Welcome to *Developing Palm OS Applications*

Part II: Memory and Communications Management

Navigate this online document as follows:

<table>
<thead>
<tr>
<th>To see bookmarks</th>
<th>Type Command-7</th>
</tr>
</thead>
<tbody>
<tr>
<td>To see information on Adobe Acrobat Reader</td>
<td>Type Command-?</td>
</tr>
<tr>
<td>To navigate</td>
<td>Click on any blue hypertext link any <em>Table of Contents</em> entry arrows in the menu bar</td>
</tr>
</tbody>
</table>
Some information in this manual may be out of date. Read all Release Notes files for the latest information.
# Table of Contents

**1 Palm OS Memory Management** .......................... 13
   Introduction to Memory Use on Palm OS ................. 14
   RAM and ROM Use .................................. 14
   PC Connectivity .................................. 14
   Memory Architecture .............................. 15
   Data Storage .................................. 16
   Accessing Data ................................ 16
      Memory Structure Overview .................. 17
      How Applications Access Data ............... 17
      Locating Storage Data With Local IDs .......... 18
   The Memory Manager ............................... 19
   Memory Hierarchy: RAM Store and ROM Store ........ 19
   Heap Overview .................................. 20
   Memory Manager Structures ...................... 20
      Heap Structures ........................... 21
      Chunk Structures ......................... 22
      Local ID Structures ....................... 23
   Using the Memory Manager ........................ 23
   Memory Manager Function Summary ............... 24
   The Data Manager ................................. 26
   Records and Databases ......................... 26
      Accessing Data with Local IDs .............. 27
      Using Presorted Lists ...................... 27
   Structure of a Database Header .................. 28
      Database Header Fields .................... 28
   Using the Data Manager ........................ 29
   Data Manager Function Summary ............... 30
   The Resource Manager ........................... 34
   Structure of a Resource Database Header ........ 35
   Using the Resource Manager .................... 36
   Resource Manager Functions .................... 37
## Table of Contents

### 2 Palm OS Communications

- Byte Ordering ........................................ 39
- Communications Architecture Hierarchy ........... 40
- The Serial Manager ................................ 42
  - Using the Serial Manager ....................... 42
  - Serial Manager Function Summary ............. 45
- The Serial Link Protocol .......................... 45
  - SLP Packet Structures ............................ 45
    - SLP Packet Format ............................... 45
    - Packet Type Assignment ....................... 47
    - Socket ID Assignment .......................... 47
    - Transaction ID Assignment .................... 47
  - Transmitting an SLP Packet ..................... 48
  - Receiving an SLP Packet ......................... 48
- The Serial Link Manager .......................... 49
  - Using the Serial Link Manager ................ 49
  - Serial Link Manager Function Summary ....... 53
- The Packet Assembly/Disassembly Protocol ......... 53
  - PADP Packet Structures ........................ 54
    - PADP Header .................................. 55
    - PADP padData Packet ............................ 55
    - PADP padAck Packet .............................. 57
    - PADP padTickle Packet .......................... 58
  - PADP Algorithms ................................ 58
    - Sending a Client Data Block .................. 59
    - Receiving a Client Data Block ................ 62
- The PAD Server ..................................... 65
  - Using the PAD Server ............................. 66
  - PAD Server Function Summary ................... 68

### 3 Memory Manager Functions

- MemCardInfo ........................................ 69
- MemChunkFree ...................................... 70
- MemDebugMode ..................................... 70
- MemHandleDataStorage ............................. 70
- MemHandleCardNo .................................. 71
<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>MemHandleFree</td>
<td>71</td>
</tr>
<tr>
<td>MemHandleHeapID</td>
<td>72</td>
</tr>
<tr>
<td>MemHandleLock</td>
<td>72</td>
</tr>
<tr>
<td>MemHandleNew</td>
<td>73</td>
</tr>
<tr>
<td>MemHandleResize</td>
<td>73</td>
</tr>
<tr>
<td>MemHandleSize</td>
<td>74</td>
</tr>
<tr>
<td>MemHandleToLocalID</td>
<td>74</td>
</tr>
<tr>
<td>MemHandleUnlock</td>
<td>75</td>
</tr>
<tr>
<td>MemHeapCheck</td>
<td>75</td>
</tr>
<tr>
<td>MemHeapCompact</td>
<td>76</td>
</tr>
<tr>
<td>MemHeapDynamic</td>
<td>76</td>
</tr>
<tr>
<td>MemHeapFlags</td>
<td>77</td>
</tr>
<tr>
<td>MemHeapFreeBytes</td>
<td>77</td>
</tr>
<tr>
<td>MemHeapID</td>
<td>78</td>
</tr>
<tr>
<td>MemHeapScramble</td>
<td>78</td>
</tr>
<tr>
<td>MemHeapSize</td>
<td>79</td>
</tr>
<tr>
<td>MemLocalIDKind</td>
<td>79</td>
</tr>
<tr>
<td>MemLocalIDToGlobal</td>
<td>80</td>
</tr>
<tr>
<td>MemLocalIDToLockedPtr</td>
<td>80</td>
</tr>
<tr>
<td>MemLocalIDToPtr</td>
<td>81</td>
</tr>
<tr>
<td>MemMove</td>
<td>81</td>
</tr>
<tr>
<td>MemNumCards</td>
<td>82</td>
</tr>
<tr>
<td>MemNumHeaps</td>
<td>82</td>
</tr>
<tr>
<td>MemNumRAMHeaps</td>
<td>82</td>
</tr>
<tr>
<td>MemPtrCardNo</td>
<td>83</td>
</tr>
<tr>
<td>MemPtrDataStorage</td>
<td>83</td>
</tr>
<tr>
<td>MemPtrFree</td>
<td>83</td>
</tr>
<tr>
<td>MemPtrHeapID</td>
<td>84</td>
</tr>
<tr>
<td>MemPtrToLocalID</td>
<td>84</td>
</tr>
<tr>
<td>MemPtrNew</td>
<td>84</td>
</tr>
<tr>
<td>MemPtrRecoverHandle</td>
<td>85</td>
</tr>
<tr>
<td>MemPtrResize</td>
<td>85</td>
</tr>
<tr>
<td>MemSet</td>
<td>86</td>
</tr>
<tr>
<td>MemSetDebugMode</td>
<td>87</td>
</tr>
<tr>
<td>MemPtrSize</td>
<td>87</td>
</tr>
<tr>
<td>MemPtrUnlock</td>
<td>88</td>
</tr>
</tbody>
</table>
# Table of Contents

<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>MemStoreInfo</td>
<td>89</td>
</tr>
<tr>
<td>Functions for System Use Only</td>
<td>90</td>
</tr>
<tr>
<td>MemCardFormat</td>
<td>90</td>
</tr>
<tr>
<td>MemChunkNew</td>
<td>90</td>
</tr>
<tr>
<td>MemHandleFlags</td>
<td>90</td>
</tr>
<tr>
<td>MemHandleLockCount</td>
<td>90</td>
</tr>
<tr>
<td>MemHandleOwner</td>
<td>91</td>
</tr>
<tr>
<td>MemHandleResetLock</td>
<td>91</td>
</tr>
<tr>
<td>MemHandleSetOwner</td>
<td>91</td>
</tr>
<tr>
<td>MemHeapFreeByOwnerID</td>
<td>91</td>
</tr>
<tr>
<td>MemHeapInit</td>
<td>91</td>
</tr>
<tr>
<td>MemInit</td>
<td>92</td>
</tr>
<tr>
<td>MemInitHeapTable</td>
<td>92</td>
</tr>
<tr>
<td>MemKernelInit</td>
<td>92</td>
</tr>
<tr>
<td>MemPtrFlags</td>
<td>92</td>
</tr>
<tr>
<td>MemPtrOwner</td>
<td>92</td>
</tr>
<tr>
<td>MemPtrResetLock</td>
<td>92</td>
</tr>
<tr>
<td>MemPtrSetOwner</td>
<td>93</td>
</tr>
<tr>
<td>MemSemaphoreRelease</td>
<td>93</td>
</tr>
<tr>
<td>MemSemaphoreReserve</td>
<td>93</td>
</tr>
<tr>
<td>MemStoreSetInfo</td>
<td>93</td>
</tr>
</tbody>
</table>

