stream-based sockets, `toAddrP` is always ignored, since by definition a `NetLibSocketAccept` socket must have a connection established with a remote host before data can be written. For datagram sockets, an error is returned if the socket was previously connected and `toAddrP` is specified.

If there isn't enough buffer space to send any data, this call will block until there is enough buffer space, or until a timeout.

Note: For stream-based sockets, this call may write only a portion of the desired data. It always returns the number of bytes actually written. Consequently, the caller should be prepared to call this routine repeatedly until the desired number of bytes have been written, or until it returns 0 or -1.

See Also

`NetLibSendPB`, `NetLibReceive`, `NetLibReceivePB`, `NetLibDmReceive`

NetLibSendPB

Purpose Send data to a socket from a scatter-write array.

Prototype

```
SWord NetLibSendPB(Word libRefnum,
                   NetSocketRef socket,
                   NetIOParamType* pbP,
                   Word flags,
                   Long timeout,
                   Err* errP)
```

Parameters

-> `libRefNum`	Reference number of the net library.
-> `socket`	`SocketRef` of the open socket.
-> `pbP`	Pointer to parameter block containing buffer info.
-> `flags`	One or more `netMsgFlagXXX` flag.
-> `timeout`	Maximum timeout in system ticks; -1 means wait forever.
<- `errP`	Address of variable used to return error code.

Result Codes

0	Socket has been shut down by remote host.
>0	Number of bytes successfully sent
-1	Error occurred, error code in `*errP`.

Errors

0	No error.
`netErrTimeout`	Call timed out.
`netErrNotOpen`	
`netErrParamErr`	
`netErrSocketNotOpen`	
`netErrMessageTooBig`	
`netErrSocketNotConnected`	
`netErrClosedByRemote`	

`netErrIPCantFragment`

`netErrIPNoRoute`

`netErrIPNoSrc`

`netErrIPNoDst`

`netErrIPPktOverflow`

Sockets Equivalent

```
int sendmsg ( int socket,
              const struct msghdr* pbP,
              int flags);
```

Comments

This call attempts to write data to the given socket and returns the number of bytes actually sent, which may be less than or equal to the requested number of bytes. The data is passed in the scatter-write array specified in the `pbP` parameter block.

If the socket is a datagram socket and the data is too large to fit in a single UDP packet, no data will be sent and -1 will be returned.

For stream-based sockets, `pbP->addrP` is always ignored since by definition a `NetLibSocketAccept` socket must have a connection established with a remote host before data can be written. For datagram sockets, an error will be returned if the socket was previously connected and `pbP->addrP` is specified.

If there isn't enough buffer space to send any data, this call will block until there is space, or until a timeout.

Note: For stream-based sockets, this call may write only a portion of the desired data. It always returns the number of bytes actually written. Consequently, the caller should be prepared to call this routine repeatedly until the desired number of bytes have been written, or until it returns 0 or -1.

See Also

`NetLibSend`, `NetLibReceive`, `NetLibReceivePB`, `NetLibDmReceive`

Utilities

NetHToNL

Purpose Converts a 32-bit value from host to network byte order.

Prototype `DWord NetHToNL (DWord x)`

Parameters -> `x` 32-bit value to convert.

Result Returns `x` in network byte order.

Errors None

Sockets Equivalent `htonl()`

See Also `NetNToHS`, `NetNToHL`, `NetHToNS`

NetHToNS

Purpose Converts a 16-bit value from host to network byte order.

Prototype `Word NetHToNS (Word x)`

Parameters -> `x` 16-bit value to convert.

Result Returns `x` in network byte order.

Errors None

Sockets Equivalent `htons()`

See Also NetNToHS, NetNToHL, NetHToNL

NetLibAddrAToIN

Purpose Converts an ASCII string representing a dotted decimal IP address into a 32 IP address in network byte order.

Prototype

```
NetIPAddr NetLibAddrAToIN ( Word libRefnum,
                            CharPtr nameP)
```

Parameters

-> `libRefNum`	Reference number of the net library.
-> `nameP`	Pointer to ASCII dotted decimal string.

Result

-1	Invalid `nameP`, `nameP` doesn't represent a dotted decimal IP address
!= -1	32-bit network byte order IP address

Sockets Equivalent `unsigned long inet_addr(char* cp)`

See Also NetLibAddrINToA

NetLibAddrINToA

Purpose Converts an IP address from 32-bit network byte order into a dotted decimal ASCII string.

Prototype

```
CharPtr NetLibAddrINToA (Word libRefnum,
                         NetIPAddr inet,
                         CharPtr spaceP)
```

Parameters

-> `libRefNum`	Reference number of the net library.
-> `inet`	32-bit IP address in network byte order.
-> `spaceP`	Buffer used for holding return name.

Result

`spaceP`	Dotted decimal ASCII string representation of IP address.

Sockets Equivalent `char* inet_ntoa(struct in_addr in)`

See Also `NetLibAddrAToIN`

NetLibGetHostByAddr

Purpose Looks up a host name given its IP address.

Prototype

```
NetHostInfoPtr NetLibGetHostByAddr (
                        Word libRefnum,
                        BytePtr addrP,
                        Word len,
                        Word type,
                        NetHostInfoBufPtr bufP,
                        Long timeout,
                        Err* errP)
```

Parameters

->	`libRefNum`	Reference number of the net library.
->	`addrP`	IP address of host to lookup.
->	`len`	Length, in bytes, of `*addrP`.
->	`type`	Type of `addrP`. `netSocketAddrINET` is currently the only supported type.
->	`bufP`	Pointer to buffer to hold results of lookup.
->	`timeout`	Maximum timeout in system ticks, -1 means wait forever.
<-	`errP`	Address of variable used to return error code.

Result

0	Name not found, `*errP` contains error code.
!=0	Pointer to `NetHostInfoType` portion of `bufP` that contains results of the lookup.

Errors

0	No Error
`netErrTimeout`	Call timed out.
`netErrNotOpen`	
`netErrDNSNameTooLong`	
`netErrDNSBadName`	
`netErrDNSLabelTooLong`	
`netErrDNSAllocationFailure`	
`netErrDNSTimeout`	
`netErrDNSUnreachable`	
`netErrDNSFormat`	
`netErrDNSServerFailure`	
`netErrDNSNonexistantName`	
`netErrDNSNIY`	
`netErrDNSRefused`	
`netErrDNSImpossible`	
`netErrDNSNoRRS`	

`netErrDNSAborted`

`netErrDNSBadProtocol`

`netErrDNSTruncated`

`netErrDNSNoRecursion`

`netErrDNSIrrelevant`

`netErrDNSNotInLocalCache`

`netErrDNSNoPort`

Sockets Equivalent

```
struct hostent* gethostbyaddr (char* addr,
                               int len,
                               int type);
```

Comments

This call queries the domain name server(s) to look up a host name given its IP address.

`BufP` must point to a structure of type `NetHostInfoBufType` that will be used to store the results of the lookup. When this call returns, it returns with a pointer to a structure of type `NetHostInfoType` which is actually part of the `NetHostInfoBufType` that `bufP` points to.

See Also

`NetLibGetHostByName`

NetLibGetHostByName

Purpose

Looks up a host IP address given a host name.

Prototype

```
NetHostInfoPtr NetLibGetHostByName (
                        Word libRefnum,
                        CharPtr nameP,
                        NetHostInfoBufPtr bufP,
                        Long timeout,
                        Err* errP)
```

Parameters	-> `libRefNum`	Reference number of the net library.
	-> `nameP`	Name of host to look up.
	-> `bufP`	Pointer to buffer to hold results of look up.
	-> `timeout`	Maximum timeout in system ticks, -1 means wait forever.
	<- `errP`	Address of variable used to return error code.
Result	0	Name not found, `*errP` contains error code.
	!=0	Pointer to `NetHostInfoType` portion of `bufP` which contains results of the lookup.
Errors	0	No Error
	`netErrTimeout`	Call timed out.
	`netErrNotOpen`	
	`netErrDNSNameTooLong`	
	`netErrDNSBadName`	
	`netErrDNSLabelTooLong`	
	`netErrDNSAllocationFailure`	
	`netErrDNSTimeout`	
	`netErrDNSUnreachable`	
	`netErrDNSFormat`	
	`netErrDNSServerFailure`	
	`netErrDNSNonexistantName`	
	`netErrDNSNIY`	
	`netErrDNSRefused`	
	`netErrDNSImpossible`	
	`netErrDNSNoRRS`	
	`netErrDNSAborted`	
	`netErrDNSBadProtocol`	
	`netErrDNSTruncated`	

`netErrDNSNoRecursion`

`netErrDNSIrrelevant`

`netErrDNSNotInLocalCache`

`netErrDNSNoPort`

Sockets Equivalent

```
struct hostent *gethostbyname(char* name);
```

Comments This call first checks the local `name -> IP` address host table in the net library preferences. If the entry is not found, it then queries the domain name server(s).

`BufP` must point to a structure of type `NetHostInfoBufType`, which is used to store the results of the lookup. When this call returns, it returns with a pointer to a structure of type `NetHostInfoType` which is actually part of the `NetHostInfoBufType` pointed to `bufP`.

See Also `NetLibGetHostByAddr`, `NetLibGetMailExchangeByName`

NetLibGetMailExchangeByName

Purpose Looks up the name of a host to use for a given mail exchange.

Prototype

```
SWord NetLibGetMailExchangeByName (Word libRefNum,
          CharPtr mailNameP,
          Word maxEntries,
          Char hostNames[][netDNSMaxDomainName+1],
          Word priorities[],
          Long timeout,
          Err* errP)
```

Parameters

-> `libRefNum`	Reference number of the net library.
-> `mailNameP`	Name of the mail exchange to look up.
-> `maxEntries`	Maximum number of hostnames to return.

<- `hostNames`	Array of character strings of length `netDNSMaxDomainName+1`. The host name results are stored in this array. This array must be able to hold at least `maxEntries` hostnames.
<- `priorities`	Array of Words. The priorities of each host name found are stored in this array. This array must be at least `maxEntries` in length.
-> `timeout`	Maximum timeout in system ticks; -1 means wait forever.
<- `errP`	Address of variable used to return error code.

Result

>=0	Number of entries successfully found.
<0	Error occurred, error code is in *`errP`.

Errors

0	No Error
`netErrTimeout`	Call timed out.
`netErrNotOpen`	
`netErrDNSNameTooLong`	
`netErrDNSBadName`	
`netErrDNSLabelTooLong`	
`netErrDNSAllocationFailure`	
`netErrDNSTimeout`	
`netErrDNSUnreachable`	
`netErrDNSFormat`	
`netErrDNSServerFailure`	
`netErrDNSNonexistantName`	
`netErrDNSNIY`	
`netErrDNSRefused`	
`netErrDNSImpossible`	
`netErrDNSNoRRS`	
`netErrDNSAborted`	

`netErrDNSBadProtocol`

`netErrDNSTruncated`

`netErrDNSNoRecursion`

`netErrDNSIrrelevant`

`netErrDNSNotInLocalCache`

`netErrDNSNoPort`

Sockets Equivalent None

Comments This call looks up the name(s) of host(s) to use for sending an e-mail. The caller passes the name of the mail exchange in `mailNameP` and gets back a list of host names to which the mail message can be sent.

See Also `NetLibGetHostByAddr`, `NetLibGetHostByName`

NetLibGetServByName

Purpose Looks up the port number for a standard TCP/IP service, given the desired protocol.

Prototype

```
NetServInfoPtr NetLibGetServByName (
                        Word libRefnum,
                        CharPtr servNameP,
                        CharPtr protoNameP,
                        NetServInfoBufPtr bufP,
                        Long timeout,
                        Err* errP)
```

Parameters

->	`libRefNum`	Reference number of the net library.
->	`servNameP`	Name of the service to look up.
->	`protoNameP`	Desired protocol to use.
->	`bufP`	Buffer to store results in.

-> `timeout`	Maximum timeout in system ticks, -1 means wait forever.
<- `errP`	Address of variable used to return error code.

Result

0	Service not found, `*errP` contains error code.
!=0	Pointer to `NetServInfoType` portion of `bufP` that contains results of the lookup.

Errors

0	No Error
`netErrTimeout`	Call timed out.
`netErrNotOpen`	
`netErrUnknownProtocol`	
`netErrUnknownService`	

Sockets Equivalent

```
struct servent* getservbyname(
   char* addr, char* proto);
```

Comments

This call is a convenience call for looking up a standard port number given the name of a service and the protocol to use (either "udp" or "tcp"). It currently supports looking up the port number for the following services: "echo", "discard", "daytime", "qotd", "chargen", "ftp-data", "ftp", "telnet", "smtp", "time", "name", "finger", "pop2", "pop3", "nntp", "imap2".

`BufP` must point to a structure of type `NetServInfoBufPtr` that's used to store the results of the lookup. When this call returns, it returns with a pointer to a structure of type `NetServInfoType` which is actually part of the `NetServInfoBufType` pointed to `bufP`.

