Device Abstraction Layer (DAL) Reference

Palm OS® 5 PDK/SPK

PalmSource Confidential

Written by Doug Fulton, Anna Schaller, Mark Dugger, and Christopher Bey.
Engineering contributions by Patrick Porlan, Michel Piquemal, Regis Nicolas, Thierry Martel, Vincent Leclaire, Larry Lai, Jim Schram, Steve Minns, Bruce Thompson, Frank Flonnoy, Tim Wiegman, Andy Stewart, Peter Wagner, Lee Taylor, and Mark Easterday.

DAL Reference
Document Number 5043-009
January 27, 2004

PalmSource, Inc.
1240 Crossman Avenue
Sunnyvale, CA 94089
USA
www.palmsource.com

Table of Contents

Part III Kernel Abstraction Layer (KAL)

About This Document

The Device Abstraction Layer (DAL) for Palm OS® 5 is software that separates the Palm OS from device-specific hardware. When the Palm OS is ported to a new hardware platform, the DAL functions must be implemented and fine-tuned to match the resources and requirements of that new hardware.

PalmSource, Inc. ships a sample DAL implementation that runs on a well-established reference board. The DAL must be modified to support your reference board and the hardware peripherals added to it. The code includes two trees: the `Development Kit\Palm_OS_DAL_Support` tree contains code that is relevant to all reference boards and, in general, need not be changed. Under the `\Development Kit\Samples\` directory you will find DAL reference implementation. It must be modified to support your hardware.

This guide describes the routines in the DAL that you are likely to modify. When modifying a routine, make sure you preserve the function prototype and purpose as described in this reference and in the comments in the source code. Most of these routines are exported from the DAL so that they can be called by the Palm OS. For a complete listing of routines that make up the DAL interface, consult the `DAL.mdf` file.

What This Document Contains

The principal sections of this document are devoted to the DAL's three components, the **Hardware Abstraction Layer** (**HAL**), the **Kernel Hardware Abstraction Layer** (**kHAL**), and the **Kernel Abstraction Layer** (**KAL**). A fourth DAL component, the **Runtime Abstraction Layer** (**RAL**), has no user servicable parts and so isn't described in this document.

The HAL

The Hardware Abstraction Layer (HAL) contains the DAL functions that provide the hardware-dependent implementation of fundamental Palm OS features. The Palm OS calls these functions to provide the services required by third-party applications and by other parts of the Palm OS. The HAL functions constitute a public interface for the Palm OS and must be preserved as such. HAL functions will probably need to be modified to work with specific hardware. For more details, see the Chapter 1, "The HAL," on page 1.

The HAL functions are divided into functional topics and presented in these sections:

- Chapter 1, "The HAL." Overview of HAL and modification of the HAL.
- Chapter 2, "Battery Support." Information about the device battery, listing of supported kinds, and status of cradle docking.
- Chapter 3, "Tracing." Facilities to follow a program's execution by outputting messages along with sender information.
- Chapter 4, "Digitizer Support." The *digitizer* is the software that decodes touch screen stylus input.
- Chapter 5, "Display." Routines that make up the display driver, including those that set and get the graphical attributes and dimensions supported by the screen.
- Chapter 6, "Initialization." System initialization, including hardware and memory initialization.
- Chapter 7, "Interrupt Handling." Enable/disabling interrupts, getting/setting interrupt handlers.
- Chapter 8, "Keyboard Support." Repeat actions, double-tap, and state of the hard keys on the device. Also a bitmask for identifying keys.
- Chapter 9, "Power States." Routines that handle power states, particularly the auto-off alarm.
- Chapter 10, "Miscellaneous Functions." All the functions that don't fit the other categories.

- Chapter 11, "Memory." Routines concerned with the memory map and memory protection.
- Chapter 12, "Real Time Clock Support." Getting/setting alarms and getting/setting the real-time clock.
- Chapter 13, "Serial Drivers." The routines that a virtual (serial) driver must support in order to work with the new serial manager of the Palm OS.
- Chapter 14, "Screen." Drawing primitives contained in the blitter. Screen Manager routines described as well.
- Chapter 15, "Sound Support." Turning sounds on and off.
- Chapter 16, "Timer Support." Timed delay.

The kHAL

The Kernel Hardware Abstraction Layer (kHAL) lies face-down just above the kernel (Palm Kernel 1.0). The kHAL defines a set of functions that you implement to run on a specific ARM CPU.

The kHAL is documented in Chapter 17, "kHAL Functions," on page 147.

The KAL

The Kernel Abstraction Layer provides objects, such as tasks, semaphores, timers, and so on, that are allocated and defined by the kernel. You can't reimplement the KAL; however, you can use the functions and structures it defines in your implementations of the HAL and kHAL functions.

Chapter 18, "The KAL," on page 157 provides a brief definition of each of the KAL objects. The rest of the KAL chapters do the real work:

- Chapter 19, "KAL Generic API."
- Chapter 20, "Tasks."
- Chapter 21, "Semaphores."
- Chapter 22, "Mutexes."
- Chapter 23, "Event Groups."
- Chapter 24, "Mailboxes."