## 4 Data and Resource Manager Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>DmArchiveRecord</td>
<td>95</td>
</tr>
<tr>
<td>DmAttachRecord</td>
<td>96</td>
</tr>
<tr>
<td>DmAttachResource</td>
<td>97</td>
</tr>
<tr>
<td>DmCloseDatabase</td>
<td>98</td>
</tr>
<tr>
<td>DmCreateDatabase</td>
<td>98</td>
</tr>
<tr>
<td>DmCreateDatabaseFromImage</td>
<td>99</td>
</tr>
<tr>
<td>DmDatabaseInfo</td>
<td>100</td>
</tr>
<tr>
<td>DmDatabaseSize</td>
<td>101</td>
</tr>
<tr>
<td>DmDeleteDatabase</td>
<td>102</td>
</tr>
<tr>
<td>DmDeleteRecord</td>
<td>103</td>
</tr>
<tr>
<td>DmDetachRecord</td>
<td>104</td>
</tr>
<tr>
<td>DmDetachResource</td>
<td>105</td>
</tr>
<tr>
<td>DmFindDatabase</td>
<td>105</td>
</tr>
</tbody>
</table>
Table of Contents

DmFindRecordByID .............................................. 106
DmFindResource .................................................. 107
DmFindResourceType .............................................. 108
DmFindSortPosition .............................................. 109
DmGetAppInfoID .................................................... 110
DmGetDatabase ..................................................... 110
DmGetLastErr ....................................................... 111
DmGetNextDatabaseByTypeCreator ............................... 112
DmGetRecord ....................................................... 113
DmGetResource ...................................................... 114
DmGetResourceIndex .............................................. 114
DmGet1Resource ...................................................... 115
DmInsertionSort ..................................................... 116
DmMoveCategory ...................................................... 117
DmMoveRecord ....................................................... 118
DmNewHandle ........................................................ 118
DmNextOpenDatabase .............................................. 119
DmNextOpenResDatabase .......................................... 119
DmNewRecord ........................................................ 120
DmNewResource ....................................................... 121
DmNumDatabases ..................................................... 121
DmNumRecords ....................................................... 122
DmNumRecordsInCategory .......................................... 122
DmNumResources ..................................................... 122
DmOpenDatabase ..................................................... 123
DmOpenDatabaseByTypeCreator ................................... 124
DmOpenDatabaseInfo ............................................... 125
DmPositionInCategory ............................................. 126
DmQueryNextInCategory ........................................... 126
DmQueryRecord ....................................................... 127
DmQuickSort ........................................................ 128
DmRecordInfo ......................................................... 129
DmResourceInfo ....................................................... 130
DmReleaseRecord ..................................................... 131
DmReleaseResource .................................................. 131
DmRemoveRecord ...................................................... 132
Table of Contents

DmRemoveResource ........................................ 132
DmRemoveSecretRecords ................................. 133
DmResetRecordStates ................................. 133
DmResizeRecord ........................................ 134
DmResizeResource ........................................ 134
DmSearchRecord ........................................ 135
DmSearchResource ........................................ 136
DmSeekRecordInCategory ............................... 137
DmSet .................................................. 138
DmSetDatabaseInfo ..................................... 138
DmSetRecordInfo ........................................ 140
DmSetResourceInfo ..................................... 141
DmStrCopy ............................................ 142
DmWrite ............................................. 142
DmWriteCheck .......................................... 143
System Use Only ......................................... 143
  DmMoveOpenDBContext ................................ 143

5 Communications Functions .......................... 145
  Serial Manager ....................................... 145
  SerClearErr .......................................... 145
  SerClose ........................................... 146
  SerGetSettings ...................................... 146
  SerGetStatus ........................................ 147
  SerOpen ............................................ 148
  SerReceive ......................................... 149
  SerReceiveCheck .................................... 150
  SerReceiveFlush .................................... 150
  SerReceiveWait ...................................... 151
  SerSend ............................................. 152
  SerSendWait ........................................ 153
  SerSetReceiveBuffer ................................. 154
  SerSetSettings ....................................... 155
  Functions Used Only by System Software .......... 156
    SerSleep ......................................... 156
    SerWake .......................................... 156
Table of Contents

SerReceiveISP .................................................. 156
Serial Link Manager Functions ................................. 157
  SlkClose ................................................................ 157
  SlkCloseSocket .................................................... 158
  SlkFlushSocket .................................................... 158
  SlkOpen ............................................................. 159
  SlkOpenSocket ..................................................... 160
  SlkReceivePacket ................................................ 161
  SlkSendPacket ..................................................... 162
  SlkSetSocketListener ............................................ 163
  SlkSocketRefNum .................................................. 164
  SlkSocketSetTimeout ............................................. 164
  Functions for Use By System Software Only ............... 164
    SlkSysPktDefaultResponse .................................... 164
    SlkProcessRPC .................................................. 165
PAD Server Functions ............................................ 165
  PsrClose .......................................................... 165
  PsrGetCommand .................................................. 166
  PsrInit ........................................................... 167
  PsrSendReply ..................................................... 168
Miscellaneous Communications Functions .................... 169
  Crc16CalcBlock .................................................. 169
Palm OS Memory Management

This chapter helps you understand memory use on Palm OS. The chapter starts with an introduction to the memory layout and to the 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 includes a discussion of the structure of heaps, chunks, and records, the basic building blocks of Palm OS memory.

The second part of the chapter explains the different parts of the system—the managers—that you can use for memory management. Each discussion includes a brief overview of the relevant functions, with links to the related function descriptions.

- **The Memory Manager** maintains 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 conveniently retrieve and save chunks of data similar to the data manager, but with the added capability of tagging each chunk with a unique resource type and ID. These tagged data chunks, called resources, are stored in resource databases. Resources are typically used to store the application’s user interface elements (e.g. images, fonts, or dialog layouts.)
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, the OS is efficient in its use of both memory and processing resources. This section looks at two aspects of the device that contribute to this: RAM and ROM Use and PC Connectivity.

RAM and ROM Use

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 512K of ROM. The target battery life is 40 hours or more of “on” time from two AAA alkaline batteries.

The Palm OS device has its main suite of applications prebuilt into ROM. The preferred method for updating or enhancing the software is by replacing the ROM. Alternatively, additional or replacement applications and system extensions can be loaded into RAM, but given the limited amount of RAM this is not always practical. The ROM and RAM on each Palm OS device is on a memory module, permitting the user to completely replace the entire system software and applications suite by installing a single replacement module. There is no RAM or ROM storage on the motherboard of the device.

Because the Palm OS device permits easy wholesale replacement of the memory module, the design and operation of the system software does not have to be cast in stone. Each new ROM module for a Palm OS device can have different system software and applications on it. It is still advantageous however, to keep applications compatible at the source code level to minimize the engineering effort required to produce each new version of the ROM module.

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 re-
synchronizing with the PC. The format of the user’s data in the stor-
age 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

The Palm OS system software is designed around a 32-bit architec-
ture. All addresses are 32-bit and the basic data types are 8, 16, and
32 bits long. The Motorola 68328 processor’s registers are all 32 bits
wide, which allows a 32-bit execution model. The external data bus
is only 16 bits wide; this reduces cost without impacting the soft-
ware model. The processor’s bus controller automatically breaks
down 32-bit reads and writes into multiple 16-bit reads and writes
externally.

The 32-bit addresses available to software provide a total of 4 GB of
address space for storing code and data. This provides a large
growth potential for future revisions of both the hardware and soft-
ware without affecting the execution model (the first shipping ver-
sion has less than 1 MB of memory, or .025% of this address space).

Although a large memory space is available, Palm OS was designed
to work efficiently with small amounts of RAM. It uses a total of
only 32K of RAM for all working space: stacks, globals, temporary
memory allocations, etc. This leaves the remainder of RAM avail-
able for storing user data like appointments, to do lists, memos, ad-
dress lists, etc.