See Also

`NetLibGetHostByName`

NetLibMaster

Purpose Retrieves the network statistics, interface statistics, and the contents of the trace buffer.

Prototype

```
Err NetLibMaster (Word libRefnum,
                  Word cmd,
                  NetMasterPBPtr pbP,
                  Long timeout)
```

Parameters

-> `libRefNum`	Reference number of the net library.
-> `cmd`	Function to perform (`NetMasterEnum` type).
-> `pbP`	Command parameter block.
-> `timeout`	Timeout in ticks, -1 means wait forever.

Result

0	No error
`netErrNotOpen`	
`netErrParamErr`	
`netErrUnimplemented`	

Sockets Equivalent None

Comments This call allows applications to can get detailed information about the net library. This information is usually helpful in debugging network configuration problems.

This function takes a command word (`cmd`) and parameter block pointer as arguments and returns its results in the parameter block on exit.

The following commands are supported:

netMasterInterfaceInfo

pbP.interfaceInfo:

index	->	Index of interface to fetch info about.
creator	<-	Creator of interface.
instance	<-	Instance of interface.
netIFP	<-	Private interface info pointer.
drvrName	<-	Driver type that interface uses (“PPP”, “SLIP”, etc.).
hwName	<-	Hardware driver name (“Serial Library”, etc.).
localNetHdrLen	<-	Number of bytes in local net header.
localNetTrailerLen	<-	Number of bytes in local net trailer.
localNetMaxFrame	<-	Local net maximum frame size.
ifName	<-	Interface name with instance number concatenated.
driverUp	<-	True if interface driver is up.
ifUp	<-	True if interface media layer is up.
hwAddrLen	<-	Length of interface’s hardware address.
hwAddr	<-	Interface’s hardware address.
mtu	<-	Maximum transfer unit of interface.
speed	<-	Speed in bits/sec.

lastStateChange	<-	Time in milliseconds of last state change.
ipAddr	<-	IP address of interface.
subnetMask	<-	Subnet mask of local network.
broadcast	<-	Broadcast address of local network.

netMasterInterfaceStats

pbP.interfaceStats:

index	->	Index of interface to fetch info about.
inOctets	<-	Number of octets received.
inUcastPkts	<-	Number of packets received.
inNUcastPkts	<-	Number of broadcast packets received.
inDiscards	<-	Number of incoming packets that were discarded.
inErrors	<-	Number of packet errors encountered.
inUnknownProtos	<-	Number of unknown protocols encountered.
outOctets	<-	Number octets sent.
outUcastPkts	<-	Number of packets sent.
outNUcastPkts	<-	Number of broadcast packets sent.
outDiscards	<-	Number of packets discarded.
outErrors	<-	Number of outbound packet errors.

netMasterIPStats

pbP.ipStats:

ipXXX	<-	see `NetMgr.h` for complete list of stats returned

netMasterICMPStats

pbP.icmpStats:

icmpXXX	<-	see `NetMgr.h` for complete list of stats returned

netMasterUDPStats

pbP.udpStats

updXXX	<-	see `NetMgr.h` for complete list of stats returned

netMasterTCPStats

pbP.tcpStats:

tcpXXX	<-	see `NetMgr.h` for complete list of stats returned

netMasterTraceEventGet

pbP.traceEventGet

index	->	Index of event to fetch.
textP	->	Pointer to text string to return event in. Should be at least 256 bytes long.

See Also `NetLibSettingSet`

NetLibSelect

Purpose Blocks until I/O is ready on one or more descriptors, where a descriptor can represent socket input, socket output, or a user input event like a pen tap or key press.

Prototype

```
SWord NetLibSelect (Word libRefnum,
                   Word width,
                   NetFDSetType* readFDs,
                   NetFDSetType* writeFDs,
                   NetFDSetType* exceptFDs,
                   Long timeout,
                   Err* errP)
```

Parameters

-> `libRefNum`	Reference number of the net library.
-> `width`	Number of descriptor bits to check in the `readFDs`, `writeFDs`, and `exceptFDs` descriptor sets.
<-> `readFDs`	Pointer to `NetFDSetType` containing set of bits representing descriptors to check for input.
<-> `writeFDs`	Pointer to `NetFDSetType` containing set of bits representing descriptors to check for output.
<-> `exceptFDs`	Pointer to `NetFDSetType` containing set of bits representing descriptors to check for exception conditions.
-> `timeout`	Maximum timeout in system ticks; -1 means wait forever.
<- `errP`	Address of variable used to return error code.

Result Codes

>0	Sum total number of ready file descriptors in *`readFDs`, *`writeFDs`, and *`exceptFDs`.
0	Timeout.
-1	Error occurred, error code in *`errP`.

Errors

0	No Error
`netErrTimeout`	Call timed out.
`netErrNotOpen`	

Sockets Equivalent

```
int select (int width, fd_set* readfds,
            fd_set* writefds, fd_set* exceptfds,
            struct timeval* timeout);
```

Comments

This call blocks until one or more descriptors are ready for I/O. In the Palm OS environment, a descriptor is either a `NetSocketRef` or the “stdin” descriptor, `sysFileDescStdIn`. The `sysFileDescStdIn` descriptor will be ready for input whenever a user event is available like a pen tap or key press.

The caller should set which bits in each descriptor set need to be checked by using the `netFDZero` and `netFDSet` macros. After this call returns, the macro `netFDIsSet` can be used to determine which descriptors in each set are actually ready.

On exit, the total number of ready descriptors is returned and each descriptor set is updated with the appropriate bits set for each ready descriptor in that set.

The following example illustrates how to use this call to check for input on a socket or a user event:

```
Err     err;
NetSocketRef   socketRef;
NetFDSetType   readFDs,writeFDs,exceptFDs;
SWord       numFDs;
Word       width;

// Create the descriptor sets
netFDZero(&readFDs);
netFDZero(&writeFDs);
netFDZero(&exceptFDs);
netFDSet(sysFileDescStdIn, &readFDs);
netFDSet(socketRef, &readFDs);

// Calculate the max descriptor number and use
// that +1 as the max width.
// Alternatively, we could simply use the
// constant netFDSetSize as the width which is
// simpler but makes the NetLibSelect call
// slightly slower.
width = sysFileDescStdIn;
if (socketRef > width) width = socketRef;

// Wait for any one of the descriptors to be
// ready.
numFDs = NetLibSelect(AppNetRefnum, width+1,
&readFDs, &writeFDs, &exceptFDs,
AppNetTimeout, &err);
```

See Also NetLibSocketOptionSet

NetLibTracePrintF

Purpose Can be used by applications to store debugging information in the net library's trace buffer.

Prototype

```
Err NetLibTracePrintF (Word libRefnum,
                       CharPtr formatStr, ...)
```

Parameters

-> `libRefNum`	Reference number of the net library.
-> `formatStr`	A `printf` style format string.
-> ...	Arguments to the format string.

Result

0	No error.
`netErrNotOpen`	

Sockets Equivalent None

Comments This call is a convenient debugging tool for developing Internet applications. It will store a message into the net library's trace buffer, which can later be dumped using the `NetLibMaster` call. The net library's trace buffer is used to store run-time errors that the net library encounters as well as errors and messages from network interfaces and from applications that use this call.

The `formatStr` parameter is a `printf` style format string which supports the following format specifiers:

%d, %i, %u, %x, %s, %c but it does NOT support field widths, leading 0's etc.

Note that the `netTracingAppMsgs` bit of the `netSettingTraceBits` setting must be set using the call `NetLibSettingSet(...netSettingTraceBits...)`. Otherwise, this routine will do nothing.

See Also `NetLibTracePutS`, `NetLibMaster`, `NetLibSettingSet`

NetLibTracePutS

Purpose Can be used by applications to store debugging information in the net library's trace buffer.

Prototype

```
Err NetLibTracePutS(Word libRefnum, CharPtr strP)
```

Parameters

-> `libRefNum`	Reference number of the net library.
-> `strP`	String to store in the trace buffer.

Result

0	No error
`netErrNotOpen`	

Sockets Equivalent None

Comments This call is a convenient debugging tool for developing internet applications. It will store a message into the net library's trace buffer which can later be dumped using the `NetLibMaster` call. The net library's trace buffer is used to store run-time errors that the net library encounters as well as errors and messages from network interfaces and from applications that use this call.

Note the `netTracingAppMsgs` bit of the `netSettingTraceBits` setting must be set using the `NetLibSettingSet(...netSettingTraceBits...)` call or this routine will do nothing.

See Also `NetLibTracePrintF`, `NetLibMaster`, `NetLibSettingSet`.

NetNToHL

Purpose Converts a 32-bit value from network to host byte order.

Prototype `DWord NetNToHL (DWord x)`

Parameters -> `x` 32-bit value to convert.

Result Returns `x` in host byte order.

Errors none

Sockets Equivalent `ntohl()`

See Also `NetNToHS`, `NetHToNL`, `NetHToNS`

NetNToHS

Purpose Converts a 16-bit value from network to host byte order.

Prototype `Word NetNToHS (Word x)`

Parameters -> `x` 16-bit value to convert.

Result Returns `x` in host byte order.

Errors None

Sockets Equivalent `ntohs()`

See Also `NetHToNL`, `NetNToHL`, `NetHToNS`

Configuration

NetLibIFAttach

Purpose

Attach a new network interface.

Prototype

```
Err NetLibIFAttach (Word libRefnum,
                    DWord ifCreator,
                    Word ifInstance,
                    SDWord timeout)
```

Parameters

-> `libRefNum`	Reference number of the net library.
-> `ifCreator`	Creator of interface to attach.
-> `ifInstance`	Instance number of interface to attach.
-> `timeout`	Timeout in ticks; -1 means infinite timeout.

Result

0	Success

`netErrInterfaceNotFound`
`netErrTooManyInterfaces`

Sockets Equivalent

None

Comments

This call can be used to attach a new network interface to the net library. Network interfaces are self-contained databases of type 'neti'. The `ifCreator` parameter to this function is used to locate the network interface database of the given creator.

If the net library is already open when this call is made, the network interface's database will be located and then called to initialize itself and attach itself to the protocol stack in real-time. If the net library is not open when this call is made, the creator and instance number of the interface are stored in the Net Prefs database and the interface is initialized and attached to the stack the next time the net library is opened.

See Also

`NetLibIFGet`, `NetLibIFDetach`

NetLibIFDetach

Purpose Detach a network interface from the protocol stack.

Prototype

```
Err NetLibIFDetach (Word libRefnum,
                    DWord ifCreator,
                    Word ifInstance,
                    SDWord timeout)
```

Parameters

-> `libRefNum`	Reference number of the net library.
-> `ifCreator`	Creator of interface to detach.
-> `ifInstance`	Instance number of interface to detach.
-> `timeout`	Timeout in ticks; -1 means infinite timeout.

Result

0	Success
`netErrInterfaceNotFound`	

Sockets Equivalent None

Comments This call can be used to detach a network interface from the net library. If the net library is already open when this call is made, the interface is brought down and detached from the protocol stack in real-time. If the net library is not open when this call is made, the creator and instance number of the interface are removed in the Net Prefs database and the interface is not attached the next time the library is opened.

See Also `NetLibIFGet`, `NetLibIFAttach`

NetLibIFDown

Purpose Bring an interface down and hang up a connection.

Prototype

```
Err NetLibIFDown (Word libRefnum,
                  DWord ifCreator,
                  Word ifInstance,
                  SDWord timeout)
```

Parameters

-> `libRefNum`	Reference number of the net library.
-> `ifCreator`	Creator of interface to attach.
-> `ifInstance`	Instance number of interface to attach.
-> `timeout`	Timeout in ticks. -1means wait forever.

Result

0	Success
`netErrNotOpen`	
`netErrInterfaceNotFound`	

Sockets Equivalent None

Comments The net library must be open before this call can be made. For dial-up interfaces, this call terminates a connection and hangs up the modem if necessary.

`NetLibClose` automatically brings down any attached interfaces, so this routine doesn't normally have to be called.

If the interface is already down, this routine returns immediately with no error.

See Also `NetLibIFGet`, `NetLibIFAttach`, `NetLibIFDetach`, `NetLibIFUp`

NetLibIFGet

Purpose Get the creator and instance number of an installed interface by index.

Prototype

```
Err NetLibIFGet (Word libRefnum,
                 Word index,
                 DWordPtr ifCreatorP,
                 WordPtr ifInstanceP)
```

Parameters

-> `libRefNum`	Reference number of the net library.
-> `index`	Index of the interface to get. Indices start at 0.
<- `ifCreatorP`	Creator of interface is returned here.
<- `ifInstanceP`	Instance number of interface is returned here.

Result

0	Success
`netErrInvalidInterface`	Index too high
`netErrPrefNotFound`	

Sockets Equivalent None

Comments To get a list of all installed interfaces, call this function with successively increasing indices starting from 0 until the error `netErrInvalidInterface` is returned.

The `ifCreator` and `ifInstance` values returned from this call can then be used with the `NetLibSettingGet` call to get more information about that particular interface.