- Chapter 25, "Timers."

Related Documentation

The following documents contain information that will further your education on the DAL for Palm OS 5. All these documents pertain to the Palm OS on the ARM platform only. Like-titled documents from PDKs prior to Palm OS 5 pertain to the Motorola 68000 platform only.

The DAL for Palm OS 5 differs in specific, though very limited, areas from earlier releases of the DAL for Palm OS 5. In addition, the SDK (Software Development Kit) for Palm OS 5 has undergone minor revisions to support new technologies. Consult documentation in the SDK for details.

Document	Description
Introduction to the PDK	Guide that orients you to all the kits, tools, code, and documentation on the PDK (Product Development Kit).
Architectural Overview	The manual provides background and conceptual information on the design of Palm OS 5.
Shared Library Design Guide	This manual provides information on customizing Palm OS using ARM-native code. Discussion includes writing ARM shared libraries, integrating ARM code with 68K applications, and creating OS patches.
DAL Customization Guide	The manual provides background, conceptual, and how-to information on the Device Abstraction Layer of the ROM image. This manual complements the APIs discussed in the *DAL Reference*. It provides the common design and implementation information that is needed to port the Palm OS® to a custom hardware platform. Information related to specific technologies can be found in the relevant technology manual.

Document	Description
DAL Reference	This manual is a companion to the *DAL Customization Guide*. It describes the API routines in the Hardware Abstraction Layer (HAL), the kernel Hardware Abstraction Layer (kHAL), and the Kernel Abstraction Layer (KAL) . These routines serve two purposes. They are either modifed by you to accomodate specific hardware features, or called to accomplish a particular task.
Building a ROM	This guide begins by providing a description of the various ROM components. It then describes the tools and steps needed to integrate the DAL, the Palm OS®, and the built-in applications into an image for installation in flash ROM or in RAM.
Display Driver Design Guide	Technology guide on creating a hardware-specific display driver that communicates with the screen manager and the blitter routines.
Serial Communications Driver Design Guide	Technology guide on writing virtual communication drivers. Supported drivers include serial as well as USB.
Expansion Manager Solutions Guide	Technology guide that provides you with background information and instruction on extending Palm OS to include expansion slot technology. The information in this guide builds on the Expansion Manager and VFS Manager chapters in the *Palm OS Programmer's Companion* and *Palm OS Programmer's API Reference*.
Sound Driver Design Guide	Technology guide to creating a hardware-specific sound driver that communicates with the Sound Manager.
Ethernet Interface Driver Design Guide	Technology guide to implementing an Ethernet interface on the Palm OS. This document is particularly relevant to those implementing wireless Ethernet interfaces such as IEEE 802.11b.

Document	Description
Customizing Palm OS Simulator	Guide to creating a custom version of the Palm OS Simulator.
Debugging a ROM Image	This manual provides conceptual, guidance, and reference information for developers who want to use Palm OS Debugger to debug Palm OS applications and shared libraries.
Building Palm OS Application Interfaces	This book describes a set of developer tools that you can use to create, edit, process, and compile Palm OS resources--forms, menus, text strings, and controls--for Palm OS applications. The Palm OS resource tools operate on an *XRD* file format rather than Macintosh resource binary format (RSRC) format that was previously used. GenerateXRD, PalmRC, and PRCMerge are the tools are used in this process.

Additional Resources

- Documentation

  PalmSource, Inc. publishes its latest versions of documents for Palm OS developers at

  http://www.palmos.com/dev/support/docs/

- Training

  PalmSource, Inc. and its partners host training classes for Palm OS developers. For topics and schedules, check

  http://www.palmos.com/dev/training

- Knowledge Base

  The Knowledge Base is a fast, web-based database of technical information. Search for frequently asked questions (FAQs), sample code, white papers, and the development documentation at

  http://www.palmos.com/dev/support/kb/

Typographical Conventions

`Bold text`
: indicates emphasis.

`Courier font`
: is used for functions, types and file names.

: `underlining indicates a hyperlinked cross-reference.`

`Italic Text`
: must be replaced by the user with appropriate information.

The following symbols are used to indicate whether an argument's value is set by the caller, by the function, or both:

- `->` the argument's value is set by the caller
- <- the argument's value is set by the function
- <-> the argument's value is set by the caller and then reset by the function

Part I
Hardware Abstraction Layer (HAL)

1

The HAL

This section describes the HAL, or Hardware Abstraction Layer, which contains routines that provide the basic functionality of the Palm OS. The HAL is a a mediating layer between the Palm OS and the underlying hardware, insulating the Palm OS from the need to know about hardware implementation. For example, when the Palm OS system manager wants to know how much life is left in the battery, it calls the HAL function `HALBatteryGetInfo`, which function talks to the hardware. Only the Palm OS—and other HAL functions—can invoke the HAL routines. They can't be called from third-party application software.

The routines of the HAL must be adapted to the particular hardware of a device. Thus, the HAL is different for each basic ARM-processor reference board. The HAL undergoes further modification when supplemental hardware peripherals are added to the reference board.