The Palm OS system software divides the total available RAM into
two virtual pieces: dynamic RAM and storage RAM. The dynamic
area of RAM is the 32K used for working space and is analogous to
the total amount of memory installed into a typical desktop system.
The remainder of the available RAM is designated as storage RAM
and is analogous to disk storage on a typical desktop system.

Since 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.” Even when the device is explicitly reset, all of memory is preserved, but the system software reinitializes the dynamic area only as part of the boot-up sequence.

Data Storage
Because the Palm OS device has a limited amount of dynamic memory available and uses nonvolatile RAM instead of disk storage, using a traditional file system is not the optimal method for storing and retrieving user data such as meetings or address book entries. Palm OS differs from traditional file systems as follows:

- Traditional file systems work by first reading all or a portion of a file into a memory buffer from disk, using or updating the information in the memory buffer, and then writing the updated memory buffer back to disk. Because of the high latency involved in reading or writing to disk, it is not practical to use small memory buffers and typically many kilobytes of data are read from or written to disk at a time.

- On the Palm OS device, it makes more sense to access and update data directly in place, because all nonvolatile information in the Palm OS device is stored in memory. This eliminates the extra overhead involved in a file system of transferring the data to and from another memory buffer and also reduces the dynamic memory requirements.

As a further enhancement, data in the Palm OS device is broken down into multiple finite size records, which can be left freely scattered throughout the memory space. Allowing records to be scattered throughout memory space means that the process of adding, deleting, or resizing a record does not require moving any other records around in memory.

Accessing Data
User data on the Palm OS device can be managed at the lowest level through the memory manager because:

- most chunks of data, like address book records, datebook records, etc., are relatively small (less than 256 bytes)
- all data is always resident in memory
This section first briefly discusses how data is organized, then explains the basic principles behind accessing data. More details, including a list of the API calls, are given in the sections on the different managers (The Memory Manager, The Data Manager, and The Resource Manager).

Memory Structure Overview

The Palm OS memory manager is designed to work best with small chunks of data; in fact, the first implementation enforces the constraint that all chunks be less than 64K each (even though the API does not have this constraint). To support this design, the memory in the Palm OS device is subdivided into multiple heaps of less than 64K each (see Heap Overview), which can each contain one or more chunks (see Chunk Structures). Because all heaps are less than 64K each, memory overhead for managing each heap is kept to a minimum since word (16-bit) offsets can be used to track each chunk in the heap. Finding and compacting free space is also faster with smaller heaps.

In the Palm OS environment all data are stored in memory manager chunks and each chunk resides in a heap. These data include dynamic data (such as global variables), nonvolatile storage data (analogous to files in disk-based systems), and any data or resources in ROM. Some heaps are ROM-based and contain only nonmovable chunks; some are RAM-based and may contain movable or nonmovable chunks. RAM-based heaps may either be dynamic heaps (for storing run-time variables) or storage heaps (for storage data).

Every memory chunk used to hold storage data (as opposed to memory chunks used to store dynamic data) is also referenced through a database. A database is analogous to a file in a traditional desktop system. In the Palm OS environment, a database is simply a list of all memory chunks that logically belong to a particular database. Every storage data chunk belongs to one and only one database. For every database, there is a database header chunk which contains a list of data chunks belonging to that database. See The Data Manager for more information.

How Applications Access Data

Applications reference most data chunks in the Palm OS device through handles to minimize fragmentation of heaps. A handle is a
reference to a master chunk pointer. 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 immediately “unlock” the handle to keep heap fragmentation to a minimum.

**Locating Storage Data With Local IDs**

Once a storage data record is located, an application can access it through its handle. A handle, however, is good only until the system is reset. Memory cards on the Palm OS device can be removed or inserted when power is off. When the system resets, it reinitializes all dynamic memory areas and relaunches applications. A handle to a storage chunk may not be the same after a reset if the user moved a memory card to a slot with a different base address. To work in this environment, all storage data on a memory card must be located through memory card–relative references, called **Local IDs**.

Note that the first version of the hardware has only one slot.

A Local ID is a card-relative reference to a data chunk and remains valid no matter what the base address of the card becomes. Once the base address of the card is determined at run time, a Local ID can be quickly converted to a real pointer or handle. A Local ID of a nonmovable chunk is simply the offset of the chunk from the base address of the card. A 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.

When an application needs the handle for a particular data record, it must use the data manager. The application tells the data manager which record to get (by index) out of which database. The data man-
ager 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.

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

- Memory Hierarchy: RAM Store and ROM Store
- Heap Overview
- Memory Manager Structures
- Using the Memory Manager
- Memory Manager Function Summary

Memory Hierarchy: RAM Store and ROM Store

The processor address space on the Palm OS device spans 4 GB since the 68328 has 32 internal address lines. 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.

Each memory card can contain ROM, RAM, or both. The ROM and RAM on each card is further divided into one or more heaps of 64K (in the current implementation) or less. 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 may be scattered throughout the memory space on the card.
Heap Overview

A heap is a 64K (or less) 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 in which heap it wishes to allocate a new chunk of memory. The memory manager manages each heap independently and rearranges chunks as necessary to defragment the heap and merge free space. Once a chunk is allocated in a specific heap, the memory manager never moves it out of that heap.

Heaps in the Palm OS environment are referenced through heap IDs. A heap ID is a 16-bit value that the memory manager uses to uniquely identify any heap in the entire address space. The heap IDs in card 0 start at 0 and increment sequentially first through the RAM heaps and then through the ROM heaps. The heap IDs in card 1 start at some value greater than 0 and also increment sequentially, first through all the RAM heaps and then through the ROM heaps.

The first heap(s) in card 0 is (are) dynamic heap(s), used for temporary memory allocations only, that is, non-file-related data, stack space, etc. Dynamic heaps are reinitialized every time the Palm OS device is reset. Every time an application quits, the system software frees any chunks in dynamic heaps that have been allocated by that application. All other heaps are nonvolatile and retain their contents through soft reset cycles. These nonvolatile heaps are used to store database directories, headers, and records.

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 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. 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.

- **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. Since each chunk header contains the size of the chunk, the heap can be easily walked by hopping from chunk to 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.

A chunk consists of a chunk header, a lock:owner byte and a Flags:size adjustment byte, and the hOffset word.

- **Chunk header.** At the start of the chunk is a 6-byte chunk header. The chunk header contains the size of the chunk which is larger than the size requested by the application and includes the size of the header itself. Since an entire heap must be 64K or less, the maximum data size for a chunk is 64K, minus the size of the heap header and master pointer table, minus 6 bytes for the chunk header.

- **Lock:owner byte.** Following the size field is a byte which holds the lock count in the high nibble and the owner ID in the low nibble. The owner ID determines the owner of a memory chunk and is set by the memory manager when allocating a new chunk. The owner ID is useful information for debugging and for garbage collection when an application terminates abnormally. 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.

- **Flags:size adjustment byte.** Following the lock:owner byte is a byte which 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.

- **hOffset word.** The last word 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 memory manager call MemLocalIDToGlobal takes a Local ID and a card number (either 0 or 1) and converts the Local ID 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

Usually, applications use the memory manager to allocate memory only in the dynamic heap(s). The data manager provides an API for allocating memory in the storage heaps used to hold user data. The data manager calls the memory manager as appropriate to do its low-level allocations.

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. MemHandleUnlock unlocks a chunk.

To determine the size of a movable chunk, pass its handle to MemHandleSize. To resize it, call MemHandleResize. You gener-
ally 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 `MemPtrXXX` 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 conveniently move memory from one place to another or to fill memory with a specific value.

When an application allocates memory in the dynamic heap(s), the memory manager gives it an owner ID that associates that chunk with the application. When the application quits, all chunks in the dynamic heap that have its owner ID are disposed of automatically. If the system needs to allocate a chunk that is not disposed of when an application quits, it has to change the owner ID to 0 by calling the system function `MemHandleSetOwner`.