See Also `NetLibIFAttach`, `NetLibIFDetach`

NetLibIFSettingGet

Purpose Retrieves a network interface specific setting.

Prototype

```
Err NetLibIFSettingGet (Word libRefnum,
                        DWord ifCreator,
                        Word ifInstance,
                        Word setting,
                        VoidPtr bufP,
                        WordPtr bufLenP)
```

Parameters

-> `libRefNum`	Reference number of the net library.
-> `ifCreator`	Creator of the network interface.
-> `ifInstance`	Instance number of the network interface.
-> `setting`	Setting to retrieve; one of the `netIFSettingXXX` enum constants.
-> `bufP`	Space for return value of setting.
<-> `bufLenP`	On entry, size of `bufP`. On exit, actual size of setting.

Result

0	Success
`netErrUnknownSetting`	Invalid setting constant.
`netErrPrefNotFound`	No current value for setting.
`netErrBufTooSmall`	`bufP` was too small to hold entire setting. Setting value was truncated to fit in `bufP`.
`netErrUnimplemented`	
`netErrInterfaceNotFound`	
`netErrBufWrongSize`	

Sockets Equivalent None

Comments

This call can be used to retrieve the current value of any network interface setting. The caller must pass a pointer to a buffer to hold the return value (`bufP`), the size of the buffer (`*bufLenP`), and the setting ID (`setting`). The setting ID is one of the `netIFSettingXXX` constants in the `netSettingEnum` type.

Some settings, such as the login script, are variable size. For these types of settings, the caller can pass 0 for `*bufLenP`, ignore the return error code of `netErrBufTooSmall`, and get the actual size from the `*bufLenP` variable after the call returns. The buffer can then be allocated and the setting retrieved by passing the actual buffer size in `*bufLenP` and calling `NetLibSettingGet` again.

The following table lists the network interface settings and the size of each setting. Some are only applicable to certain types of interfaces. Settings not applicable to a specific interface can be safely ignored and not set to any particular value.

Setting	Type	Description
ResetAll	void	Used for `NetLibIFSettingSet` only. This clears all other settings for the interface to their default values.
Up	Byte	True if interface is currently up - Read-only
Name	Char[32]	Name of this interface - Read-only.
IPAddr	DWord	IP address of interface.
SubnetMask	DWord	Subnet mask for interface. Doesn't need to be specified for PPP or SLIP type connections.
Broadcast	DWord	Broadcast address for interface. Doesn't need to be specified for PPP or SLIP type connections.
Username	Char[32]	Username. Only required if the login script uses the username substitution escape sequence in it. Call `NetLibIFSettingSet` with a bufLen of 0 to remove this setting.

Setting	Type	Description
Password	Char[32]	Password. Optionally required if the login script uses the password substitution escape sequence in it. Call `NetLibIFSettingSet` with a `bufLen` of 0 to remove this setting. If the login script uses password substitution and no password setting is set, the user will be prompted for a password at connect time.
Dialback Username	Char[32]	Dialback Username. Only required if the login script uses the dialback username substitution escape sequence in it. Call `NetLibIFSettingSet` with a `bufLen` of 0 to remove this setting.
Dialback Password	Char[32]	Dialback Password. Optionally required if the login script uses the dialback password substitution escape sequence in it. Call `NetLibIFSettingSet` with a bufLen of 0 to remove this setting. If the login script uses password substitution and no password setting is set, the user will be prompted for a password at connect time.
AuthUsername	Char[32]	Authentication Username. Only required if the authentication protocol uses a different username than the what's in the Username setting. If this setting is empty (bufLen of 0), the Username setting will be used instead. Call `NetLibIFSettingSet` with a `bufLen` of 0 to remove this setting.
AuthPassword	Char[32]	Authentication Password. If "$" then the user will be prompted for the authentication password at connect time. Else, if 0 length, then the Password setting or the result of its prompt will be used instead. Call `NetLibIFSettingSet` with a `bufLen` of 0 to remove this setting.
ServiceName	Char[]	Service Name. Used for display purposes while showing the connection progress dialog box. Call `NetLibIFSettingSet` with a `bufLen` of 0 to remove this setting.

Setting	Type	Description
LoginScript	Char[]	Login script. Only required if the particular service requires a login sequence. Call `NetLibIFSettingSet` with a bufLen of 0 to remove this setting. See below for a description of the login script format.
ConnectLog	Char[]	Connect log. Generally, this setting is just retrieved, not set. It contains a log of events from the most recent login. To clear this setting, call `NetLibIFSettingSet` with a `bufLen` of 0.
InactivityTimer	Word	Maximum number of seconds of inactivity allowed. Set to 0 to ignore.
Establishment-Timeout	Word	Maximum delay, in seconds, allowed between each stage of connection establishment or login script line. Must be non-zero.
DynamicIP	Byte	If non-zero, negotiate for an IP address. If false, the IP address specified in the `IPAddr` setting will be used. Default is 0.
VJCompEnable	Byte	If non-zero, enable JV header compression. Default is true for PPP and false for SLIP.
VJCompSlots	Byte	Number of slots to use for VJ compression. Default is 4 for PPP and 16 for SLIP. More slots require more memory so it is best to keep this number to a minimum.
MTU	Word	Maximum transmission unit in octets. Currently not implemented in SLIP or PPP.
AsyncCtlMap	DWord	Bitmask of characters to escape for PPP. Default is 0.
PortNum	Word	Which serial communication port to use. Port 0 is the only port available on the device. Ports 0 (modem) and 1 (printer) are available on the Macintosh. Default is port 0.

Setting	Type	Description
BaudRate	DWord	Serial port baud rate to use in bits/sec. MUST be specified.
FlowControl	Byte	If bit 0 is 1, use hardware handshaking on the serial port. Default is no hardware handshaking.
StopBits	Byte	Number of stop bits. Default is 1.
ParityOn	Byte	True if parity detection enabled. Default is false.
ParityEven	Byte	True for even parity detection. Default is true.
UseModem	Byte	If true, dial-up through modem. If false, go direct over serial port
PulseDial	Byte	If true, pulse dial modem. Else, tone dial. Default is tone dial.
ModemInit	Char[]	Zero-terminated modem initialization string, not including the "AT". If not specified (bufLen of 0), the modem init string from system preferences are used.
ModemPhone	Char[]	Zero-terminated modem phone number string. Only required if `UseModem` is true.
RedialCount	Word	Number of times to redial modem when trying to establish a connection. Only required if `UseModem` is true.
TraceBits	DWord	A bitfield of various trace bits (`netTracingXXX`). Default value is `netTracingErrors` which tells the interface to record only run-time errors in the trace buffer. An application can get a list of events in the trace buffer using the `NetLibMaster` call. Each interface has its own trace bits setting so that trace event recording in each interface can be selectively enabled or disabled.

Setting	Type	Description
GlobalsPtr	DWord	Read-only. Interfaces pointer to its global variables.
ActualIPAddr	DWord	Read-only. The actual IP address that the interface ends up using. The login script execution engine stores the result of the “g” (get IP address) command here as does the PPP negotiation logic.

As noted above, the `netIFSettingLoginScript` setting is used to store the login script for an interface. The login script format is a rigidly formatted text string designed to be generated programmatically from user input. If a syntactically incorrect login script is presented to the net library, the results will be unpredictable. The basic format is a series of null terminated command lines followed by a null byte at the end of the script. Each command line has the format:

```
<command-byte> [<parameter>]
```

where the command byte is the first character in the line and there is 1 and only 1 space between the command byte and the parameter string. Following is a list of possible commands:

Function	Command	Parameter	Example
send	s	<string>	’s go PPP’
wait	w	<string>	’w password:’
delay	d	<seconds>	’d 1’
parity	p	e | o | n	’p n’
data bits	b	7 | 8	’b 8’
getIPAddr	g		’g’

Function	Command	Parameter	Example
ask	a	<string>	'a Enter Name:'
callback	c	<seconds>	'c 30' // hang up and wait 30 sec.s for callback

The parameter string to the send ('s') command can contain the following escape sequences:

$USERID	substitutes user name
$PASSWORD	substitutes password
$DBUSERID	substitutes dialback user name
$DBPASSWORD	substitutes dialback password
^c	if c is '@' -> '_', then byte value 0 -> 31 else if c is 'a' -> 'z', then byte value 1 -> 26 else c
<cr>	carriage return (0x0D)
<lf>	line feed (0x0A)
\"	"
\^	^
\<	<
\\	\

See Also `NetLibIFSettingSet`, `NetLibSettingGet`, `NetLibSettingSet`

NetLibIFSettingSet

Purpose Sets a network interface specific setting.

Prototype

```
Err NetLibIFSettingSet (Word libRefnum,
                        DWord ifCreator,
                        Word ifInstance,
                        Word setting,
                        VoidPtr bufP,
                        Word bufLen)
```

Parameters

-> `libRefNum`	Reference number of the net library.
-> `ifCreator`	Creator of the network interface.
-> `ifInstance`	Instance number of the network interface.
-> `setting`	The setting to retrieve, one of the `netSettingXXX` enum constants.
-> `bufP`	Space for return value of setting.
-> `bufLen`	Size of new setting.

Result

0	Success.
`netErrUnknownSetting`	Invalid setting constant.
`netErrPrefNotFound`	No current value for setting.
`netErrBufTooSmall`	`bufP` was too small to hold entire setting. Setting value was truncated to fit in `bufP`.
`netErrUnimplemented`	
`netErrInterfaceNotFound`	
`netErrBufWrongSize`	
`netErrReadOnlySetting`	

Sockets Equivalent None

Comments This call can be used to set the current value of any network interface setting. The caller must pass a pointer to a buffer which holds the new value (`bufP`), the size of the buffer (`bufLen`), and the setting ID (`setting`). The setting ID is one of the `netIFSettingXXX` constants in the `netSettingEnum` type.

See `NetLibIFSettingGet` for an explanation of each of the settings.

Of particular interest is the `netIFSettingResetAll` setting, which, if used, resets all settings for the interface to their default values. When using this setting, `bufP` and `bufLen` are ignored.

See Also `NetLibIFSettingGet`, `NetLibSettingGet`, `NetLibSettingSet`

NetLibIFUp

Purpose Bring an interface up and establish a connection.

Prototype

```
Err NetLibIFUp (Word libRefnum,
                DWord ifCreator,
                Word ifInstance)
```

Parameters

-> `libRefNum`	Reference number of the net library.
-> `ifCreator`	Creator of interface to attach.
-> `ifInstance`	Instance number of interface to attach.

Result

0	Success
`netErrNotOpen`	
`netErrInterfaceNotFound`	
`netErrUserCancel`	
`netErrBadScript`	
`netErrPPPTimeout`	
`netErrAuthFailure`	
`netErrPPPAddressRefused`	

Sockets Equivalent

None

Comments

The net library must be open before this call can be made. For dial-up interfaces, this call will dial up the modem if necessary and run through the connect script to establish the connection.

Important: Some interfaces need or want to display UI to show progress information as the connection is established so. THIS ROUTINE MUST BE CALLED FROM THE UI TASK!

`NetLibOpen` calls this routine for every interface that was specified as attached in its preferences. `NetLibOpen` must therefore be called from the UI task as well.

If the interface is already up, this routine returns immediately with no error. This call doesn't take a timeout parameter because it relies on each interface to have its own established timeout setting.

See Also

`NetLibIFGet`, `NetLibIFAttach`, `NetLibIFDetach`, `NetLibIFDown`

NetLibSettingGet

Purpose Retrieves a general setting.

Prototype

```
Err NetLibSettingGet ( Word libRefnum,
                       Word setting,
                       VoidPtr bufP,
                       WordPtr bufLenP)
```

Parameters

-> `libRefNum`	Reference number of the net library.
-> `setting`	Setting to retrieve, one of the `netSettingXXX` enum constants.
-> `bufP`	Space for return value of setting.
<-> `bufLenP`	On entry, size of `bufP`. On exit, actual size of setting.

Result

0	Success
`netErrUnknownSetting`	Invalid setting constant
`netErrPrefNotFound`	No current value for setting
`netErrBufTooSmall`	`bufP` was too small to hold entire setting. Setting value was truncated to fit in `bufP`.
`netErrBufWrongSize`	

Sockets Equivalent None

Comments This call retrieves the current value of any general setting. The caller must pass a pointer to a buffer to hold the return value (`bufP`), the size of the buffer (*`bufLenP`), and the setting ID (`setting`). The setting ID is one of the `netSettingXXX` constants in the `netSettingEnum` type.

Some settings are variable size, like the host table for example. For these types of settings, the caller can pass 0 for *`bufLenP`, ignore the return error code of `netErrBufTooSmall`, and get the actual size from the *`bufLenP` variable after the call returns. The buffer can then be allocated and the setting retrieved by passing the actual buffer size in *`bufLenP` and calling `NetLibSettingGet` again.

The following table lists the general settings and the type of each setting.