HAL Interface

While customizing HAL functions for your hardware, keep in mind that the functions described here form a public interface required by the Palm OS. You must continue to support these functions for Palm OS to work properly. Your implementations must preserve the function prototype, accomplish the same purpose, and perform the same tasks as those in the sample HAL.

A few routines that are purely internal to the HAL are included in this reference if they are likely to require modification. Otherwise, routines called only by other HAL routines are not documented here. Please consult the source code for details on their operation. The HAL interface is a subset of the DAL interface. For a complete listing of the routines that make up the DAL interface, see the `DAL.mdf` file.

Sample HAL

The sample DAL shipped by PalmSource, Inc. includes a sample HAL. This implementation was written for the DBPXA25x or DBPXA26x reference boards from Intel. The files are located in two main places. Files that you will not need to modify are found under the `Development Kit\Palm_OS_DAL_Support` directory. These will generally remain the same for all reference boards. Files that you modify for your hardware are shipped to you under `\Development Kit\Samples\` directory. You will probably clone these files and then start modifying them to match your hardware requirements.

File names

Each function description in this reference includes the name of the file in which the function is defined. Files that are under the `Development Kit\Palm_OS_DAL_Support` directory have generic names such as `ROMBoot.c` and `HALDebug.c`, because they apply to all reference boards. Files that you must modify are shipped with names such as `LBC_KeyMgr.c` and `CTLDisplayBoot.c` . They are all found under the `Development Kit\Samples` directory of the development kit.

When you clone the sample to establish your source code, you will most likely attach the name of your particular reference board as a prefix to the names of files.

Assembly-Language Code

For the sake of performance, a few very low-level routines are written in ARM assembly language. Examples are `RegionInit_A.s` and `Reset_A.s`.

The routines written in assembly language, of course, interact with C-language functions. In some cases, they use symbolic constants that are conceptually the same and must have the same value. For instance, a constant value might be defined by a `#define` in a C-language header and by an `EQU` in the assembly language header. Modifying one constant definition is not sufficient: the assembly pre-processor replaces only occurrences of the `equate` identifier.

Occurrences of the `#define` identifier are replaced at a separate time by the C compiler's pre-processor.

IMPORTANT: When modifying constant values in an assembly language routine, make sure to modify any C-language files that use the same constants. The same advice can apply to a variable and a constant, although this situation is less common.

Whenever you modify the HAL source code, pay attention to the comments. Important dependencies like these are called out in the code comments.

Intended Audience

The intended audience for this section is the ARM expert responsible for porting Palm OS to a new ARM CPU platform.

The development kit you received includes a sample HAL implementation on a well-known reference board. In general, you will start with that code and modify it to support the specific hardware of your reference board and device.

What This Section Contains

The HAL section of the DAL Reference includes the following chapters and topics:

- Chapter 1, "The HAL." Overview of HAL and modification of the HAL.
- Chapter 2, "Battery Support." Information about the device battery, listing of supported kinds, status of cradle docking.
- Chapter 3, "Tracing." Facilities to follow a program's execution by outputting messages along with sender information.
- Chapter 4, "Digitizer Support." The "digitizer" is the software that decodes touch screen stylus input.
- Chapter 5, "Display." Routines that make up the display driver, including those that set and get the graphical attributes and dimensions supported by the screen.

- Chapter 6, "Initialization." System initialization, including hardware and memory initialization.
- Chapter 7, "Interrupt Handling." Enable/disabling interrupts, getting/setting interrupt handlers.
- Chapter 8, "Keyboard Support." Repeat actions, double-tap, and state of the hard keys on the device. Also a bitmask for identifying keys.
- Chapter 9, "Power States." Routines that handle power states, particularly the auto-off alarm.
- Chapter 10, "Miscellaneous Functions." All the functions that don't fit the other categories.
- Chapter 11, "Memory." Memory map and memory protection.
- Chapter 12, "Real Time Clock Support." Getting/setting alarms and getting/setting the real-time clock.
- Chapter 13, "Serial Drivers." The routines that a virtual (serial) driver must support in order to work with the new serial manager of the Palm OS.
- Chapter 14, "Screen." Drawing primitives contained in the blitter. Screen manager routines described as well.
- Chapter 15, "Sound Support." Turning sounds on and off.
- Chapter 16, "Timer Support." Timed delays.

2

Battery Support

The DAL implementation must enqueue a low battery event when the battery level is low enough to need user attention.

This chapter describes the API functions of the HAL that deal with the battery. The include files `HALBattery.h` and `HALDock.h` contain definitions for the data types and routines discussed in this chapter.

Battery Support Constants

HALDockStatus Constants

The following constants are returned from `HALDockStatus`. They indicate which state the hardware is in. The constants are defined in the include file `HALDock.h`

Constant	Value	Description
`kHALDockStatusUndocked`	0x0000	Nothing is attached to the connector.
`kHALDockStatusModemAttached`	0x0001	Some type of modem is attached to the connector.
`kHALDockStatusDockAttached`	0x0002	Some type of dock is attached to the connector.
`kHALDockStatusUsingExternalPower`	0x0004	The connector is using some type of external power source.
`kHALDockStatusCharging`	0x0008	Connector is in cradle and internal power cells are recharging.
`kHALDockStatusUSBCradleAttached`	0x0010	USB hardware is attached to the connector.