**Memory Manager Function Summary**

- `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
The Data Manager

The Palm OS device has only a limited amount of dynamic memory available and uses nonvolatile RAM instead of disk storage. Using a traditional file system is therefore not the optimal method for storing and retrieving user data such as meetings, address book entries, and so on. 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 all nonvolatile information in the Palm OS device is stored in memory, it makes sense to access and update the data directly in place. This eliminates the overhead of transferring the data to and from another memory buffer involved in a file system. It also reduces the dynamic memory requirements.

As a further enhancement, data in the Palm OS device is broken down into multiple, finite-size records which can be left freely scattered throughout the memory space. Allowing records to be scattered throughout memory space means that adding, deleting, or resizing a record does not require moving any other records around in memory.

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, or all datebook entries, and so on. An application on
Palm OS 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 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.

This database method of storing data fits in nicely with the design of the Palm OS memory manager. Each record in a database is in fact a memory manager chunk. The data manager uses memory manager calls to allocate, delete, and resize database records. All heaps except for the dynamic heap(s) are nonvolatile, so database records can be stored in any heap except for the dynamic heap(s) (see “Heap Overview” on page 20). Because the records can be stored anywhere on the memory card, databases can even be distributed over multiple contiguous 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 of the use of Local IDs, it is possible to place the memory card 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.

**Using Presorted Lists**

One side benefit of the Palm OS database method of storing records by index is that it becomes fairly cheap to maintain one or more presorted versions of the database record list. A sorted list for a database can simply be a list of record indices, presorted in the correct manner. For example, the address book database can be presorted by last name, company, or city, just by maintaining three separate sort lists. Since each sort list entry is only a 16-bit record index, this is a relatively small data array. Having precalculated sort lists available allows different sorted views of the address book to be displayed quickly.
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 Database Header Fields and 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; this allows applications to 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 that can be used by an application 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. These fields are used by the system 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 1” 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.”

**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 are used so that the database is slot-independent. Since all the 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 deleted bit is set in the attributes flags, but its entry in the database header is kept around 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. By using this scheme, the next time synchronization is performed with the desktop system, the desktop can quickly determine which records the user wants to delete (since their record entries are still around on the Palm OS device). In the case of archived records, it 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.

![Record Attributes Diagram](image)

**Figure 1.1 Record Attributes**

**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 when 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 which is stored in the lower-4 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**

- `DmQuickSort`
- `DmFindSortPosition`
- `DmInsertionSort`
- `DmCreateDatabaseFromImage`
- DmGetNextDatabaseByTypeCreator
- DmCreateDatabase
- DmDeleteDatabase
- DmNumDatabases
- DmGetDatabase
- DmFindDatabase
- DmOpenDatabaseByTypeCreator
- DmCloseDatabase
- DmGetAppInfoID
- DmDatabaseInfo
- DmSetDatabaseInfo
- DmDatabaseSize
- DmOpenDatabase
- DmCloseDatabase
- DmNextOpenDatabase
- DmOpenDatabaseInfo
- DmResetRecordStates
- DmGetLastErr
- DmNumRecords
- DmRecordInfo
- DmSetRecordInfo
- DmAttachRecord
- DmDetachRecord
- DmMoveRecord
- DmNewRecord
- DmRemoveRecord
- DmDeleteRecord
- DmArchiveRecord
- DmNewHandle
- DmRemoveSecretRecords
- DmFindRecordByID
- DmSearchRecord
- DmQueryRecord
The Resource Manager

Applications can use the Resource Manager much like the data manager to conveniently retrieve and save chunks of data. It 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 (2 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, etc. 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, etc. These preferences settings can be stored in a separate resource database.

This section explains how to work with the resource manager by discussing 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. This allows applications to 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 resize it.
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:

- DmGetResource
- DmGet1Resource
- DmReleaseResource
- DmResizeResource
Palm OS Memory Management
The Resource Manager

- DmNextOpenResDatabase
- DmFindResourceType
- DmFindResource
- DmSearchResource
- DmNumResources
- DmResourceInfo
- DmSetResourceInfo
- DmAttachResource
- DmDetachResource
- DmNewResource
- DmRemoveResource
- DmGetResourceIndex
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 packet send and receive mechanism.
- **The Serial Link Manager** is the Palm OS implementation of the serial link protocol.
- **The Packet Assembly/Disassembly Protocol (PADP)**.
- **The PAD Server** is the Palm OS implementation of the PADP.

### Byte Ordering

By convention, all data originating from and destined for the Palm OS device uses Motorola byte ordering. That is, data of compound types such as Word (2 bytes) and DWord (4 bytes), as well as their integral counterparts, is 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, with higher layers depending on more primitive functionality provided by lower layers. Functionality of all layers is available to applications. The software consists of these 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.
- The modem manager provides modem dialing capabilities.
- The Serial Link Protocol (SLP) provides best-effort packet send and receive capabilities with CRC-16. SLP does not guarantee packet delivery; this is left to the higher-level protocols.
- 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.
- 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 sub-systems. DLP facilitates efficient data synchronization between desktop (i.e., PC, Macintosh, etc.) and Palm OS applications, database backup, installation of code patches, extensions, applications, and other databases, as well as Remote Inter-Application Communication (RIAC) and Remote Procedure Calls (RPC).
Figure 2.1  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 sends data using the polling model.

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. 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 is provided for use by cooperating tasks which need 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 time-out, number of data and stop bits, parity options, and handshaking op-
tions, call `SerSetSettings`. For baud rates above 19200, use of hardware handshaking is advised.

To retrieve the current serial port settings, call `SerGetSettings`. 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. It is therefore important to 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 is transferred to the UART or a time-out error (if CTS handshaking is enabled) occurs. If your software needs to detect when all data has been transmitted, see `SerSendWait`.

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 time-out 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 has 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 time out. 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, it is important to acknowledge the detection of line errors by calling `SerClearErr`. 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 inter-byte time-out. 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.”
Serial Manager Function Summary

• SerClearErr
• SerClose
• 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 (this 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, and Transaction ID Assignment.

SLP Packet Format

Each SLP packet consists of a packet header, client data of variable size, and a packet footer.

• 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 signa-
ture 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.

![Figure 2.2 Structure of a Serial Link Packet](image-url)
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 which are reserved by the components of the system software. The dynamic socket IDs are assigned at run time 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:
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.

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.
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, it must open it 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 `SerOpen`), a pointer to a memory location for returning the socket ID, and a Boolean indicating whether the socket is static or dynamic. If opening a static socket, 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, you must 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 time out.

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:

- the packet header buffer (size of `SlkPktHeaderType`)
- the packet body buffer, which 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. The serial link manager does not free the `SlkSocketListenType` structure or the buffers when the socket is closed; that 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 2.1 Sending a Serial Link Packet**

```c
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 2.2 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 which does not have a socket listener. The parameters also specify buffers for the packet header and client data, and a time out. The time out indicates how long the receiver should wait for a packet to begin arriving before timing out. A time-out value of (-1) means “wait forever.” If a packet is received for a socket with a registered socket listener, it is dispatched via its socket listener procedure.
Serial Link Manager Function Summary

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

The Packet Assembly/Disassembly Protocol

The Packet Assembly/Disassembly Protocol (PADP) provides the infrastructure for sending variable-size commands and receiving variable-size responses. As is common for transport layer protocols, PADP is asymmetric in the sense that only one side of the connection can issue commands, while the other side can only send responses. For convenience, this document uses the term workstation to refer to the side of the connection which sends commands. The side of the connection which sends responses is referred to as the server. A single command-response cycle is a transaction.

PADP provides reliable buffered data transfer capabilities. It is a simple and efficient half-duplex protocol featuring variable-size block transfers with robust error checking and automatic retries. The packet assembly/disassembly technique is used to break up a large block of client data into multiple data packets, thus improving error recovery performance over possibly noisy connections such as telephone lines. Up to 65,535 bytes of client data can be transferred in each direction within a single PADP transaction.

PADP builds on top of the Serial Link Protocol (SLP) by building its own packet structure into the client data section of the SLP packet.

The following sections describe the PADP packets and their formats, and the PADP algorithms for sending and receiving client data.
PADP Packet Structures

PADP employs three types of packets: padData, padAck, and padTickle.

- A **PADP padData Packet** transfers client data.
- A **PADP padAck Packet** acknowledges the receipt of valid padData packets.
- A **PADP padTickle Packet** keeps the session “alive” while the workstation is performing a time-consuming activity between commands.

PADP packets are embedded within the client data section of SLP packets. SLP reserves SLP packet type 0x02 for PADP packets. (see **PADP padTickle Packet** below)
The following sections describe the formats of the PADP structures embedded within the SLP client data. For a detailed description of SLP packet structure refer to The Serial Link Protocol.

**PADP Header**

All PADP packets contain the PADP header. The PADP header contains the PADP packet type field, a flags field, and a sizeOrOffset field. The type field identifies the PADP packet as one of the following three PADP packet types:

- 0x01 = padData
- 0x02 = padAck
- 0x04 = padTickle

The usage of the individual fields within each type of PADP packet is described in detail in the following sections. presents the PADP header fields, with the field size (in bytes) indicated in parentheses.

![Figure 2.4 PADP Packet Header](image)

**PADP padData Packet**

The padData packets are used to transfer client data. A padData packet consists of the fixed-size PADP header followed by a variable-size section of PADP client data. A single padData packet may contain at most 1024 bytes of PADP client data.

The flags field in the PADP header of a padData packet is used to identify first and last padData packets within the block of client data being transferred. When the entire block of client data fits within a single padData packet, the packet is marked as both first and last. All unused bits must be set to zero.

Usage of the sizeOrOffset field in the PADP header of a padData packet depends on whether this padData packet is the first packet within the block of client data being transferred.
• If this is the first padData packet of the block (it will be marked as “first” in the PADP header flags field), the sizeOrOffset field contains the total size of the client data block being transferred. This provides the receiver with the necessary information to determine whether it can accommodate a block of this size, as well as the opportunity to allocate a memory buffer for the entire client data block being received.

• If the padData packet is not marked as first in the PADP header flags field, the sizeOrOffset fields holds the relative zero-based offset of the client data contained in the packet from the beginning of the entire client data block being transferred.

Figure 2.5 presents the padData packet.
PADP padAck Packet

The padAck packets are used to acknowledge valid padData packets. A padAck packet consists of the fixed-size PADP header only.

The “first” and “last” packet bits of the flags field in the PADP header of a padAck packet match those of the padData packet being acknowledged. The memory error bit is for signaling to the data sender that the receiver cannot accommodate the incoming data block whose size is indicated in the first padData packet. When the data sender receives a padAck packet with the memory error bit set in response to the first padData packet, it must abort sending the data block immediately, returning an error code to the caller. All unused bits must be set to zero.

The value of the sizeOrOffset field in the PADP header of a padAck packet matches that of the padData packet being acknowledged.

Figure 2.6 presents the padAck packet.

![Figure 2.6 PADP padAck Packet Format](image-url)
PADP padTickle Packet

The padTickle packets are used for keeping the session alive while the workstation is performing a time-consuming activity between transactions.

The flags and sizeOrOffset fields in the PADP header of a padTickle packet are set to zero.

Figure 2.7 presents the padTickle packet.

![Figure 2.7 PADP padTickle Packet Format](image)

PADP Algorithms

The model employed by PADP consists of two entities: the workstation and the server.

- The workstation issues commands and receives responses.
- The server receives commands and sends responses. The server entity is not allowed to initiate commands.

A single command and its matching response constitute one transaction.

To keep the session alive between transactions, the workstation entity sends padTickle packets to the server entity at 7-second inter-
In the future, the protocol may be extended to have the server entity also send padTickle packets to the workstation entity.

A maximum of 65535 bytes of client data may be sent in a single PADP command or response. The client data block is logically divided into segments of 1024 bytes; the last segment may contain less than 1024 bytes. Each segment is then sent in a padData packet, with retries if necessary. Since the protocol is half-duplex, each padData packet must be acknowledged by the receiver before the next segment can be sent. Each padData packet is resent at fixed intervals until it is acknowledged or the maximum retry count (discussed later) is exceeded. Refer to PADP Packet Structures for packet format details.

All padData and padAck packets within a single transaction are identified by the same transaction ID value. Subsequent transactions increment through the transaction ID values, wrapping around eventually. The workstation entity issuing the command generates the transaction ID. The server entity uses that transaction ID value in the corresponding response. While waiting for a new command, the server entity filters out any PADP packets which have the transaction ID of the last successfully received command. Refer to The Serial Link Protocol and The Serial Link Manager for information about reserved transaction ID values.

After sending a packet, the implementations needs to wait for the transmit queue to empty before starting the time-out counter to receive the next expected packet. Only then the protocol timing schemes will work correctly and will be independent of the baud rate and packet size,

**Sending a Client Data Block**

This section presents the algorithm for sending a block of client data (i.e., a command to the server or response to the workstation). Note that

- For the workstation implementation, `retryInterval` is currently 4 seconds and `maxRetries` is 14 seconds.
- For the server implementation on Palm OS, `retryInterval` is 2 seconds and `maxRetries` is 10 seconds.

The values of `retryInterval` and `maxRetries` are greater for the workstation implementation to allow for heap compaction on the
device. On rare occasions, compaction may take as long as 20 seconds per storage heap (when receiving a large data block, the Palm OS receiver attempts to allocate the buffer space from one of the storage heaps before acknowledging the first padData packet from the sender, and this could require heap compaction).

**Listing 2.3 Sending a block of data**

```c
// Algorithm for sending a block of data

initialize reference to the first client data segment to be sent;
while (there are more segments to send)
{
    // generate the correct PADP packet header flags and sizeOrOffset values for the current segment;
    // Retry loop
    for ( up to maxRetries )
    {
        send a padData packet containing the current client data segment;
        wait for retryInterval seconds to receive a matching padAck packet;
        if ( matching padAck packet received )
        {
            if ( the "memory error" bit is set in the padAck header )
                abort transmission of this client data block;
            else
                break out of the retry loop;
        }
    }