Setting	Type	Description
ResetAll	void	Used for `NetLibSettingSet` only. This will clear all other settings to their default values.
PrimaryDNS	DWord	IP address of primary DNS server. This setting MUST be set to a non-zero IP address in order to support any of the name lookup calls.
SecondayDNS	DWord	IP address of primary DNS server. Set to 0 to have stack ignore this setting.
DefaultRouter	DWord	IP address of default router. Default value is 0 which is appropriate for most implementations with only 1 attached interface (besides loopback). Packets with destination IP addresses that don't lie in the subnet of an attached interface will be sent to this router through the default interface specified by the `DefaultIFCreator` / `DefaultIFInstance` pair.
DefaultIFCreator	DWord	Creator of the default network interface. Default value is 0, which is appropriate for most implementations. Packets with destination IP addresses that don't lie in the subnet of a directly attached interface are sent through this interface. If this setting is 0, the stack automatically makes the first non-loopback interface the default interface.
DefaultIFInstance	Word	Instance number of the default network interface. Packets with destination IP addresses that don't lie in the subnet of an attached interface are sent through the default interface. Default value is 0.

Setting	Type	Description
HostName	Char[]	A zero-terminated character string of 64 bytes or less containing the host name of this machine. This setting is not actually used by the stack. It's present mainly for informative purposes and to support the `gethostname`/`sethostname` sockets API calls. To clear the host name, call `NetLibIFSettingSet` with a `bufLen` of 0.
DomainName	Char[]	A zero-terminated character string of 256 bytes or less containing the default domain. This default domain name is appended to all host names before name lookups are performed. If the name is not found, the host name is looked up again without appending the domain name to it. To have the stack not use the domain name, call `NetLibIFSettingSet` with a `bufLen` of 0.
HostTbl	Char[]	A zero-terminated character string containing the host table. This table is consulted first before sending a DNS query to the DNS server(s). To have the stack not use a host table, call `NetLibIFSettingSet` with a `bufLen` of 0. The format of a host table is a series of lines separated by '\n' in the following format:host.company.com A 111.222.333.444
CloseWaitTime	DWord	The close-wait time in milliseconds. This setting MUST be specified. See the discussion of the `NetLibOpen` and `NetLibClose` calls for an explanation of the close-wait time.
TraceBits	DWord	A bitfield of various trace bits (netTracingXXX). Default value is (`netTracingErrors | netTracingAppMsgs`) which tells the net library to record only run-time errors and application trace messages in its trace buffer. An application can get a list of events in the trace buffer using the `NetLibMaster` call.

Setting	Type	Description
TraceSize	DWord	Maximum trace buffer size in bytes. Setting this setting always clears the existing trace buffer. Default is 2 KB.
TraceRoll	Byte	Boolean value, default is true (non-zero). If true, trace buffer will roll over when it fills. If false, tracing will stop as soon as trace buffer fills.

See Also `NetLibSettingSet`, `NetLibIFSettingSet`, `NetLibIFSettingGet`, `NetLibMaster`

NetLibSettingSet

Purpose Sets a general setting.

Prototype

```
Err NetLibSettingSet (Word libRefnum,
                      Word setting,
                      VoidPtr bufP,
                      Word bufLen)
```

Parameters

->	`libRefNum`	Reference number of the net library.
->	`setting`	Setting to retrieve; one of the `netSettingXXX` enum constants.
->	`bufP`	Space for return value of setting.
->	`bufLen`	Size of new setting.

Result

0	Success
`netErrUnknownSetting`	Invalid setting constant.
`netErrInvalidSettingSize`	`bufLen` was invalid for the given setting.
`netErrBufTooSmall`	`bufP` was too small to hold entire setting. Setting value was truncated to fit in `bufP`.
`netErrBufWrongSize`	
`netErrReadOnlySetting`	

Sockets Equivalent None

Comments

This call can be used to set the current value of any general setting. The caller must pass a pointer to a buffer which holds the new value (`bufP`), the size of the buffer (`bufLen`), and the setting ID (`setting`). The setting ID is one of the `netSettingXXX` constants in the `netSettingEnum` type.

See `NetLibSettingGet` for an explanation of each of the settings.

Of particular interest is the `netSettingResetAll` setting, which, if used, will reset all general settings to their default values. When using this setting, `bufP` and `bufLen` are ignored.

See Also

`NetLibSettingGet`, `NetLibSettingSet`, `NetLibIFSettingSet`, `NetLibMaster`

Berkeley Sockets API Calls

When the `<sys/socket.h>` header file is included, code written to the Berkeley sockets API can be compiled for the Palm OS environment with little or no source code modifications. The `<sys/socket.h>` header file contains a set of macros which map Berkeley sockets API calls into net library and Palm OS calls. In addition, a Palm OS application using the sockets API must link with the module `NetSocket.c` which contains glue code and global variables used by the sockets API.

Before an application can use any sockets API calls, it must open the net library as described in Initialization and Shutdown. The code fragment in that section correctly sets up the application global variable `AppNetRefnum` with the `refnum` of the net library which is used by the sockets API macros.

Another important global declared in "`NetSocket.c`" is `AppNetTimeout`. This global gets passed as the `timeout` parameter to the native net library call by sockets API macros. This timeout variable is a 32-bit value representing the maximum number system ticks to wait. Most applications will probably want to adjust this timeout value and possibly adjust it for different sections of code.

Finally, the global `errno` must be declared in the application's own source code UNLESS the application is linked with the standard C library which also declares it.

The following code fragment illustrates the above steps:

```
#include <sys/socket.h>
....
// Declare errno global; we don't link with stdlib
Err  errno;
...
// Open up the net library
err = SysLibFind("Net.lib", &AppNetRefnum);
if (err) {/* error handling here */}
err = NetLibOpen(AppNetRefnum, &ifErrs);
if (err || ifErrs) {/* error handling here */}
```

```
// Change the default timeout
AppNetTimeout = SysTicksPerSecond() * 10;
                                    // 10 seconds.
```

The following section list the calls in the Berkeley sockets API which are supported by the net library. In some cases, the calls have limited functionality from what's found in a full implementation of the sockets API and these limitations are described here.

Supported Socket Functions

Function	Description
bind()	This function binds a socket to a local address
close()	This function closes a socket
connect()	This function connects a socket to a remote endpoint to establish a connection.
fcntl()	This function is supported only for socket `refnum`s and the only commands it supports are F_SETFL and F_GETFL. The commands can be used to put a socket into non-blocking mode by setting the FNDELAY flag in the argument parameter appropriately — all other flags are ignored. The F_SETFL, F_GETFL, and FNDELAY constants are defined in `<unix/fcntl.h>`.
getpeername()	This function gets the remote socket address for a connection.
getsockname()	This function gets the local socket address of a connection.
getsockopt()	This function gets control options of a socket. Only the following options are implemented:
TCP_NODELAY	This option returns the current state of the TCP_NODELAY option. This option allows the application to disable the TCP output buffering algorithm so that TCP sends small packets as soon as possible. This constant is defined in `<netinet/tcp.h>`.

Function	Description
TCP_MAXSEG	This option allows the application to get the TCP maximum segment size. This constant is defined in `<netinet/tcp.h>`.
SO_KEEPALIVE	This option returns the keep-alive state. Keep-alive enables periodic transmission of probe segments when there is no data exchanged on a connection. If the remote endpoint doesn't respond, the connection is considered broken, and `so_error` is set to ETIMEOUT.
SO_LINGER	This option specifies what to do with the unsent data when a socket is closed. It uses the `linger` structure defined in `sys/socket.h`.
SO_ERROR	This option returns the current value of the variable `so_error`, defined in `sys/socketvar.h`.
SO_TYPE	This option returns the socket type to the caller.
listen()	Sets up the socket to listen for incoming connection requests. The queue size is quietly limited to 1.
read(), recv(), recvmsg(), recvfrom()	These functions read data from a socket. The recv, recvmsg, and recvfrom calls support the MSG_PEEK flag but NOT the MSG_OOB or MSG_DONTROUTE flags.

Function	Description
`select()`	This function allows the application to block on multiple I/O events. The system will wake up the application process when any of the multiple I/O events occurs. This function uses the timeval structure defined in `<sys/time.h>` and the fd_set structure defined in `sys/types.h`. Also associated with this function are the following four macros defined in `sys/types.h` FD_ZERO() FD_SET() FD_CLR() FD_ISSET() Besides socket descriptors, this function also works with the "stdin" descriptor, `sysFileDescStdIn`. This descriptor is marked as ready for input whenever a user or system event is available in the event queue. This includes any event that would be returned by `EvtGetEvent`. No other descriptors besides `sysFileDescStdIn` and socket `refnum`s are allowed.
send(), sendmsg(), sendto()	These functions write data to a socket. These calls, unlike the recv calls, do support the MSG_OOB flag. The MSG_PEEK flag is not applicable and the MSG_DONTROUTE flag is not supported.
setsockopt()	This function sets control options of a socket. Only the following options are allowed:
TCP_NODELAY	This option allows the application to disable the TCP output buffering algorithm so that TCP sends small packets as soon as possible. This constant is defined in `netinet/tcp.h`.
SO_KEEPALIVE	This option enables periodic transmission of probe segments when there is no data exchanged on a connection. If the remote endpoint doesn't respond, the connection is considered broken, and `so_error` is set to ETIMEOUT.
SO_LINGER	This option specifies what to do with the unsent data when a socket is closed. It uses the linger structure defined in `sys/socket.h`.

Function	Description
shutdown()	This function is similar to `close()`; however, it gives the caller more control over a full-duplex connection.
socket()	This function creates a socket for communication.The only valid address family is AF_INET. The only valid socket types are SOCK_STREAM and SOCK_DGRAM; SOCK_RAW is not supported. The protocol parameter should be set to 0.
write()	This function writes data to a socket.

Supported Network Utility Functions

Function	Description
getdomainname()	This function returns the domain name of the local host
gethostbyaddr()	This function looks up host information given the host's IP address. It returns a hostent structure, is defined in `<netdb.h>`.
gethostbyname()	This function looks up host information given the host's name. It returns a hostent structure which is defined in `<netdb.h>`.
gethostname()	This function returns the name of the local host
getservbyname()	This function returns a servent structure, defined in `<netdb.h>` given a service name.
gettimeofday()	This function returns the current date and time.
setdomainname()	This function sets the domain name of the local host
sethostname()	This function sets the name of the local host
settimeofday()	This function sets the current date and time.

Supported Byte Ordering Functions

The byte ordering functions are defined in `<netinet/in.h>.` They convert and integer between network byte order and the host byte order.

Function	Description
htonl()	Converts a 32-bit integer from host byte order to network byte order.
htons()	Converts a 16-bit integer from host byte order to network byte order.
ntohl()	Converts a 32-bit integer from network byte order to host byte order.
ntohs()	Converts a 16-bit integer from network byte order to host byte order.

Supported Network Address Conversion Functions

The network address conversion functions are declared in the `<arpa/inet.h>` header file. They convert a network address from one format to another, or manipulate parts of a network address.

Function	Description
inet_addr()	Converts an IP address from dotted decimal format to 32-bit binary format.
inet_network()	Converts an IP network number from a dotted decimal format to a 32-bit binary format
inet_makeaddr()	Returns an IP address in an in_addr structure given an IP network number and an IP host number in 32-bit binary format.
inet_lnaof()	Returns the host number part of an IP address.

Function	Description
inet_netof()	Returns the network number part of an IP address.
inet_ntoa()	Converts an IP address from 32-bit format to dotted decimal format.

Supported System Utility Functions

The following byte operation functions are not related to network API per se. However, they are almost always used in BSD network application source.

Function	Description
bcopy()	This function copies a block of data from one memory location to another.
bzero()	This function sets a buffer to all zeros.
bcmp()	This function compares data stored in two buffers.
sleep()	This function causes the current task to sleep for a given period of time.

8

Exchange Manager

The Palm OS exchange manager provides a simple interface for Palm OS applications to send and receive typed data from any number of remote devices and protocols. The device at the remote end of a connection does not need to know it is talking to a Palm OS device. The exchange manager can be used with industry standard protocols and data formats. The burden of understanding the protocols and data formats is on the Palm OS application using the exchange manager.

The exchange manager was developed to provide a facility by which Palm OS applications could communicate directly with external devices and foreign data formats, without having to be tied to the HotSync mechanism and conduits. In the increasingly complex world of the Internet, wireless communications, and infrared communications, it cannot be expected that all these modes of communication must support HotSync and provide the appropriate conduits on the other end. The Palm OS device must be able to deal directly with foreign data formats since there will not be conduits on the remote end to prepare the data. The data may also be sent without regard to the version or even the existence of particular software on the device.

Overview

The exchange manager is designed as a generic communications facility by which typed data objects can be sent and received. It is designed to support a variety of underlying transport mechanisms. Currently, the exchange manager supports only the IR (beaming) capability of the Palm III devices (and upgraded PalmPilot devices).