Constant	Value	Description
`kHALDockStatusUSBPeripheralAttached`	0x0020	USB Peripheral is attached to the connector.
`kHALDockStatusPeripheralAttached`	0x0040	RS232 Peripheral is attached to the connector.
`kHALDockStatusMfgTestCradleAttached`	0x0080	The manufacture's cradle is attached to the connector.

Battery Support Data Structures

SysBatteryKind

The `SysBatteryKind` data type is used to declare parameters that indicate the kind of battery used by the handheld. The constants defined in the `SysBatteryKindTag` enumeration correspond to the different kinds of batteries, such as Alkaline or Lithium Ion. These constants are passed to HALBatteryGetInfo and HALBatterySetInfo, and HALBatteryGetValidKinds. The definition for `SysBatteryKindTag` is found in `CmnBatteryTypes.h`.

All the batteries defined here are removable batteries, except `sysBatteryKindLiIon` and `sysBatteryKindLiIon1400`. The latter are rechargeable from the device. A given device HAL can provide the following battery support: either support for all removable kinds or support for *one* of the two LiIon kinds.

```
enum SysBatteryKindTag {
  sysBatteryKindAlkaline=0,
  sysBatteryKindNicad,
  sysBatteryKindLiIon,
  sysBatteryKindRechAlk,
  sysBatteryKindNiMH,
  sysBatteryKindLiIon1400,
  sysBatteryKindLast=0xFF
};

typedef Enum8 SysBatteryKind;
```

Field Description

`sysBatteryKindAlkaline`

`sysBatteryKindNicad`

`sysBatteryKindLiIon`

`sysBatteryKindRechAlk`

`sysBatteryKindNiMH`

`sysBatteryKindLiIon1400`

`sysBatteryKindLast`

SysBatteryState

The `SysBatteryState` data type is used to declare parameters that indicate the state of the handheld's battery. The constants defined in the `SysBatteryStateTag` enumeration correspond to the different battery states. These constants are passed to HALBatteryGetInfo. The definition for `SysBatteryStateTag` is found in `CmnBatteryTypes.h`.

```
enum SysBatteryStateTag {
  sysBatteryStateNormal=0,
  sysBatteryStateLowBattery,
  sysBatteryStateCritBattery,
  sysBatteryStateShutdown,
};

typedef Enum8 SysBatteryState;
```

Field Description

`sysBatteryStateNormal`

`sysBatteryStateLowBattery`

`sysBatteryStateCritBattery`

`sysBatteryStateShutdown`

Battery Support Functions

HALBatteryGetInfo Function

Purpose This function returns information about the handheld's battery: the kind of battery, the battery state, the percent of power left in the battery, and whether the handheld is connected to an external power supply.

Prototype
```
Err
    HALBatteryGetInfo(UInt16 *oWarnThresholdPercen
    t, UInt16 *oCriticalThresholdPercent,
    UInt16 *oShutdownThresholdPercent,
    UInt32 *oWarnMaxTicks, SysBatteryKind *oKind,
    UInt8 *oPluggedIn, SysBatteryState *oState,
    UInt8 *oPercent)
```

Parameters

←*oWarnThresholdPercent*
Pointer to the voltage warning threshold, or null.

←*oCriticalThresholdPercent*
Pointer to the voltage critical threshold, or null.

←*oShutdownThresholdPercent*
Pointer to the voltage shutdown threshold, or null.

←*oWarnMaxTicks*
Pointer to timeout until next battery warning, or null.

←*oKind*
Kind of battery supported by the handheld. The values for this parameter are the fields defined in the enumeration `SysBatteryKindTag`. Refer to the section "SysBatteryKind" for a description of the available constants.

←*oPluggedIn*
Non-zero if external power is currently being supplied to the handheld.

←*oState*
Current state of the battery, computed from its voltage and kind. The values for this parameter are the fields defined in the enumeration `SysBatteryStateTag`. Refer to the section "SysBatteryState" for a description of the available constants.

←*oPercent*
Percent return value of remaining load.

Returns `0`
If no error.

Comments `LBC_HALBattery.c`

Replaces `HwrBattery()` and `HwrPluggedIn()` functions from the HAL API for Palm OS 4.0.

See Also `HALBatterySetInfo`
`HALBatteryGetValidKinds`
`HALDockStatus`
`SysUILaunch`

HALBatteryGetValidKinds Function

Purpose This function returns a list of available battery types.

Prototype

```
Err HALBatteryGetValidKinds
    (const SysBatteryKind **oKind)
```

Parameters ←*oKind*
A pointer to an array of valid battery kinds. The last valid value will always be `sysBatteryKindLast`. The values for this array are the fields defined in the enumeration `SysBatteryKindTag`. Refer to the section "SysBatteryKind" for a description of the available constants.

Returns `0`
If no error.