    if ( (we were sending an intermediate
```
There is a special case which arises and must be addressed in the implementation to ensure error recovery under adverse line conditions.

Consider the case of a lost or damaged padAck packet. If an intermediate (other than last) padData packet of the data block is sent, and the matching padAck is lost, the receiver, who is still waiting for subsequent padData packets, will acknowledge retries, ensuring recovery.

The situation is different if the last padData packet of the block is sent and the matching padAck is lost. In this case, the receiver, having received and acknowledged the last padData packet of the block, ceases to wait and returns the received block to its client for processing. In the meantime, the sender, who never received that ill-fated padAck, is in its retry loop resending the last padData packet and awaiting the matching padAck.

In this situation the entire block of data was successfully received but the sender doesn’t know this because of one lost padAck. Because a padAck is as likely to be lost on a noisy line as any other packet, a recovery technique must be introduced. The solution, which differs slightly between the workstation and server implementations, is discussed next.

When the workstation is sending a client data block, it’s sending a command for which it expects a response from the server. When the client of the server entity finishes processing the command, it initiates a response by sending the response data block.
The padData packets of the response carry the same transaction ID as the padData packets of the command. If the workstation is still in its retry loop waiting for a matching padAck to the last padData packet of the block, but instead receives a “first” padData packet with a matching transaction ID from the server, the workstation entity can recover by treating the received padData packet as the equivalent of the expected padAck packet.

It is also possible that the workstation entity exhausts all the retries of the last padData without receiving the first padData packet of the response block due to time-consuming processing of the command. In this case, the workstation entity can assume that the last padData packet of the block was delivered successfully and leave it to the workstation receiver to detect a lost connection if it times out while waiting to receive the response.

When the server entity is sending a client data block, it is sending a response to the command it received from the workstation entity. After the client of the workstation entity receives the response, it eventually sends a new command (unless that was its last command). The new command uses a different transaction ID. Therefore, if the server entity is still in its retry loop waiting for a matching padAck to the last padData packet of the block, but instead receives a “first” padData packet with a different transaction ID from the workstation entity, the server entity can recover by treating the received padData packet as the equivalent of the expected padAck packet.

It is also possible that the server entity exhausts all the retries of the last padData without receiving the first padData packet of a new command block due to time-consuming processing on the workstation end. In this case, the server entity can make the assumption that the last padData packet of the block was delivered successfully, leaving it to the server receiver to detect a lost connection if it times out while waiting to receive the next command.

**Receiving a Client Data Block**

This section presents the algorithm for receiving a block of client data. Please note that for the workstation implementation, the term “expected transaction ID” means the same transaction ID as that used for the matching command. For the server implementation, the term “expected transaction ID” means a transaction ID value which
is different from that of the last successfully received command. The receiver must filter out any packet which does not have the expected transaction ID. For the workstation implementation, blockReceiveTimeout and segmentReceiveTimeout are 45 seconds each. For the server implementation on the Palm OS device, blockReceiveTimeout and segmentReceiveTimeout are 30 seconds each.

Listing 2.4  Receiving a Block of Data

initialize expected offset to zero;

// Receive the first data segment
reset the timeout counter;
while ( elapsed time is less than blockReceiveTimeout )
{
    attempt to receive the first padData packet
    with the expected transaction id.
    if ( succeeded )
    {
        if ( there is enough storage to receive the
            entire data block )
        {
            // The implementation may choose to use a
            // preallocated buffer or allocate a new
            // buffer for the incoming block.
            save the first data segment in our buffer;
            increment the expected offset by the size
            of the data segment;
            acknowledge this padData packet with a
            matching padAck;
            break out of this loop and go on to receive
            remaining segments;
        }
        else
        {
            send a padAck packet with the "memory
error" flag set;
return to caller with appropriate error
code;
}
}
else
if ( received a padTickle packet )
{
reset the timeout counter, continue waiting;
}
}

if ( we timed out without receiving the first
data segment )
{
// The connection is presumed lost
return to caller with appropriate error code;
}

// Receive the remaining data segments
while ( there are more segments to receive )
{
// Wait for the next data segment
reset the timeout counter;
while ( elapsed time is less than
segmentReceiveTimeout )
{
attempt to receive a padData packet with the
expected transaction id.
if ( succeeded )
{
if ( the padData packet has the expected
offset )
{
save the data segment in our buffer;
increment the expected offset by the size
of the data segment;
acknowledge this padData packet with a
matching padAck;
break out of the inner loop;
}
else
{
  // This is a retry of an already received
  padData packet
  acknowledge this padData packet with a
  matching padAck;
  reset the timeout counter;
  continue waiting for expected data
  segment;
  }
}

if ( we timed out without receiving the
expected data segment )
{
  // The connection is presumed lost
  return to caller with appropriate error code;
}

The PAD Server

The PAD Server is the Palm OS implementation of the Palm OS
PADP Server entity.

The PAD Server provides the mechanisms for receiving PADP com-
mands and sending PADP responses via synchronous function calls.

PAD Server provides an API for receiving PADP commands from
the PADP workstation entity, and for sending PADP responses. The
present implementation of PAD Server supports only one client ses-
sion at a time. Higher-level services are built on top of those pro-
vided by PAD Server. For example, the connection manager and
Desktop Link Server (discussed later) both use PAD Server for reliable data transfer. The services of PAD Server are available to any application which needs to incorporate a reliable data transport layer.

See The Packet Assembly/Disassembly Protocol for a detailed discussion of PADP concepts.

Using the PAD Server

Before an application can use the services of the PAD Server, it has to open and initialize a serial port (see The Serial Manager), open the serial link manager and open a Serial Link socket (see The Serial Link Manager).

The next step is to call PsrInit to open and initialize the PAD Server. An error code of 0 (zero) indicates success. Other error codes indicate failure. In the call to PsrInit you can specify a pointer to a Cancel Callback procedure. If specified, the Cancel Callback is called periodically while waiting for a command or sending a response. If the Cancel Callback returns non-zero, the wait aborts immediately, permitting fast response in situations such as cancelling by the user.

When you finish using the PAD Server, you have to call PsrClose. PsrClose may be called only if PsrInit returned 0 (zero). PsrClose frees the resources allocated by PsrInit.

To receive a PADP command, call PsrGetCommand. On success, PsrGetCommand returns the command block, the remote socket ID, and the transaction ID of the command.

To send a PADP response, call PsrSendReply, passing the remote socket ID, transaction ID, an array of PmSegmentType structures and the number of elements in the array. For convenience, the response block is specified as a list of data segments via an array of PmSegmentType structures. The PmSegmentType structure allows selective specification of word alignment for each data segment. If word alignment is enabled for a segment and the previous segment’s data size forces it to begin at an odd offset, PsrSendReply automatically inserts a byte to force word alignment of the segment’s data. Any bytes inserted as the result of word alignment are set to 0 (zero) in the resulting response block.
Listing 2.5  Sending a PADP Response

//
//Using PsrSendReply to send a PADP response.
//

Err SendPADPResponseExample(Byte remoteSocketID, Byte transactionID)
{
    Err err;
    PmSegmentType seg[3];
    Byte dataSegment0[53];
    Byte dataSegment1[10];
    Byte dataSegment2[15];

    seg[0].dataP = dataSegment0;
    seg[0].dataSize = sizeof(dataSegment0);
    seg[0].wordAlign = false;

    seg[1].dataP = dataSegment1;
    seg[1].dataSize = sizeof(dataSegment1);
    seg[1].wordAlign = true;

    seg[2].dataP = dataSegment2;
    seg[2].dataSize = sizeof(dataSegment2);
    seg[2].wordAlign = false;

    err = PsrSendReply( remoteSocketID, transactionID, seg, 3/*segCount*/);

    return( err );
}
PAD Server Function Summary

- PsrClose
- PsrGetCommand
- PsrInit
- PsrSendReply
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.
Memory Manager Functions

**MemChunkFree**

**Purpose**
Dispose of a chunk.

**Prototype**
Err MemChunkFree (VoidPtr chunkDataP)

**Parameters**
chunkDataP Chunk data pointer.

**Result**
0 No error
memErrInvalidParam Invalid parameter

**Comments**
Call this routine to dispose of a chunk, which is disposed of even if it’s locked.

**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.

**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
**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 which card number (0 or 1) a movable chunk resides on.

**See Also**
MemPtrCardNo

---

**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
Memory Manager Functions

**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.

**Prototype**
VoidHand MemHandleNew (ULong size)

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

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

**Comments**
Allocates a movable chunk in the dynamic heap and returns a handle it. Use this call when allocating dynamic memory.

**See Also**
MemPtrFree, MemPtrNew, MemHandleFree

**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.

**See Also**
MemHandleNew, MemHandleSize
Memory Manager Functions

**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**
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.

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.

**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.
<P> <-> freeP Pointer to a variable of type ULong for free bytes.
<P> <-> maxP Pointer to a variable of type ULong for max free chunk size.

Result Always returns 0.

Comments Call this routine to retrieve the total number of free bytes left in a heap and the size of the largest free chunk. This routine doesn’t compact the heap but the caller may compact the heap explicitly before calling this routine to determine if an allocation will succeed or not.

See Also MemHeapSize, MemHeapID, MemHeapCompact
**Memory Manager Functions**

---

**MemHeapID**

**Purpose**
Return the heapID 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 given heap.

**Prototype**
`Err MemHeapScramble (UInt heapID)`

**Parameters**
- `heapID` ID of heap to scramble.

**Comments**
The system does multiple passes over the heap attempting to move each movable chunk.

Useful during 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  The 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.

**Comments**  
This routine converts a Local ID back to a pointer or handle, given the card number that the chunk resides in.

**See Also**  
MemLocalIDKind, MemLocalIDToLockedPtr

**MemLocalIDToLockedPtr**

**Purpose**  
Return a pointer to a chunk designated by 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 chunkID.
- `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 zero to indicate an error.

See Also
MemLocalIDToGlobal, MemLocalIDToLockedPtr

MemMove

Purpose
Move a range of memory to another range in the dynamic heap.

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.
Memory Manager 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 heapID. The heapID 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
Memory Manager Functions

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
Returns 0 if no error or memErrInvalidParam (Invalid parameter).

Comments
Call this routine to dispose of a nonmovable chunk.
Memory Manager Functions

**MemPtrHeapID**

**Purpose**
Return the heap ID of a chunk.

**Prototype**
UInt MemPtrHeapID (VoidPtr p)

**Parameters**
-> chunkP 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.

**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

**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.
### Memory Manager Functions

#### MemPtrRecoverHandle

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

**Prototype**

```c
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 non-movable data chunks.