NOTE: When used for IR communication, the exchange manager uses the OBEX IrDA protocol. The only level of OBEX supported currently is for the Put operation. The Palm III can act as both a client and a server.

The exchange manager API provides a mechanism for exchanging typed data objects between applications. An object is a stream of bytes with some information about its contents attached. The content information includes a creator ID, a MIME data type and an optional filename. An application that wants to send data using the exchange manager must provide at least one of these pieces of information. An application that is able to receive an object registers itself with the exchange manager (`ExgRegisterData`) and specifies what data types and file extensions it can accept.

A key data structure used by the exchange manager is the `ExgSocketType` data type. This exchange socket structure defines information about the connection and the type of data to be exchanged. When you are sending data, you must supply this structure with the appropriate information filled in. When you are receiving, this structure gives you information about the connection and the incoming data. (Note that the use of the term "socket" in the exchange manager API is not related to the term "socket" as used in sockets communication programming.)

Exchange Manager and Launch Codes

When receiving incoming data, the exchange manager communicates with applications via launch codes. The exchange manager sends an application a series of three launch codes when it receives data for it. These are:

- `sysAppLaunchCmdExgAskUser`
- `sysAppLaunchCmdExgReceiveData`
- `sysAppLaunchCmdGoto`

The exchange manager sends the first launch code, `sysAppLaunchCmdExgAskUser`, when it has determined that incoming data is destined for a particular application (based on which application has registered to receive data of that type). This launch code lets the application tell the exchange manager whether or not to display a dialog asking the user if they want to accept the data. If the application chooses not to handle this launch command, the default course of action is that the exchange manager displays a dialog asking the user if they want to accept the incoming data. In most cases, applications won't need to handle this launch code, since the default action is the preferred alternative.

The application can respond to this launch code by setting the `result` field in the parameter block to the appropriate value. If it wants to allow the exchange manager to display a dialog, it should leave the `result` field set to `exgAskDialog` (the default value). To disable display of the dialog and to automatically accept the incoming data (as if the user had pressed OK in the dialog), set the `result` field to `exgAskOk`. To disable display of the dialog and to automatically reject the incoming data (as if the user had pressed Cancel in the dialog), set the `result` field to `exgAskCancel`. In the later case, the data is discarded and no further action is taken by the exchange manager.

If the application sets the result field to `exgAskOk`, or the dialog is displayed and the user presses the OK button, then the exchange manager sends the application the next launch code, `sysAppLaunchCmdExgReceiveData`, so that it can actually receive the data. This launch code notifies the application that it should receive the data.

The application should use the exchange manager functions `ExgAccept`, `ExgReceive`, and `ExgDisconnect` to receive the data and store it or do whatever it needs to with the data.

The parameter block sent with this launch code is of the `ExgSocketPtr` data type. It is a pointer to the `ExgSocketType` structure corresponding to the exchange manager connection via which the data is arriving. You will need to pass this pointer to the `ExgAccept` function to begin receiving the data. Note that in the socket structure, the `length` field may not be accurate, so in your receive loop you should be flexible in handling more or less data than `length` specifies.

After you have finished receiving the data and before you return from the `PilotMain` routine, you must set up the `goToCreator` and `goToParams` fields in the socket structure. Set in the `goToCreator` field the creator ID of the application that should be launched to view the received data (normally the same application that received the data). If no application should be launched, then set this to NULL. Set in the `goToParams` structure information that identifies the record to go to when the application is launched. It is recommended that you use a unique ID to identify the record, rather than the record index, since indexes might change. You can put unique ID information into the `goToParams.matchCustom` field.

Note that the application may not be the active application, and thus may not have globals available when it is launched with this launch code. Be sure to check if you have globals available and don't try to access them if they are not available.

Assuming that everything has proceeded normally, the exchange manager again launches the application identified in the `goToCreator` field of the socket structure with the `sysAppLaunchCmdGoto` launch code. This allows the user to view the received item.

Exchange Manager Function Summary

The following functions are available for application use:

- `ExgAccept`
- `ExgDBRead`
- `ExgDBWrite`
- `ExgDisconnect`
- `ExgPut`
- `ExgReceive`
- `ExgRegisterData`
- `ExgSend`

Exchange Manager Functions

ExgAccept

Purpose Accepts a connection from a remote device.

Prototype `Err ExgAccept (ExgSocketPtr socketP)`

Parameters --> `socketP` Pointer to the socket structure.

Result Returns the following result codes:

0	No error
`exgErrBadLibrary`	Couldn't find default exchange library
`exgErrStackInit`	Couldn't initialize the IR stack (not enough battery power or unsupported hardware)

Comments An application calls this function when it has been called with the special application launch code `sysAppLaunchCmdExgReceiveData`. The application is passed `socketP` as a parameter and it should pass this parameter to `ExgAccept` to accept the connection. Then call `ExgReceive` one or more times to receive the data.

See Also `ExgReceive`

ExgDBRead

Purpose Reads a Palm OS database in its internal format and writes it to storage RAM. For example, this function might read in a database transmitted by a beaming operation using the exchange manager.

Prototype

```
Err ExgDBRead (ExgDBReadProcPtr readProcP,
               ExgDBDeleteProcPtr deleteProcP,
               void* userDataP,
               LocalID* dbIDP,
               Int cardNo,
               Boolean* needResetP,
               Boolean keepDates)
```

Parameters

--> `readProcP`	A pointer to a function that you supply that reads in the database and passes it to `ExgDBRead`. See the Comments section for details.
--> `deleteProcP`	A pointer to a function that is called if a database with an identical name already exists on the device, so you can erase it before `ExgDBRead` stores the received database. See the Comments section for details.
--> `userDataP`	A pointer to any data you want to pass to either the `readProcP` or `deleteProcP` functions.
<-- `dbIDP`	The id of the database that `ExgDBRead` created on the local device.
<-- `cardNo`	The number of the card on which the database was stored by `ExgDBRead`.
<-- `needResetP`	Set to `TRUE` by `ExgDBRead` if the `dmHdrAttrResetAfterInstall` attribute bit is set in the received database.
--> `keepDates`	Specify `TRUE` to retain the creation, modification, and last backup dates as set in the received database header. Specify `FALSE` to reset these dates to the current date.

Result Returns 0 if successful; otherwise, returns one of the data manager error codes (`dmErr...`) or a callback-specific error code (if the `readProcP` function returns an error, it is also returned by `ExgDBRead`).

Comments The `readProcP` parameter points to a function that you supply and that is called by `ExgDBRead` to read in a database. The read callback function is called with three parameters, as follows:

--> `void* dataP` A pointer to a buffer where this function should place the database data.

<--> `ULong* sizeP`
The size of `dataP`. This value is set by `ExgDBRead` to the number of bytes it expects to receive in `dataP`. You must set this value to the number of bytes you return in `dataP` (if it's not the same).

--> `void* userDataP`
The `userDataP` parameter passed to `ExgDBRead` is simply passed on to the read function. You can use it for application-specific data.

The read callback function should return an error number, or 0 if there is no error. If the callback function returns an error, `ExgDBRead` deletes the database it was creating, cleans up any memory it allocated, then exits, returning the error passed back from the callback function.

The read callback function is called multiple times by `ExgDBRead`. Each time, it passes in `sizeP` the number of bytes it expects to receive in the next chunk you are to return in `dataP`. In `sizeP`, it's important to set the number of bytes that you actually place in `dataP`, if it's not the same as what `ExgDBRead` expected. `ExgDBRead` stops calling the read callback function after it receives the entire database (it knows when it's got it all based on the header information).

The `deleteProcP` function is called if `ExgDBRead` finds that an identically named database already exists on the local device. This delete callback function gives you a chance to delete the existing database, or take some other action (such as changing the database name, if appropriate).

The delete callback function is called with five parameters, as follows:

`const char* nameP`
: A pointer to the name of the identical database that already exists.

`Word version`
: The version of the identical database that already exists.

`Int cardNo`
: The card number of the identical database that already exists.

`LocalID dbID`
: The database ID of the identical database that already exists.

`void* userDataP`
: The `userDataP` parameter passed to `ExgDBRead` is simply passed on to the delete function. You can use it for application-specific data.

The delete callback function should return a Boolean value. `TRUE` means that the delete callback function handled the situation successfully; that is, it deleted, renamed, or moved the database so there would no longer be a conflict with the one that `ExgDBRead` is writing. `FALSE` means that the delete callback function did not handle the situation successfully; in this case, `ExgDBRead` exits with no error (same as if the user cancelled the operation).

See Also `ExgDBWrite`

ExgDBWrite

Purpose Reads a given Palm OS database in its internal format from the local device and writes it out using a function you supply. For example, this function might read a local database and transmit it by a beaming operation using the exchange manager.

Prototype

```
Err ExgDBWrite (ExgDBWriteProcPtr writeProcP,
                void* userDataP,
                const char* nameP,
                LocalID dbID,
                Int cardNo)
```

Parameters

--> `writeProcP`	A pointer to a function that you supply that writes out the database identified by `dbID`. See the Comments section for details.
--> `userDataP`	A pointer to any data you want to pass to the `writeProcP` function.
--> `nameP`	A pointer to the name of the database that you want `ExgDBWrite` to read and pass to `writeProcP`.
--> `dbID`	The id of the database that you want `ExgDBWrite` to read and pass to `writeProcP`. If you don't supply an ID, then `nameP` is used to search for the database by name.
--> `cardNo`	The number of the card on which to look for the database identified by `nameP`.

Result

Returns 0 if successful; otherwise, returns one of the data manager error codes (`dmErr...`) or a callback-specific error code (if the `writeProcP` function returns an error, it is also returned by `ExgDBWrite`).

Comments

The `writeProcP` parameter points to a function that you supply and that is called by `ExgDBWrite` to write out a database. For example, you might use this function to call exchange manager functions to beam the database to another unit.

The write callback function is called with three parameters, as follows:

--> `void* dataP`	A pointer to a buffer containing the database data, placed there by `ExgDBWrite`.

<--> `ULong* sizeP`
: The number of bytes placed in `dataP` by `ExgDBWrite`. If you were unable to write out or send all of the data in this chunk, on exit, you should set `sizeP` to the number of bytes you did write out.

--> `void* userDataP`
: The `userDataP` parameter passed to `ExgDBWrite` is simply passed on to the write function. You can use it for application-specific data.

The write callback function should return an error number, or 0 if there is no error. If the callback function returns an error, `ExgDBWrite` closes the database it was reading, cleans up any memory it allocated, then exits, returning the error passed back from the callback function.

The write callback function is called multiple times by `ExgDBWrite`. In the `sizeP` parameter, `ExgDBWrite` passes the number of bytes in `dataP`. Due to transport errors, timeouts, or other problems, you may not be able to successfully send all this data. If you didn't handle it all, it's important to set in `sizeP` the number of bytes that you did handle successfully. `ExgDBWrite` stops calling the write callback function after you write out the entire database (it knows when you've done it all based on the header information and number of bytes you return in `sizeP` each time).

See Also `ExgDBRead`

ExgDisconnect

Purpose Terminates an exchange manager transfer and disconnects.

Prototype

```
Err ExgDisconnect(ExgSocketPtr socketP, Err error)
```

Parameters

--> `socketP`	Pointer to the socket structure identifying the connection to terminate.
--> `error`	Any application error that occurred.

Result Returns the following result codes:

0	No error
`exgErrBadLibrary`	Couldn't find default exchange library
`exgMemError`	Couldn't read data to send
`exgErrUserCancel`	User cancelled transfer

Comments In the `error` parameter, pass any error that occurs during the application loop, including errors returned from other exchange manager functions. This ensures that the connection is shut down knowing that it failed rather than succeeded.

It's especially important to check the result code from this function, since this will tell you if the transfer was successful. A 0 return value means that the item was delivered to the destination successfully. It does not mean that the user on the other end actually kept the data.

`ExgDisconnect` is used for sending and receiving. When receiving, the application can insert its creator ID into the `goToCreator` field in the socket structure and add other goto information. After the application returns from the `sysAppLaunchCmdExgReceiveData` call, the system will launch the application with a standard `sysAppLaunchCmdGoto` launch code built from the information in the socket header `gotoParams` field.

See Also `ExgPut`, `ExgReceive`, `ExgSend`

ExgPut

Purpose Initiates the transfer of data to the destination device.

Prototype `Err ExgPut (ExgSocketPtr socketP)`

Parameters --> `socketP` Pointer to the socket structure containing connection information and information identifying the object to send.

Result Returns the following result codes:

0	No error
`exgErrBadLibrary`	Couldn't find default exchange library
`exgErrStackInit`	Couldn't initialize the IR stack (not enough battery power or unsupported hardware)
`exgMemError`	Not enough memory to initialize transfer

Comments If the connection does not already exist, this function establishes one. You must create and pass a pointer to an `ExgSocketType` structure containing information about the data to send and the destination application. All unused fields in the structure MUST be zeroed.

If no error is returned, this call MUST be followed by `ExgSend`, to begin sending data, or `ExgDisconnect`, to disconnect. You may need to call `ExgSend` multiple times to send all the data.