Comments `LBC_HALBattery.c`

See Also `HALBatteryGetInfo`
`HALBatterySetInfo`
`HALDockStatus`
`SysUILaunch`

HALBatterySetInfo Function

Purpose This function allows the battery type to be modified dynamically.

Prototype

```
Err HALBatterySetInfo(SysBatteryKind *iKind,
    UInt16 *oWarnThresholdPercent,
    UInt16 *oCriticalThresholdPercent,
    UInt16 *oShutdownThresholdPercent)
```

Parameters

→*iKind*
Kind of battery supported by the handheld. The values for this parameter are the fields defined in the enumeration `SysBatteryKindTag`. Refer to the section "SysBatteryKind" for a description of the available constants.

←*oWarnThresholdPercent*
Pointer to the voltage warning threshold, or null.

←*oCriticalThresholdPercent*
Pointer to the voltage critical threshold, or null.

←*oShutdownThresholdPercent*
Pointer to the voltage shutdown threshold, or null.

Returns

`0`
If no error.

Comments `LBC_HALBattery.c`

Replaces `HwrBattery()` function from the HAL API for Palm OS 4.0.

See Also

`HALBatteryGetInfo`
`HALBatteryGetValidKinds`
`HALDockStatus`
`SysUILaunch`

HALDockStatus Function

Purpose Returns a bitmap indicating the hardware docking status. Note that docking is different from being connected to power. Docking implies a hardware connection for the HotSync. There are bitfield constants defined for the different possible states, such as undocked, modem attached, USB cradle attached, etc. The bitmap can have more than one bit set at any given time.

Prototype `Err HALDockStatus(UInt16 *status)`

Parameters

← *`status`*

Bitfield. Refer to the section on "HALDockStatus Constants" on page 5 for a description of the values returned from this routine.

Returns

`0`

If no error.

Comments `HALDock.c`

This function is called by `SysBatteryInfo()`,`HwrPluggedIn()`, and `HwrDockSignals()`. The bit definitions for the inputs and outputs are defined in `HALDock.h`.

All unused bits in the `bitmap` are reserved for future use by Palm.

See Also

`HALBatteryGetInfo`
`HALBatterySetInfo`
`HALBatteryGetValidKinds`
`SysBatteryInfo`

3
Tracing

In the Palm OS context, *traces* are an easy way to follow a program's execution by outputting messages along with sender information. By using traces in conjunction with a tool like Serial Reporter or HyperTerminal, it is easy to automatically sort messages and provide automatic filtering while debugging.

If you decide to implement these functions, you may also have to modify two other functions in `HALTrace.c`. They are `HALTraceInit()` and `HALTraceClose()`.

For details on the tracing facilities, see "Debugging" in *DAL Customization Guide*.

Trace Data Structures

None applicable.

Trace Functions

The APIs described in this section provide an easy way to communicate with the Reporter and to format raw data like memory buffers.

"B" suffix means "trace buffer". This ends up in an organized hexadecimal dump in the Reporter.

"VT" and "VTL" suffixes mean the function is called like the standard C function `vprintf`. The "L" means "add carriage return at end of trace".

"T" and "TL" suffixes mean the function is called like the standard C function `printf`. The "L" means "add carriage return at end of trace".

HALTraceClose

Function that takes no arguments and returns nothing. Defined in `LBC_ROMHardware.c`. It closes the serial connection opened by `HALTraceInit`.

HALTraceInit

Function that takes no arguments and returns nothing. You must explicitly call this function before you call the other trace functions. A good place to do so is in the `HALSetInitiStage(UInt32 uiValue)` routine in the `LBC_ROMHardware.c` file, when `uiValue` equals 3. See also `HALTraceClose`.

HALTraceOutputB Function

Outputs a memory dump trace to a trace receiver/display system, normally the Serial Reporter with Palm Reporter.

Prototype

```
void HALTraceOutputB(unsigned short aErrModule,
   const void *aBuffer, long aBufferLen)
```

Parameters

→*aErrModule*
An error class, must be unique among all modules that use traces.

→*aBuffer*
Pointer to the buffer to dump.

→*aBufferLen*
Length of area to dump, in bytes.

Returns None.

Comments `HALTrace.c`

HALTraceOutputT Function

Outputs a line of text trace to a trace receiver/display system, normally the Serial Reporter with Palm Reporter. This function is analogous to the standard C function `printf`.

Prototype
```
void HALTraceOutputT(unsigned short aErrModule,
   const char *aFormatString, ...)
```

Parameters
→*aErrModule*
An error class, which must be unique among all modules that use traces.

→*aFormatString*
Format string for arglist.

. . .Variable-length argument list to trace.

Returns None.

Comments `HALTrace.c`

`HALTraceOutputTL` is identical to this function, except that it adds a carriage return at the end of the trace.

HALTraceOutputVT Function

Outputs a line of text trace to a trace receiver/display system, normally the Serial Reporter with Palm Reporter. This function is analogous to the standard C function `vprintf`.

Prototype
```
void HALTraceOutputVT(unsigned short aErrModule,
   const char *aFormatString, const void
   *arglist)
```

Parameters
→*aErrModule*
An error class, which must be unique among all modules that use traces.