#### MemPtrResize

**Purpose**
Resize a chunk.

**Prototype**

```c
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`
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**
Provide one (or none) of the following flags:
- memDebugModeCheckOnChange
- memDebugModeCheckOnAll
- memDebugModeScrambleOnChange
- memDebugModeScrambleOnAll
- memDebugModeFillFree
- memDebugModeAllHeaps
- memDebugModeAllHeaps
- memDebugModeRecordMinDynHeapFree

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

**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.
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
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.
Memory Manager Functions

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.

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.
Memory Manager Functions

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.
Memory Manager Functions

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)`
Memory Manager Functions

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)

Warning: 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)
Memory Manager Functions
Data and Resource 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 param-
          eter atP points to an index variable. If oldHP is nil, the new record
          is inserted at index *atP and all following record indices are
          shifted down. If *atP is greater than the number of records cur-
          rentsly in the database, the new record is appended to the end and
          the index of it 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
Data and Resource Manager Functions

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.
Data and Resource Manager Functions

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. This routine doesn’t check for a database with the same name, so check for it yourself. 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
Data and Resource Manager Functions

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` Which card number 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.
Data and Resource Manager Functions

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

DmDatabaseSize

Purpose
Retrieve size information on a database.

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

Parameters
- cardNo Which 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
**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 data-
base 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**  
```c
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
Data and Resource Manager Functions

**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**
```c
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 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 DmFindSortPosition ( 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
compar, the comparison function, 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
DmQuickSort, DmInsertionSort
Data and Resource Manager Functions

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 Which card number.
- 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,
UIPtr 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 cardNo 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 invocation.

If the type parameter is nil, this routine can be called successively to return all databases of the given creator. If the creator param-
eter is nil, this routine can be called successively to return all data-
bases of the given type.

If the onlyLatestVers parameter is set, only the latest version of
each database with a given creator/type pair is returned.

If you’re searching for the latest version and either type or
creator is nil (wildcard), this routine returns the index of the next
database which matches the search criteria. This database can’t
have been superseded by a newer version of that database with the
same type and creator.

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  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 struc-
ture.

DmReleaseRecord should be called as soon as the caller is done
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’s done 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’s done 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.

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 amount 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.

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

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

Result
Returns 0 if no error or dmErrInvalidParam.
Data and Resource Manager Functions

Comments
Called by SysAppLaunch (see Part 1) to move an application database is launching out of the system list and into the application’s list.

See Also
DmFindSortPosition, 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.
Data and Resource Manager Functions

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. Note
that 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, tries 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 obtain and save its LocalID in the appInfoID or sortInfoID
fields of the header.
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. This routine would not normally be called by applications 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.
Data and Resource Manager Functions

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 Allocate 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 following record indices 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: Void 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

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
Data and Resource Manager Functions

**DmNumRecords**

**Purpose**
Return the number of records in a database.

**Prototype**
UInt DmNumRecords (DmOpenRef dbR)

**Parameters**
- dbR
  
**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.

**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
  
**Result**
Returns the total number of resources in the given database.
## DmOpenDatabase

**Purpose**
Open a database and return a reference to it.

**Prototype**
```c
DmOpenRef DmOpenDatabase ( UInt cardNo,
                                 LocalID dbID,
                                 UInt mode)
```

**Parameters**
- `-> cardNo` Which 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 it.

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**

```c
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
Data and Resource Manager Functions

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
Data and Resource Manager Functions

**DmPositionInCategory**

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

**Prototype**
```c
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**
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**
```c
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.
Data and Resource Manager Functions

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.

compar, the comparison function, 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
DmFindSortPosition, DmInsertionSort
**DmRecordInfo**

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

**Prototype**

```c
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
Data and Resource Manager Functions

DmResourceInfo

**Purpose**
Retrieve information on a given resource.

**Prototype**
```c
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]
Data and Resource Manager Functions

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 Which 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 finished modifying or reading a record that you’ve called DmGetRecord on. It sets the dirty bit for the record if the dirty parameter is set.

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
Data and Resource Manager Functions

**DmRemoveRecord**

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

**Prototype**
```c
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 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**
```c
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
Data and Resource Manager Functions

**DmRemoveSecretRecords**

**Purpose**
Remove all secret records.

**Prototype**
```c
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**
```c
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.
Data and Resource Manager Functions

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 return 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.
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
Data and Resource Manager Functions

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**
```c
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 or dmErrIndexOutOfRange or dmErrSeekFailed if an error occurred.

**See Also**
DmNumRecordsInCategory, DmQueryNextInCategory, DmPositionInCategory, DmMoveCategory
**Data and Resource Manager Functions**

---

**DmSet**

**Purpose**
Check the validity of the chunk pointer for a record and makes sure that writing the record does not exceed the chunk bounds.

**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 or `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 chunkID (if any) is marked as an orphan chunk and the new chunk ID is un orphaned. 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 or `dmErrIndexOutOfRange` or `dmErrReadOnly` if an error occurred.

**Comments**  
Set information about a record.

**See Also**  
[DmNumRecords](#), [DmRecordInfo](#)
Data and Resource Manager Functions

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 or 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.
Data and Resource Manager Functions

**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 or 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 or dmErrNotValidRecord or dmErrWriteOutOfBounds if an error occurred.

**See Also**
DmSet
Data and Resource Manager Functions

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 or dmErrNotValidRecord or
dmErrWriteOutOfBounds if an error occurred.

System Use Only

DmMoveOpenDBContext

Prototype
Err DmMoveOpenDBContext (DmOpenRef* dstHeadP,
DmOpenRef dbR)

Warning: System Use Only!
Communications Functions

Serial Manager

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.

Comments  
Other serial manager functions, such as SerReceive, immediately return with the error code serErrLineErr if any line errors are pending. It is therefore important to check the result of serial manager function calls and call SerClearErr in acknowledgment if line error(s) occur.
Communications Functions
Serial Manager

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  The port wasn’t open.
       serErrStillOpen  The port is still held open by someone else.

Comments  Releases the serial port and shuts down serial port hardware if the open count has reached 0. SerClose may be called only if the return value from SerOpen was 0 (zero) or serErrAlreadyOpen. 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.

See Also  SerOpen

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, data format options. See the definition of the SerSettingsType structure for more details.

See Also  SerSend
**SerGetStatus**

**Purpose**
Return the pending line error status for errors which have been detected since the last time SerClearErr was called.

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

**Parameters**
- `refNum` The serial library reference number.
- `ctsOnP` Pointer to location for storing a Boolean value.
- `dsrOnP` Pointer to location for storing a Boolean value.

**Result**
Any combination of the following constants bitwise or’ed together:
- `serLineErrorParity` Parity error.
- `serLineErrorHWOverrun` Hardware overrun.
- `serLineErrorFraming` Framing error.
- `serLineErrorBreak` Break signal detected.
- `serLineErrorHShake` Line hand-shake 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 hand-shaking 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 is provided for use by cooperating tasks which need to share the serial port. Other tasks must refrain from using the port if serErrAlreadyOpen is returned and close it by calling SerClose.

See Also
SerClose
SerReceive

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.
Communications Functions
Serial Manager

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 time out occurs while waiting for the next byte to arrive.
SerReceiveWait

Purpose Wait for at least $\text{bytes}$ 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 time out in system ticks (-1 = forever).

Result
0 No error.
serErrTimeOut Interbyte time out 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`
**Communications Functions**

*Serial Manager*

---

**SerSend**

**Purpose**
Send a stream of bytes to the serial port.

**Prototype**
```
Err SerSend (UInt refNum, VoidPtr bufP, ULong size)
```

**Parameters**
- `-> refNum` The 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 time out (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 time out 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  The serial library reference number.
-> timeout  Reserved for future enhancements.
            Set to (-1) for compatibility.

Result  0  No error.
        serErrTimeOut  Handshake time out (such as waiting for CTS to become asserted).

Comments  SerSendWait blocks until all data is transferred or a time-out error (if CTS handshaking is enabled) occurs. This routine observes the current CTS timeout setting if CTS handshaking is enabled (see SerGetSettings and SerSend).
Communications Functions
Serial Manager

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**  

```c
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 time out, handshaking options, and data format options. See the definition of the `SerSettingsType` structure for more details.