See Also `ExgDisconnect`, `ExgSend`

ExgReceive

Purpose Receives data from a remote device.

Prototype

```
ULong ExgReceive (ExgSocketPtr socketP,
                  VoidPtr bufP,
                  const ULong bufLen,
                  Err * errP)
```

Parameters

-->	`socketP`	Pointer to the socket structure.
-->	`bufP`	Pointer to the buffer to receive the data.
-->	`bufLen`	Number of bytes to receive.
<--	`errP`	Pointer to an error code result.

Result Returns the number of bytes actually received. A zero result indicates the end of the transmission. An error code is returned in the address indicated by `err`. The error code `exgErrUserCancel` is returned if the user cancels the operation.

Comments Call this function one or more times to receive all the data, following a successful call to `ExgAccept`. After receiving the data, call `ExgDisconnect` to terminate the connection.

This function blocks the application until the end of the transmission or until the requested number of bytes has been received. However, it does provide its own user interface that will be updated as necessary and will allow the user to cancel the operation in progress.

See Also `ExgAccept`, `ExgDisconnect`

ExgRegisterData

Purpose Registers an application to receive a specific type of data.

Prototype

```
Err ExgRegisterData ( const DWord creatorID,
                      const Word id,
                      const Char * const
dataTypesP)
```

Parameters

--> `creatorID`	Creator ID of the registering application.
--> `id`	Registry ID identifying the type of the items being registered. Specify `exgRegExtensionID` or `exgRegTypeID`.
--> `dataTypesP`	Pointer to a tab-delimited, null-terminated string listing the items to register. These include file extensions or MIME types. To unregister, pass a null value.

Result Returns 0 if successful, otherwise, one of the data manager error codes (`dmErr`...).

Comments

Applications that wish to receive data from anything other than another Palm OS device running the same application, must use this function to register for the kinds of data they can receive. Call this function when your application is loaded on the device.

Specify the `exgRegExtensionID` id to register to receive data that has a filename with a particular extension. For example, if your application wants to receive files with a .TXT extension, it could register like this:

```
ExgRegisterData(myCreator, exgRegExtensionID,
  "TXT");
```

Specify the `exgRegTypeID` id to register to receive data with a specific MIME type. For example, if your application wants to receive "setext" text files, it could register like this:

```
ExgRegisterData(myCreator, exgRegTypeID,
  "text/x-setext");
```

Registrations are active until the device is hard reset or until the application is removed. The registration information is backed up and restored across a soft reset. When an application is removed, its registry information is also automatically removed from the registry, so there is not normally a need to unregister. If you want to unregister, you can register with a nil value.

ExgSend

Purpose Sends data to the destination device.

Prototype

```
ULong ExgSend (ExgSocketPtr socketP,
               const void * const bufP,
               const ULong bufLen,
               Err * errP)
```

Parameters

--> `socketP`	Pointer to the socket structure.
--> `bufP`	Pointer to the data to send.
--> `bufLen`	Number of bytes to send.
<-- `errP`	Pointer to an error code result.

Result Returns the number of bytes sent, normally the same number as specified in `bufLen`. An error code is returned in the address indicated by `err`. The error code `exgErrUserCancel` is returned if the user cancels the operation.

Comments Call this function one or more times to send all the data, following a successful call to `ExgPut`. After sending the data, call `ExgDisconnect` to terminate the connection.

The lower level protocol may break large amounts of data into multiple packets or assemble small send commands together into larger packets, but the application will not be aware of these transport level details.

This function blocks the application until all the data is sent. However, it does provide its own user interface that will be updated as necessary and will allow the user to cancel the operation in progress.

See Also `ExgDisconnect`, `ExgPut`

9

IR Library

The IR (InfraRed) library is a shared library that provides a direct interface to the IR communications capabilities of the Palm OS. It is designed for applications that want more direct access to the IR capabilities than the exchange manager provides.

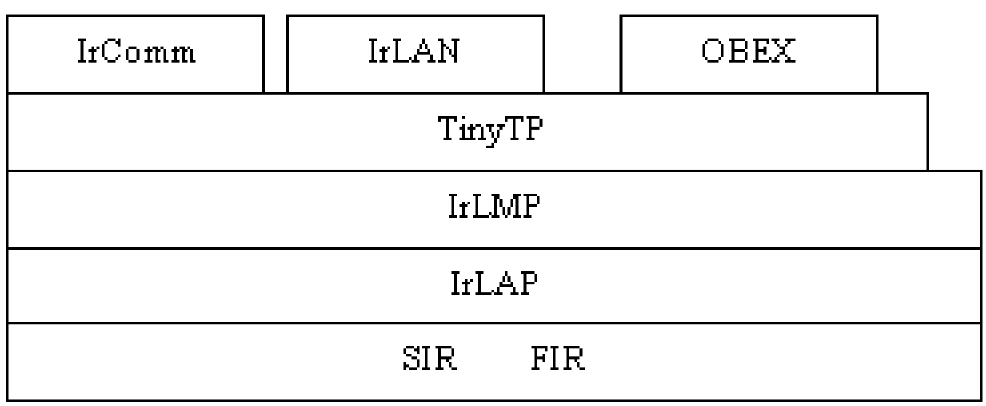
The IR support provided by the Palm OS is compliant with the IrDA specifications. IrDA (Infrared Data Association), is an industry body consisting of representatives from a number of companies involved in IR development. For a good introduction to the IrDA standards, see the IrDA web site at:

`http://www.IrDA.org.`

IrDA Stack

The IrDA stack comprises a number of protocol layers, of which some are required and some are optional. The complete stack looks something like Figure 9.1.

Figure 9.1 IrDA Protocol Stack



The SIR/FIR layer is purely hardware. The SIR (Serial IR) layer supports speeds up to 115k bps while the FIR (Fast IR) layer supports speeds up to 4M bps. IrLAP is the IR Link Access Protocol that provides a data pipe between IrDA devices. IrLMP, the IR Link Management Protocol, manages multiple sessions using the IrLAP. Tiny TP is a lightweight transfer protocol on which some higher-level IrDA layers are built.

One or more of SIR/FIR must be implemented, and Tiny TP, IrLMP and IrLAP must also be implemented. IrComm provides serial and parallel port emulation over an IR link and is optional (it is not currently supported in the Palm OS). IrLAN provides an access point to Local Area Network protocol adapters. It too is optional (and is not supported in the Palm OS).

OBEX is an object exchange protocol that can be used (for instance) to transfer business cards, calendar entries or other objects between devices. It too is optional and is supported in the Palm OS. The capabilities of OBEX are made available through the exchange manager; there is no direct API for it.

The Palm OS implements all the required protocol layers (SIR, IrLAP, IrLMP, and Tiny TP), as well as the OBEX layer, to support the Exchange Manager. Palm III devices provide SIR (Serial IR) hardware supporting the following speeds: 2400, 9600, 19200, 38400, 57600, and 115200 bps. The software (`IrOpen`) currently limits bandwidth to 57600 bps by default, but you can specify a connection speed of up to 115200 bps if desired.

The stack is capable of connection-based or connectionless sessions.

IrLMP Information Access Service (IAS) is a component of the IrLMP protocol that you will see mentioned in the interface. IAS provides a database service through which devices can register information about themselves and retrieve information about other devices and the services they offer.

Loading the IR Library

Before you can use the IR library, you must obtain a reference number for it by calling the function `SysLibFind`, as in this example:

```
err = SysLibFind(irLibName, &refNum);
```

This function returns the library reference number in the `refNum` parameter. This parameter is passed to most of the other functions in the IR library.

IR Data Structures

This section lists some of the more important data types used by IR library functions.

IrConnect

This data structure is used to manage an IrLMP or Tiny TP connection.

Listing 9.1 IrConnect Data Structure

```
/* Forward declaration of the IrConnect structure */
typedef struct _hconnect IrConnect;

/*---------------------------------------------------------------
*/
typedef struct _hconnect {
Byte lLsap; /* Local LSAP this connection will listen on */
Byte rLsap; /* Remote Lsap */

/*============== For Internal Use Only =======================
 *
 * The following is used internally by the stack and should not
be
 * modified by the user.
 *
 *==============================================================*/
```

```
Byte flags; /* Flags containing state, type, etc. */
IrCallBack callBack; /* Pointer to callback function */

/* Tiny TP fields */
IrPacket packet; /* Packet for internal use */
ListEntry packets; /* List of packets to send */
Word sendCredit; /* Amount of credit from peer */
Byte availCredit; /* Amount of credit to give to peer */
Byte dataOff; /* Amount of data less than IrLAP size */
} _hconnect;
```

IrPacket

This data structure is used for sending IrDA packets.

Listing 9.2 IrPacket Data Structure

```
typedef struct _IrPacket {
/* The node field must be the first field in the structure. It is
 * used internally by the stack. */
ListEntry node;

/* The buff field is used to point to a buffer of data to send
and
 * len field indicates the number of bytes in buff. */
BytePtr buff;
Word len;

/*==================  For Internal Use Only
======================
 *
 * The following is used internally by the stack and should not
be
 * modified by the upper layer.
 *

*==============================================================*/
IrConnect* origin; /* Pointer to connection which owns packet */
```

```
Byte headerLen; /* Number of bytes in the header */
Byte header[14]; /* Storage for the header */
} IrPacket;
```

IrIASObject

This data structure is used as storage for an IAS object managed by the local IAS server. An object of this type is passed as the `obj` parameter to the `IrIAS_Add` function.

Listing 9.3 IrIASObject Data Structure

```
typedef struct _IrIasObject {
BytePtr name; /* Pointer to name of object */
Byte len; /* Length of object name */

Byte nAttribs; /* Number of attributes */
IrIasAttribute* attribs; /* A pointer to an array of attributes
*/
} IrIasObject;
```

IrIasQuery

This data structure is used for performing IAS queries. An object of this type is passed as the `token` parameter to the `IrIAS_Query` function (and several other functions as well).

Listing 9.4 IrIasQuery Data Structure

```
* Forward declaration of a structure used for performing IAS
* Queries so that a callback type can be defined for use in
* the structure. */
typedef struct _IrIasQuery IrIasQuery;
typedef void (*IrIasQueryCallBack)(IrStatus);

* Actual definition of the IrIasQuery structure. */
typedef struct _IrIasQuery
{
```

```
/* Query fields. The query buffer contains the class name and
 * class attribute whose value is being queried--it is as
follows:
 *
 * 1 byte - Length of class name
 * "Length" bytes - class name
 * 1 byte - length of attribute name
 * "Length" bytes - attribute name
 *
 * queryLen - contains the total number of byte in the query */
Byte queryLen; /* Total length of the query */
BytePtr queryBuf; /* Points to buffer containing the query */

/* Fields for the query result */
Word resultBufSize; /* Size of the result buffer */
Word resultLen; /* Actual number of bytes in the result buffer */
Word listLen; /* Number of items in the result list. */
Word offset; /* Offset into results buffer */
Byte retCode; /* Return code of operation */
Byte overFlow; /* Set TRUE if result exceeded result buffer
size*/
BytePtr  result; /* Pointer to buffer containing result; */

/* Pointer to callback function */
IrIasQueryCallBack callBack;
} _IrIasQuery;
```

IrCallbackParms

This data structure is used to pass information from the stack to the upper layer of the stack (application). Not all fields are valid at any given time. The type of event determines which fields are valid. An object of this type is passed as the second parameter to the `IrCallback` function.

Listing 9.5 IRCallbackParms Data Structure

```
typedef struct {
IrEvent event; /* Event causing callback */
```

```
BytePtr rxBuff; /* Receive buffer already advanced to app data */
Word rxLen; /* Length of data in receive buffer */
IrPacket* packet; /* Pointer to packet being returned */
IrDeviceList* deviceList; /* Pointer to discovery device list */
IrStatus status; /* Status of stack */
} IrCallBackParms;
```

IR Stack Callback Events

The IR stack calls the application via a callback function stored in each `IrConnect` structure. The callback function is called with a pointer to the `IrConnect` structure and a pointer to a parameter structure. The parameter structure contains an `event` field, which indicates the reason the callback is called, and other parameters, which have meaning based on the event.

The meaning of the events is described in the following sections.

LEVENT_DATA_IND

Data has been received. The received data is accessed using fields `rxBuff` and `rxLen`.

LEVENT_DISCOVERY_CNF

Indicates the completion of a discovery operation. The field `deviceList` points to the discovery list.

LEVENT_LAP_CON_CNF

The requested IrLAP connection has been made successfully. The callback function of all bound `IrConnect` structures is called.

LEVENT_LAP_CON_IND

Indicates that the IrLAP connection has come up. The callback of all bound `IrConnect` structures is called.

LEVENT_LAP_DISCON_IND

Indicates that the IrLAP connection has gone down. This means that all IrLMP connections are also down. A callback with event `LEVENT_LM_CON_IND` will not be given. The callback function of all bound `IrConnect` structures is called.