→*aFormatString*
Format string for arglist.

→*argList*
Argument list to trace.

Returns None.

Comments `HALTrace.c`

`HALTraceOutputVTL` is identical to this function, except that it adds a carriage return at the end of the trace.

Trace Macros

The functions described in the preceding section have a one-to-one correspondence with a set of similarly-named macros. You will find the macros called throughout the code, rather than the functions. The following table lists the syntax for each macro followed by the function call to which the preprocessor expands it. The ellipses (. . .) represent where you would specify a variable-length argument list, such as the ones you pass to the standard C library function `printf()`.

You will find the definition of these macros in `HALTrace.h`:

```
#define TraceOutput(X)   HALTraceOutput##X
```

It uses the preprocessor operator ## to concatenate the replacement text with the argument.

Table 3.1 Syntax of Trace Macros and Their Expansions

`TraceOutput(T   (errorClass, "formatString", ...)   )` `HALTraceOutputT(errorClass, "formatString",  ...)`
TraceOutput(TL (errorClass, “formatString”, ...)) HALTraceOutputTL(errorClass, “formatString”, ...)
TraceOutput(B (errorClass, bufferPtr, bufferLength)) `HALTraceOutputB(errorClass, bufferPtr,  bufferLength)`
TraceOutput(VT (errorClass, “formatString”,argListPtr)) `HALTraceOutputVT (errorClass, "formatString", argListPtr)`
TraceOutput(VTL (errorClass,“formatString”,argListPtr)) `HALTraceOutputVTL (errorClass, "formatString", argListPtr)`

4

Digitizer Support

This chapter describes the API functions of the HAL that deal with the digitizer. They are described in alphabetical sequence.

The silkscreen region of the display typically varies from device to device. Consequently, the HAL must provide a mechanism for determining the characteristics of this region. In addition, this mechanism must allow for patching the information so that different silkscreen regions can also be associated with a single device. An example of the latter is the Japanese version of the Palm V.

The DAL implementations must enqueue and return a pen down event to Palm OS when the pen is detected to be down on the digitizer. After the pen is first down, pen down events must be enqueued every (`1/PEN_SAMPLING_RATE`) second with new valid raw pen coordinates, until the pen is detected to be up. Upon pen up, a pen up event must be enqueqed and returned to Palm OS with coordinates of (`-1, -1`).

The Palm OS provides the UI for collecting the calibration information; however, the specific calibration algorithm and result storage is platform dependent.

For more information about the Pen Manager, see the *Palm OS Programmer's Companion* and the *Palm OS Programmer's API Reference*.

Digitizer Support Data Structures

Coord

Found in: `PalmTypes.h`

The `Coord` data type identifies one coordinate of a point. The PointType data type uses the Coord data type twice to identify the x- and y- coordinates of a point on the screen.

```
typedef Int16 Coord;
```

PointType

Found in: `CmnRecTypes.h`

The `PointType` data type uses x- and y-coordinates to identify a point on a screen or window. It is used extensively by digitizer functions, including `HALPenCalibrate`, and `HALPenScreenToRaw`.

```
typedef struct PointType {
  Coord x;
  Coord y;
  } PointType;
```

Digitizer Support Functions

HALPenCalibrate Function

Purpose Sets calibration of the pen, from top left and bottom right points for screen and digitizer.

Prototype
```
Err HALPenCalibrate (PointType *DigTopLeftP,
    PointType *DigBotRightP, PointType
    *ScrTopLeftP, PointType *ScrBotRightP)
```

Parameters → *`DigTopLeftP`*
Digitizer output from top-left coordinate.

→*DigBotRightP*
Digitizer output from bottom-right coordinate.

→*ScrTopLeftP*
Screen coordinate near top-left corner.

→*ScrBotRightP*
Screen coordinate near bottom-right corner.

Returns `0`
If no error.

Comments `LBC_PenMgr.c`

Replaces `PenCalibrate()` function from the HAL API for Palm OS 4.0.

The DAL needs to make sure that the necessary parameters for transforming between raw digitizer coordinates and screen coordinates have been calculated and stored. How the DAL accomplishes this task is platform dependent.

See Also `HALPenResetCalibration`

HALPenRawToScreen Function

Purpose This function converts a raw pen coordinate into screen coordinates.

Prototype `Err HALPenRawToScreen(PointType *ioPoint)`

Parameters ↔ *ioPoint*
Coordinate to be converted.

Returns `0`
If no error.

Comments `LBC_PenMgr.c`

Replaces `PenRawToScreen()` function from the HAL API for Palm OS 4.0.

This function is called by `EvtGetSysEvent()`, `EvtDequeueStrokeInfo()`, `EvtDequeuePenPoint()`, and `EvtGetPen()` before returning pen coordinates to the application.

See Also

`HALPenScreenToRaw`
`EvtGetSysEvent`
`EvtDequeuePenStrokeInfo`
`EvtDequeuePenPoint`
`EvtGetPen`

HALPenResetCalibration Function

Purpose This function resets the calibration in preparation for recalibrating the pen again.