**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.

SerSleep

Prototype: `Err SerSleep (UInt refNum)`

WARNING: This function for use by system software only.

SerWake

Prototype: `Err SerWake (UInt refNum)`

WARNING: This function for use by system software only.

SerReceiveISP

Prototype: `Boolean SerReceiveISP (void)`

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.
Communications Functions
Serial Link Manager Functions

**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**  
```
Err SlkFlushSocket (UInt socket, Long timeout)
```

**Parameters**
- **socket**  Socket ID.
- **timeout**  Interbyte time out in system ticks.

**Result**
- 0  No error.
- `slkErrSocketNotOpen`  The socket was not open.
Communications Functions

Serial Link Manager Functions

SlkOpen

<table>
<thead>
<tr>
<th>Purpose</th>
<th>Initialize the serial link manager.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Prototype</td>
<td>Err SlkOpen (void)</td>
</tr>
<tr>
<td>Parameters</td>
<td>None.</td>
</tr>
<tr>
<td>Result</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>No error.</td>
</tr>
<tr>
<td></td>
<td>slkErrAlreadyOpen</td>
</tr>
<tr>
<td></td>
<td>No error.</td>
</tr>
</tbody>
</table>

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. slkErrAlreadyOpen 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** Communications 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 ID’s, 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**
```c
Err SlkReceivePacket( UInt socket,
                      Boolean andOtherSockets,
                      SlkPktHeaderPtr headerP,
                      void* bodyP,
                      UInt bodySize,
                      Long timeout)
```

**Parameters**
- `socket` The socket ID.
- `andOtherSockets` If true, ignore actual dest 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 may 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 time out.

**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 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 non-contiguous 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 may not be an automatic variable (that is, local variable 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: 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.
Communications Functions
Serial Link Manager Functions

**SlkSocketRefNum**

**Purpose**
Get the reference number of the communications library associated with a particular socket.

**Prototype**
```
Err SlkSocketRefNum (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 time out for a particular socket.

**Prototype**
```
Err SlkSocketSetTimeout (UInt socket, Long timeout)
```

**Parameters**
- `socket` Socket ID.
- `timeout` Interbyte packet receive time out 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.

---

**PAD Server Functions**

**PsrClose**

**Purpose**

Close the PAD server.

**Prototype**

```
Err PsrClose(void)
```

**Parameters**

None.

**Result**

0  No error.

**Comments**

This routine frees resources allocated by the PAD server. It should be called when the PAD server client is finished using PAD server and only if the call to **PsrInit** was successful.

The routine **must** be called by the client when finished with the session if the call to **PsrInit** was successful.
Communications Functions
PAD Server Functions

**PsrGetCommand**

**Purpose**
Receive a command.

**Prototype**
```
Err PsrGetCommand(
    DmOpenRef refDBP, VoidPtr* cmdPP,
    VoidHand* cmdBufHP, WordPtr rcvdCmdLenP,
    BytePtr tidP, BytePtr remoteSocketP)
```

**Parameters**
- **-> refDBP**
  Database reference for allocating a command buffer, or 0 (zero) for none.

- **<-> cmdPP**
  Pointer to location for storing a pointer to the internal command buffer.

- **<-> cmdBufHP**
  Pointer to location for storing a handle of the command buffer allocated from a data storage heap.

- **<-> rcvdCmdLenP**
  Pointer to location for storing the size (in number of bytes) of the received command.

- **<-> tidP**
  Pointer to location for storing the transaction ID of the command.

- **<-> remoteSocketP**
  Pointer to location for storing the remote socket ID (the source socket).

**Result**
- 0: No error.
- **psrErrUserCan**
  Cancelled by user (Cancel callback returned non-zero).
- **psrErrParam**
  Invalid parameter.
- **psrErrBlockFormat**
  Invalid command data format detected (severe protocol error).
- **psrErrTimeOut**
  Timed out waiting for command.

**Comments**
PsrGetCommand blocks until a command is received, a time-out error occurs, or the Cancel callback (see **PsrInit**) returns non-zero. On success, the command is in the buffer, referenced either by *cmdPP or by *cmdBufHP. In the first case (cmdPP), the command will be in a Pad Server internal buffer in the dynamic heap. This buffer
must be treated as read-only. In the second case (cmdBufHP), the internal buffer was not big enough to contain the entire command (such as when writing a large record), and a data heap chunk was allocated by PAD server via DmNewHandle (provided that a valid refDBP was specified). The caller inherits ownership of this chunk and is responsible for freeing it if it is not needed (it can be resized, attached to a database, deleted, etc.).

**PsrlInit**

**Purpose** Initialize the PAD server.

**Prototype**

```c
Err PsrlInit ( Byte serverSocket,
              PsrUserCanProcPtr canProcP,
              DWord userRef,
              Int cmdWaitSec)
```

**Parameters**

- `>- serverSocket` Socket ID of an open Serial Link socket.
- `>- canProcP` Pointer to the Cancel callback procedure or 0 (zero) if none.
- `>- userRef` Any DWord(32-bit) parameter to be passed to the Cancel callback procedure.
- `>- cmdWaitSec` Number of seconds to wait for command; 0 = default; -1 = forever.

**Result**

- `0` No error.
- `psrErrInUse` PAD server is in use.
- `psrErrMemory` Insufficient memory to initialize PAD server.

**Comments**

This routine initializes the PAD server, allocating any necessary resources. Return code of 0 (zero) indicates success; any other return code indicates failure. If the PAD server was successfully opened by the client, the client needs to call PsrClose when it has finished using the PAD server. If specified, the cancel callback procedure is called periodically. If the cancel callback procedure returns non-zero, the current PAD server request aborts and returns immediately with an error code of psrErrUserCan.
**PsrSendReply**

**Purpose**
Send a response to the workstation.

**Prototype**
```
Err PsrSendReply (Byte remoteSocket,
                 Byte refTID,
                 PmSegmentPtr segP,
                 Int segCount)
```

**Parameters**
- `-> remoteSocket` Remote socket ID.
- `-> refTID` Transaction ID of the response (should be same as that returned by the matching PsrGetCommand call).
- `-> segP` Pointer to array of response data segments.
- `-> segCount` Number of reply data segments in the array.

**Result**
- `0` No error.
- `psrErrParam` Invalid ID parameter(s).
- `psrErrSizeErr` Sum of the response data segments exceeded PADP block size limit.
- `psrErrTooManyRetries` Maximum retry count was exceeded but acknowledgment wasn’t received. (connection is presumed lost).
- `psrErrTimeOut` Transmission handshake time out (connection is presumed lost).
- `psrErrUserCan` Cancelled by user (cancel callback returned non-zero).

**Comments**
PsrSendReply blocks until the entire response data block is successfully delivered to the workstation, lost connection is detected, or the cancel callback (see PsrInit) returns non-zero. For convenience, the response data block is specified as a list of data segments via an array of PmSegmentType structures. The PmSegmentType structure allows selective specification of word alignment for each data segment. Any bytes inserted as the result of word alignment are set to 0 (zero) in the resulting response block.
Miscellaneous Communications Functions

**Crc16CalcBlock**

**Purpose**  Calculate the 16-bit CRC of a data block using the table lookup method.

**Prototype**  

```c
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.