LEVENT_LM_CON_CNF

The requested IrLMP / Tiny TP connection has been made successfully. Connection data from the other side is found using fields `rxBuff` and `rxLen`.

LEVENT_LM_CON_IND

Other device has initiated a connection. `IrConnectRsp` should be called to accept the connection. Any data associated with the connection request can be found using fields `rxBuff` and `rxLen`, for the data pointer and length, respectively.

LEVENT_LM_DISCON_IND

The IrLMP / Tiny TP connection has been disconnected. Any data associated with the disconnect indication can be found using fields `rxBuff` and `rxLen`, for the data pointer and length, respectively.

LEVENT_PACKET_HANDLED

A packet is being returned. A pointer to the packet exists in field `packet`.

LEVENT_STATUS_IND

Indicates that a status event from the stack has occurred. The `status` field indicates the status generating the event. Possible statuses are as follows.

- `IR_STATUS_NO_PROGRESS` means that IrLAP has no progress for 3 seconds threshold time (e.g. the beam is blocked).

- IR_STATUS_LINK_OK indicates that the no progress condition has cleared.
- IR_STATUS_MEDIA_NOT_BUSY indicates that the IR media has transitioned from busy to not busy.

LEVENT_TEST_CNF

Indicates that a TEST command has completed. The status field indicates if the test was successful. IR_STATUS_SUCCESS indicates that operation was successful and the data in the test response can be found by using the rxBuff and rxLen fields. IR_STATUS_FAILED indicates that no TEST response was received. The packet passed to perform the test command is passed back in the packet field and is now available (no separate packet handled event will occur).

LEVENT_TEST_IND

Indicates that a TEST command frame has been received. A pointer to the received data is in rxBuff and rxLen. A pointer to the packet that will be sent in response to the test command is in the packet field. The packet is currently set up to respond with the same data sent in the command TEST frame. If different data is desired as a response, then modify the packet structure. This event is sent to the callback function in all bound IrConnect structures. The IAS connections ignore this event.

IAS Query Callback Function

The result of IAS queries is signaled by calling the callback function pointed to by the callBack field of the IrIasQuery structure. The callback has the following prototype:

```
void callBack(IrStatus);
```

The callback is called with a status as follows:

IR_STATUS_SUCCESS means the query operation finished successfully and the results can be parsed.

IR_STATUS_DISCONNECT means the link or IrLMP connection was disconnected during the query, so the results are not valid.

IR Library Function Summary

The following general functions are available for application use:

- IrAdvanceCredit
- IrBind
- IrClose
- IrConnectIrLap
- IrConnectReq
- IrConnectRsp
- IrDataReq
- IrDisconnectIrLap
- IrDiscoverReq
- IrIsIrLapConnected
- IrIsMediaBusy
- IrIsNoProgress
- IrIsRemoteBusy
- IrLocalBusy
- IrMaxRxSize
- IrMaxTxSize
- IrOpen
- IrSetConTypeLMP
- IrSetConTypeTTP
- IrSetDeviceInfo
- IrTestReq
- IrUnbind

The following functions and macros are related to IAS databases:

- IrIAS_Add

IR Library Functions

IrAdvanceCredit

Purpose Advances credit to the other side of the connection.

Prototype `void IrAdvanceCredit (IrConnect* con, Byte credit)`

Parameters

--> `con`	Pointer to `IrConnect` structure representing connection to which credit is advanced.
--> `credit`	Amount of credit to advance.

Result Returns nothing.

Comments The total amount of credit should not exceed 127. The credit passed by this function is added to the existing available credit, which is must not exceed 127. This function only makes sense for a Tiny TP connection.

IrBind

Purpose Obtains a local LSAP selector and registers the connection with the protocol stack.

Prototype

```
IrStatus IrBind (UInt refNum,
                 IrConnect* con,
                 IrCallBack callBack)
```

Parameters

--> `refnum`	IR library refNum.
<--> `con`	Pointer to `IrConnect` structure.
--> `callBack`	Pointer to a callBack function that handles the indications and confirmation from the protocol stack.

Result `IR_STATUS_SUCCESS` means the operation completed successfully. The assigned LSAP can be found in `con->lLsap`.

`IR_STATUS_FAILED` means the operation failed for one of the following reasons:

- `con` is already bound to the stack
- no room in the connection table

Comments This `IrConnect` structure will be initialized. Any values stored in the structure will be lost. The assigned LSAP will be in the `lLsap` field of `con`. The type of the connection will be set to IrLMP. The `IrConnect` must be bound to the stack before it can be used.

IrClose

Purpose Closes the IR library. This releases the global memory for the IR stack and any system resources it uses. This must be called when an application is done with the IR library.

Prototype

```
Err IrClose (Word refnum)
```

Parameters

-->	`refnum`	IR library refNum.

Result Returns 0 if successful.

IrConnectIrLap

Purpose Starts an IrLAP connection.

Prototype

```
IrStatus IrConnectIrLap (UInt refNum,
                         IrDeviceAddr deviceAddr)
```

Parameters

-->	`refnum`	IR library refNum.
-->	`deviceAddr`	32-bit address of device to which connection should be made.

Result `IR_STATUS_PENDING` means the operation is started successfully; the result is returned via callback.

`IR_STATUS_MEDIA_BUSY` means the operation failed because the media is busy. Media busy is caused by one of the following reasons:

- Other devices are using the IR medium.
- An IrLAP connection already exists.
- A discovery process is in progress.

Comments The result is signaled to all bound `IrConnect` structures via the callback function. The callback event is `LEVENT_LAP_CON_CNF` if successful or `LEVENT_LAP_DISCON_IND` if unsuccessful.

IrConnectReq

Purpose Requests an IrLMP or Tiny TP connection.

Prototype

```
IrStatus IrConnectReq ( UInt refNum,
                        IrConnect* con,
                        IrPacket* packet,
                        Byte credit)
```

Parameters

--> `refnum` IR library refNum.

--> `con` Pointer to `IrConnect` structure for handling the connection. The `rLsap` field must contain the LSAP selector for the peer on the other device. Also the type of the connection must be set. Use `IR_SetConTypeLMP` to set the type to an IrLMP connection or `IR_SetConTypeTTP` to set the type to a Tiny TP connection.

--> `packet` Pointer to a packet that contains connection data. Even if no connection data is needed, the packet must point to a valid `IrPacket` structure. The packet will be returned via the callback with the `LEVENT_PACKET_HANDLED` event if no errors occur. The maximum size of the packet is `IR_MAX_CON_PACKET` for an IrLMP connection or `IR_MAX_TTP_CON_PACKET` for a Tiny TP connection.

--> `credit` Initial amount of credit advanced to the other side. Must be less than 127. It is ANDed with 0x7f, so if it is greater than 127 unexpected results will occur. This parameter is ignored if the connection is an IrLMP connection.

Result `IR_STATUS_PENDING` means the operation has been started successfully and the result will be returned via the callback function with the event `LEVENT_LM_CON_CNF` if the connection is made or

LEVENT_LM_DISCON_IND if connection fails. The packet is returned via the callback with the event LEVENT_PACKET_HANDLED.

IR_STATUS_FAILED means the operation failed because of one of the following reasons. Note that the packet is available immediately.

- Connection is busy (already involved in a connection)
- IrConnect structure is not bound to the stack
- Packet size exceeds maximum allowed

IR_STATUS_NO_IRLAP means the operation failed because there is no IrLAP connection (the packet is available immediately).

Comments

The result is signaled via the callback specified in the IrConnect structure. The callback event is LEVENT_LM_CON_CNF indicates that the connection is up and LEVENT_LM_DISCON_IND indicates that the connection failed. Before calling this function the fields in the con structure must be properly set.

IrConnectRsp

Purpose Accepts an incoming connection that has been signaled via the callback with the event LEVENT_LM_CON_IND.

Prototype

```
IrStatus IrConnectRsp ( UInt refNum,
                        IrConnect* con,
                        IrPacket* packet,
                        Byte credit)
```

Parameters

--> refnum	IR library refNum.
--> con	Pointer to IrConnect structure to managed connection.
--> packet	Pointer to a packet that contains connection data. Even if no connection data is needed, the packet must point to a valid IrPacket structure. The packet will be returned via the callback with the LEVENT_PACKET_HANDLED event if no errors occur. The maximum size of the packet is IR_MAX_CON_PACKET for an IrLMP connection or IR_MAX_TTP_CON_PACKET for a Tiny TP connection.
--> credit	Initial amount of credit advanced to the other side. Must be less than 127. It is ANDed with 0x7f, so if it is greater than 127 unexpected results will occur. This parameter is ignored if the connection is an IrLMP connection.

Result IR_STATUS_PENDING means the operation has been started successfully and the packet will be returned via the callback function with the event LEVENT_PACKET_HANDLED.

IR_STATUS_FAILED means the operation failed because of one of the following reasons. Note that the packet is available immediately.

- Connection is not in the proper state to require a response
- IrConnect structure is not bound to the stack

- Packet size exceeds maximum allowed

IR_STATUS_NO_IRLAP means the operation failed because there is no IrLAP connection (the packet is available immediately).

Comments IrConnectRsp can be called during the callback or later to accept the connection. The type of the connection must already have been set to IrLMP or Tiny TP before the LEVENT_LM_CON_IND event.

IrDataReq

Purpose Sends a data packet.

Prototype

```
IrStatus IrDataReq ( UInt refNum,
                     IrConnect* con,
                     IrPacket* packet)
```

Parameters

--> refnum	IR library refNum.
--> con	Pointer to IrConnect structure that specifies the connection over which the packet should be sent.
--> packet	Pointer to a valid IrPacket structure that contains data to send. The packet should not exceed the max size found with IrMaxTxSize.

Result IR_STATUS_PENDING means the packet has been queued by the stack. The packet will be returned via the callback with event LEVENT_PACKET_HANDLED.

IR_STATUS_FAILED means the operation failed because of one of the following reasons. Note that the packet is available immediately.

- IrConnect structure is not bound to the stack
- Packet size exceeds maximum allowed
- IrConnect structure does not represent an active connection

Comments The packet is owned by the stack until it is returned via the callback with event LEVENT_PACKET_HANDLED. The largest packet that can be sent is found by calling IrMaxTxSize.

IrDisconnectIrLap

Purpose Disconnects an IrLAP connection.

Prototype `IrStatus IrDisconnectIrLap (UInt refNum)`

Parameters --> `refnum` IR library refNum.

Result `IR_STATUS_PENDING` means the operation started successfully and all bound `IrConnect` structures will be called back when complete.

`IR_STATUS_NO_IRLAP` means the operation failed because no IrLAP connection exists.

Comments When the IrLAP connection goes down, the callback of all bound `IrConnect` structures is called with event `LEVENT_LAP_DISCON_IND`.

IrDiscoverReq

Purpose Starts an IrLMP discovery process.

Prototype

```
IrStatus IrDiscoverReq (UInt refNum,
                        IrConnect* con)
```

Parameters

--> `refnum`	IR library refNum.
--> `con`	Pointer to a bound `IrConnect` structure.

Result `IR_STATUS_PENDING` means the operation is started successfully; the result is returned via callback.

`IR_STATUS_MEDIA_BUSY` means the operation failed because the media is busy. Media busy is caused by one of the following reasons:

- Other devices are using the IR medium.
- A discovery process is already in progress.
- An IrLAP connection exists.

`IR_STATUS_FAILED` means the operation failed because the `IrConnect` structure is not bound to the stack.

Comments The result will be signaled via the callback function specified in the `IrConnect` structure with the event `LEVENT_DISCOVERY_CNF`. Only one discovery can be invoked at a time.

IrIsIrLapConnected

Purpose Determines if an IrLAP connection exists.

Prototype `BOOL IrIsIrLapConnected (UInt refNum)`

Parameters --> `refnum` IR library refNum.

Result True if IrLAP is connected, false otherwise.

Comments Only available if `IR_IS_LAP_FUNCS` is defined.

IrIsMediaBusy

Purpose Determines if the IR media is busy.

Prototype `BOOL IrIsMediaBusy (UInt refNum)`

Parameters --> `refnum` IR library refNum.

Result True if IR media is busy, false otherwise.

IrIsNoProgress

Purpose Determines if IrLAP is not making progress.

Prototype `BOOL IrIsNoProgress (UInt refNum)`

Parameters --> `refnum` IR library refNum.

Result True if IrLAP is not making progress, false otherwise.

IrIsRemoteBusy

Purpose Determines if the other device's IrLAP is busy.

Prototype `BOOL IrIsRemoteBusy (UInt refNum)`

Parameters --> `refnum` IR library refNum.

Result True if the other device's IrLAP is busy, false otherwise.

IrLocalBusy

Purpose Sets the IrLAP local busy flag.

Prototype `void IrLocalBusy (UInt refNum, BOOL flag)`

Parameters

--> `refnum` IR library refNum.

--> `flag` Value (true or false) to set for IrLAP's local busy flag.

Result Returns nothing.