WARNING! The digitizer is no longer calibrated after calling this routine and must be calibrated again!

Prototype `Err HALPenResetCalibration (void)`

Parameters None.

Returns `0`

No error; 0 is always returned.

Comments `LBC_PenMgr.c`

Replaces `PenResetCalibration()` function from the HAL API for Palm OS 4.0.

This function is called by the Preferences application, before capturing points, when calibrating the digitizer. It must be called before points are captured from the digitizer for calibration.

See Also `HALPenCalibrate`

HALPenScreenToRaw Function

Purpose This function converts a screen coordinate into a raw digitizer coordinate.

Prototype `Err PenScreenToRaw(PointType *ioPoint)`

Parameters ↔ *ioPoint*
Coordinate to be converted.

Returns `0`
If no error.

Comments Replaces `PenScreenToRaw()` function from the HAL API for Palm OS 4.0.

This function is called by the SerialLink Manager when processing a remote pen event from the host with `PrvProcessRemoteEvt()`. It must call this routine in order to modify the point as if it had come from the digitizer, because the `SysEvtMgr()` always calls `PenRawToScreen()` on enqueued points.

See Also `HALPenRawToScreen`
`PrvProcessRemoteEvt`
`SysEvtMgr`

5

Display

This chapter describes functions that are part of the display driver. They are described in alphabetical sequence. For details, see "Writing a Display Driver" in *Display Driver Design Guide*.

Display Data Constants

Display Attribute Constants

The following constants deal with display attributes, such as amount of VRAM, bit depth, brightness level, etc. Used by `HALDisplayGetAttributes` and `HALDisplaySetAttributes`, these constants are defined in the include file `HALDisplay.h`.

In the table, the `Get/Set` column indicates whether each constant can be used only to get display attributes (`Get`) or to both get and set display attributes (`Both`).

Constant	Get/ Set	Description	Value Returned by Get or Set
`kHALDispAddr`	Both	Base address for video memory. Screen Manager uses this base for determining address of the destination bitmap. Always returns a memory address, whether video memory is physically located on VRAM or is allocated from the dynamic heap.	Valid memory address.
`kHALDispAllDepth`	Get	A mask indicating all supported bit depths.	If a depth is supported, the bit at depth-1 is set. For example: a display driver that supports 2-bit grayscale and monochrome would have a mask value of 0x03. A driver that supports 8-bit color as well as 4,2, and 1-bit grayscale would have a mask value of 0xAB.
`kHALDispAllowDirectAccess`	Get	A Boolean that indicates if direct access to video memory is allowed.	Values are `0,1` (disallowed/allowed)
`kHALDispBacklight`	Both	Boolean for backlight status.	Values are `0,1` (off/on)
`kHALDispBootDepth`	Get	Bit depth used when system is being unitized. Typically set to lowest bit depth supported: 1 for monochrome/grayscale or 8 for color.	Unsigned integer representing number of bits.

Constant	Get/ Set	Description	Value Returned by Get or Set
`kHALDispBufferMask`	Get	Mask for determining required display address alignment. Set this value only if video memory is allocated from the system heap and the allocated block must be aligned to a certain address boundary.	Bitmask.
`kHALDispBrightness`	Both	A value representing the current brightness level	0 (min) - 255 (max)
`kHALDispColor`	Get	A Boolean showing whether the controller and the display can show color.	`True` if both controller AND display support color. `False` if both controller and display support only gray scale.
`kHALDispContrast`	Both	A value representing the current contrast level.	0 (min) - 255 (max)
`kHALDispDbgIndicator`	Get	A flag that shows current state of debug cursor. Can be used by HAL for debugging. For instance, the indicator could be a blinking cursor to denote boot progress at various times.	`True` if debug cursor is displayed, indicating that execution has been halted, awaiting input from debugger.
`kHALDispDensity`	Get	Returns the screen density.	Screen density, using `DensityType` enum constants defined in `CmnBitmapTypes.h`.

Constant	Get/ Set	Description	Value Returned by Get or Set
kHALDispDgtScale	Get	Returns density, as a fixed point value, of digitizer used for pen samples in relation to density of screen. Value is 16-bit fixed-point.	FixedFromInteger: if digitizer accurate enough to return coordinates that match screen density.
			kFixedOneHalf: if display is double density but digitizer can accurately return only single density pen samples
kHALDisDgtStdScale	Get	Scaling factor for converting digitizer pen coordinates to standard coordinates. Standard coordinates are single density coordinates. Value is 16-bit fixed point.	kFixedOneHalf: if digitizer generates double-density pen samples. Standard coordinates are single density coordinates.
			FixedFromInteger: if digitizer returns single density pen samples.
kHALDispDepth	Both	Current bit depth being used.	Unsigned integer representing number of bits.
kHALDispEndAddr	Get	Address of last byte of video memory. No longer used by OS.	Unsigned integer representing number of bits.
kHALDispHeight	Get	Physical height of display in pixels.	In pixels.