Comments If local busy is set to true, then the local IrLAP layer will send RNR (Receive Not Ready) frames to the other side indicating it cannot receive any more data. If the local busy is set to false, IrLAP is ready to receive frames.

The setting takes effect the next time IrLAP sends an RR (Receive Ready) frame. If IrLAP has data to send, the data will be sent first, so it should be used carefully.

This function should not be used when using Tiny TP or when multiple connections exist.

IrMaxRxSize

Purpose Returns the maximum size buffer that can be sent by the other device.

Prototype `Word IrMaxRxSize (UInt refNum, IrConnect* con)`

Parameters

--> `refnum`	IR library refNum.
--> `con`	Pointer to `IrConnect` structure that represents an active connection.

Result Returns the maximum size buffer that can be sent by the other device (maximum bytes that can be received). The value returned is only valid for active connections. The maximum size will vary for each connection and is based on the negotiated IrLAP parameters and the type of the connection.

IrMaxTxSize

Purpose Returns the maximum size allowed for a transmit packet.

Prototype `Word IrMaxTxSize (UInt refNum, IrConnect* con)`

Parameters

--> `refnum`	IR library refNum.
--> `con`	Pointer to `IrConnect` structure that represents an active connection.

Result Returns the maximum size allowed for a transmit packet. The value returned is only valid for active connections. The maximum size will vary for each connection and is based on the negotiated IrLAP parameters and the type of the connection.

IrOpen

Purpose Opens the IR library. This allocates the global memory for the IR stack and reserves the system resources it requires. This must be done before any other IR library calls are made.

Prototype `Err IrOpen (Word refnum, DWord options)`

Parameters

--> `refnum`	IR library refNum. This value is returned from the function `SysLibFind`, which you must call first to load the IR library.
--> `options`	Open options flags. See the Comments section for details.

Result Returns 0 if successful.

Comments The following flags can be specified for the `options` parameter to set the speed of the connection:

`irOpenOptSpeed115200`	Set maximum negotiated baud rate
`irOpenOptSpeed57600`	Set 57600 bps (default if no flags given)
`irOpenOptSpeed9600`	Set 9600 bps

IrSetConTypeLMP

Purpose Sets the type of the connection to IrLMP. This function must be called after the `IrConnect` structure is bound to the stack.

Prototype `void IrSetConTypeLMP (IrConnect* con)`

Parameters

--> `con`	Pointer to `IrConnect` structure.

Result Returns nothing.

IrSetConTypeTTP

Purpose

Sets the type of the connection to Tiny TP. This function must be called after the `IrConnect` structure is bound to the stack.

Prototype

```
void IrSetConTypeTTP (IrConnect* con)
```

Parameters

--> `con`	Pointer to `IrConnect` structure.

Result

Returns nothing.

IrSetDeviceInfo

Purpose

Sets the XID info string used during discovery to the given string and length.

Prototype

```
IrStatus IrSetDeviceInfo ( UInt refNum,
                           BytePtr info,
                           Byte len)
```

Parameters

--> `refnum`	IR library refNum.
--> `info`	Pointer to array of bytes.
--> `len`	Number of bytes pointed to by `info`.

Result

`IR_STATUS_SUCCESS` means the operation is successful.

`IR_STATUS_FAILED` means the operation failed because `info` is too big.

Comments

The XID info string contains hints and the nickname of the device. The size cannot exceed `IR_MAX_DEVICE_INFO` bytes.

IrTestReq

Purpose Requests a TEST command frame be sent in the NDM (Normal disconnect Mode) state.

Prototype

```
IrStatus IrTestReq( UInt refNum,
                    IrDeviceAddr devAddr,
                    IrConnect* con,
                    IrPacket* packet)
```

Parameters

--> `refnum`	IR library refNum.
--> `devAddr`	Device address of device where TEST will be sent. This address is not checked so it can be the broadcast address or 0.
--> `con`	Pointer to `IrConnect` structure specifying the callback function to call to report the result.
--> `packet`	Pointer to an `IrPacket` structure that contains the data to send in the TEST command packet. The maximum size data that can be sent is `IR_MAX_TEST_PACKET`. Even if no data is to be sent, a valid packet must be passed.

Result `IR_STATUS_PENDING` means the operation has been started successfully and the result will be returned via the callback function with the event `LEVENT_TEST_CNF`. This is also the indication returning the packet.

`IR_STATUS_FAILED` means the operation failed because of one of the following reasons. Note that the packet is available immediately.

- `IrConnect` structure is not bound to the stack
- Packet size exceeds maximum allowed

`IR_STATUS_MEDIA_BUSY` means the operation failed because the media is busy or the stack is not in the NDM state (the packet is available immediately).

Comments The result is signaled via the callback specified in the `IrConnect` structure. The callback event is `LEVENT_TEST_CNF` and the status field indicates the result of the operation. `IR_STATUS_SUCCESS` indicates success and `IR_STATUS_FAILED` indicates no response was received. A packet must be passed containing the data to send in the TEST frame. The packet is returned when the `LEVENT_TEST_CNF` event is given.

IrUnbind

Purpose Unbinds the `IrConnect` structure from the protocol stack, freeing it's LSAP selector.

Prototype `IrStatus IrUnbind (UInt refNum, IrConnect* con)`

Parameters

--> `refnum`	IR library refNum.
--> `con`	Pointer to `IrConnect` structure to unbind.

Result `IR_STATUS_SUCCESS` means the operation completed successfully.

`IR_STATUS_FAILED` means the operation failed for one of the following reasons:

- the `IrConnect` structure was not bound
- the `lLsap` field contained an invalid number

IAS Functions

This section describes functions and macros related to IAS databases:

- `IrIAS_Add`
- `IrIAS_GetInteger`
- `IrIAS_GetIntLsap`
- `IrIAS_GetObjectID`
- `IrIAS_GetOctetString`

IrIAS_Add

Purpose Adds an IAS object to the IAS Database.

Prototype `IrStatus IrIAS_Add (UInt refNum, IrIasObject* obj)`

Parameters

--> `refnum`	IR library refNum.
--> `obj`	Pointer to an `IrIASObject` structure.

Result `IR_STATUS_SUCCESS` means the operation is successful.

IR_STATUS_FAILED means the operation failed for one of the following reasons:

- No space in the database.
- An entry with the same class name already exists.
- The attributes of the object violate the IrDA Lite rules (attribute name exceeds `IR_MAX_IAS_NAME`, or attribute value exceeds `IR_MAX_IAS_ATTR_SIZE`).
- The class name exceeds `IR_MAX_IAS_NAME`.

Comments The object is not copied, so the memory for the object must exist for as long as the object is in the database. The IAS database is designed to allow only objects with unique class names, and it checks for this. Class names and attributes names must not exceed

IR_MAX_IAS_NAME. Also, attribute values must not exceed IR_MAX_IAS_ATTR_SIZE.

IrIAS_GetInteger

Purpose Returns an integer value, assuming that the current result item is of type IAS_ATTRIB_INTEGER. (Call IrIAS_GetType to determine the type of the current result item.)

Prototype `DWord IrIAS_GetInteger (IrIasQuery* token)`

Parameters --> token Pointer to an IrIasQuery structure.

Result Integer value.

IrIAS_GetIntLsap

Purpose Returns an integer value that represents an LSAP, assuming that the current result item is of type IAS_ATTRIB_INTEGER. (Call IrIAS_GetType to determine the type of the current result item.) Usually integer values returned in a query are LSAP selectors.

Prototype `Byte IrIAS_GetIntLsap (IrIasQuery* token)`

Parameters --> token Pointer to an IrIasQuery structure.

Result Integer value.

IrIAS_GetObjectID

Purpose Returns the unique object ID of the current result item.

Prototype `Word IrIAS_GetObjectID (IrIasQuery* token)`

Parameters --> `token` Pointer to an `IrIasQuery` structure.

Result Returns the object ID.

IrIAS_GetOctetString

Purpose Returns a pointer to an octet string, assuming that the current result item is of type IAS_ATTRIB_OCTET_STRING. (Call `IrIAS_GetType` to determine the type of the current result item.)

Prototype `Byte* IrIAS_GetOctetString (IrIasQuery* token)`

Parameters --> `token` Pointer to an `IrIasQuery` structure.

Result Pointer to octet string.

IrIAS_GetOctetStringLen

Purpose Gets the length of an octet string, assuming that the current result item is of type `IAS_ATTRIB_OCTET_STRING`. (Call `IrIAS_GetType` to determine the type of the current result item.)

Prototype `Word IrIAS_GetOctetStringLen (IrIasQuery* token)`

Parameters --> `token` Pointer to an `IrIasQuery` structure.

Result Length of octet string.

IrIAS_GetType

Purpose Returns the type of the current result item.

Prototype `Byte IrIAS_GetType (IrIasQuery* token)`

Parameters --> `token` Pointer to an `IrIasQuery` structure.

Result Type of result item such as `IAS_ATTRIB_INTEGER`, `IAS_ATTRIB_OCTET_STRING` or `IAS_ATTRIB_USER_STRING`.

IrIAS_GetUserString

Purpose Returns a pointer to a user string, assuming that the current result item is of type `IAS_ATTRIB_USER_STRING`. (Call `IrIAS_GetType` to determine the type of the current result item.)

Prototype `Byte* IrIAS_GetUserString(IrIasQuery* token)`

Parameters --> `token` Pointer to an `IrIasQuery` structure.

Result Pointer to result string.

IrIAS_GetUserStringCharSet

Purpose Returns the character set of the user string, assuming that the current result item is of type `IAS_ATTRIB_USER_STRING`. (Call `IrIAS_GetType` to determine the type of the current result item.)

Prototype
```
IrCharSet IrIAS_GetUserStringCharSet
                        (IrIasQuery* token)
```

Parameters --> `token` Pointer to an `IrIasQuery` structure.

Result Character set.

IrIAS_GetUserStringLen

Purpose Gets the length of a user string, assuming that the current result item is of type `IAS_ATTRIB_USER_STRING`. (Call `IrIAS_GetType` to determine the type of the current result item.)

Prototype `Byte IrIAS_GetUserStringLen (IrIasQuery* token)`

Parameters

--> `token`	Pointer to an `IrIasQuery` structure.

Result Length of user string.

IrIAS_Next

Purpose Moves the internal pointer to the next result item.

Prototype `BytePtr IrIAS_Next (UInt refNum,IrIasQuery* token)`

Parameters

--> `refnum`	IR library refNum.
--> `token`	Pointer to an `IrIasQuery` structure.

Result Pointer to the next result item, or 0 if there are no more items.

Comments This function returns a pointer to the start of the next result item. If the pointer is 0, then there are no more result items.

IrIAS_Query

Purpose Makes an IAS query of another device's IAS database.

Prototype

```
IrStatus IrIAS_Query ( UInt refNum,
                       IrIasQuery* token)
```

Parameters

--> `refnum`	IR library refNum.
--> `token`	Pointer to an `IrIasQuery` structure initialized as described in the Comments section.

Result `IR_STATUS_SUCCESS` means the operation is started successfully and the result will be signaled via the callback function.

IR_STATUS_FAILED means the operation failed for one of the following reasons:

- The query exceeds `IR_MAX_QUERY_LEN`.
- The `result` field of `token` is 0.
- The `resultBufSize` field of `token` is 0.
- The `callback` field of `token` is 0.
- A query is already in progress.

`IR_STATUS_NO_IRLAP` means the operation failed because there is no IrLAP connection.

Comments An IrLAP connection must exist to the other device. The IAS query token must be initialized as described below. The result is signaled by calling the callback function whose pointer exists in the `IrIasQuery` structure. Only one query can be made at a time.

The `IrIasQuery` structure passed in the `token` parameter must be initialized as follows:

- pointer to a callback function in which the result will signaled.
- `result` points to a buffer large enough to hold the result of the query.
- `resultBufSize` is set to the size of the result buffer.

- queryBuf must point to a valid query.
- queryLen is set to the number of bytes in queryBuf. The length must not exceed IR_MAX_QUERY_LEN.

IrIAS_SetDeviceName

Purpose Sets the value field of the device name attribute of the “Device” object in the IAS database.

Prototype

```
IrStatus IrIAS_SetDeviceName ( UInt refNum,
                               BytePtr name,
                               Byte len)
```

Parameters

-->	refnum	IR library refNum.
-->	name	Pointer to an IAS value field for the device name attribute of the device object. It includes the attribute type, character set and device name. This value field should be a constant and the pointer must remain valid until IrIAS_SetDeviceName is called with another pointer.
-->	len	Total length of the value field. Maximum size allowed is IR_MAX_IAS_ATTR_SIZE.

Result IR_STATUS_SUCCESS means the operation is successful.

IR_STATUS_FAILED means len is too big, or the value field is not a valid user string.

IrIAS_StartResult

Purpose Puts the internal pointer to the start of the result buffer.

Prototype `void IrIAS_StartResult (IrIasQuery* token)`

Parameters --> `token` Pointer to an `IrIasQuery` structure.

Result Returns nothing.

Index

Index

L

M

N

O

T

U

V

W