Constant	Get/ Set	Description	Value Returned by Get or Set
`kHALDispInputAreaBmp`	Both	Pointer to a `BitmapType` that contains the regular input area bitmap if it is not printed on the display. The display driver must support both get and set of this bitmap if the device supports a dynamic input area.	Pointer
`kHALDispInputAreaLoc`	Both	A `RectangleType` that specifies the bounds of the input area. Define this attribute only if the input area is not printed on the display. If this attribute is defined, Palm OS® draws the input area bitmap to the specified location at boot time and then sends the bitmap pointer to the display driver using the `kHALDispInputAreaBmp` attribute. Therefore, you must support setting the `kHALDisplayInputAreaBmp` attribute in `HALDisplaySetAttributes()` if you define an input area location.	Bounds rectangle.

Constant	Get/ Set	Description	Value Returned by Get or Set
`kHALDispInputAreaSelectedBmp`	Both	Pointer to a `BitmapType` that contains the selected input area bitmap. This bitmap looks similar to the regular bitmap (`kHALDispInputAreaBmp`) except that each button is drawn in its inverted (selected) state. `HalRedrawInputArea()` uses this bitmap when its *selected* parameter is true. The display driver must support both get and set of this bitmap if the device supports a dynamic input area.	Pointer
`kHALDispMaxDepth`	Get	Maximum bit depths. Supported LCD depths bit mask (color depth or gray levels depending on `kHALDisplayColor` value). Output value is a bitfield of supported screen depths.	Each bit in the returned UInt16 value indicates support (1) or not (0) for each display depth. To decode an individual bit in the bit map, use the following: `bit position = bit depth - 1` Some examples: Support for bit depths of 2 and 1 is indicated by 0x03 Support for bit depths of 4, 2, and 1 is indicated by 0x0B. Support for bit depths of 24, 8, 4, and 2 is indicated by 0x80008A.
`kHALDispMemAccessOK`	Get	Boolean that indicates if drawing to screen when the controller is disabled causes a bus error.	`True` if drawing does not cause bus error. `False` if drawing causes error.

Constant	Get/ Set	Description	Value Returned by Get or Set

`kHALDispPixelFormat`	Get	Pixel format of screen.
		Returns one:
		`a) pixelFormat565LE`
		`b) pixelFormatIndexed,`
		`c) pixelFormatIndexedLE`
			Constants are for these bit depths;
			a) 16-bit
			b) 8-bit
			c) 1,2,or 4-bit

`kHALDispName`	Get	Name of display driver	Get a 32-character string for controller and hardware.
`kHALDispResolutionX`	Get	Not currently used.	
`kHALDispResolutionY`	Get	Not currently used.	
`kHALDispRev`	Get	Controller hardware's revision number.	Unsigned integer.
`kHALDispRowBytes`	Get	Number of bytes it takes store a single row of pixels.	Equals: width*bitdepth / 8
`kHALDispType`	Get	4-character constant specifying the controller/display combination	4 characters
`kHALDispVers`	Get	Display driver's version number	Unsigned integer.
`kHALDispVRAMAddr`	Get	Base address for video memory physically located on VRAM. See `kHALDispAddr`	Valid memory address. If video memory is allocated from the dynamic heap, this value is 0.
`kHALDispVRAMSize`	Get	Size in bytes of video memory	In bytes. Is 0 if video memory is allocated from the dynamic heap.
`kHALDispWidth`	Get	Width of display in pixels	In pixels.

Constant	Get/ Set	Description	Value Returned by Get or Set
`kHALDispXferBufStatic`	Get	Boolean indicating if intermediate buffer used by display transfer function is statically allocated. Used only if licensee is using an intermediate buffer.	`True` if statically allocated. `False` if allocated from the dynamic heap.
`kHALDispXferDepths`	Get	Bit depths at which the display transfer function is used. Bitfield returned that represents zero or more depths for which the display transfer function will be called. Note that some DAL implementations might call transfer function only for higher bit depths. Used only if licensee is using an intermediate buffer.	Each bit in the returned UInt16 value indicates support (1) or not (0) for each display depth. To decode an individual bit in the bitfield, use the following: `bit position = bit depth - 1.`
`kHALDispXferFunc`	Get	Display transfer function used to translate between what blitter draws and what driver expects. Used only if licensee is using an intermediate buffer.	Function pointer.

Display Data Structures

RGBColorType

Found in: `CmnBitmapTypes.h`

The `RGBColorType` data type indicates the color of a pixel, using the red-green-blue system. This structure specifies the amount of red, green, and blue (on a scale of 0-255) that combine to make this color. The `index` field is the color, or closest matching color, in the current CLUT; otherwise, it is unused. The color table in question is table of non-sequential colors, which contains a maximum of 256. The color table is represented by the `ColorTableType`, which

contains an array of `RGBColorType` structures.
See"ColorTableType" on page 109 for more information about the `ColorTableType`.

```
typedef struct RGBColorType {
  UInt8 index;
  UInt8 r;
  UInt8 g;
  UInt8 b;
  } RGBColorType;
```

Display Functions

HALDisplayDrawBootScreen Function

Purpose This function is used by the ROM startup code to put bitmaps on the screen. It displays a given bitmap at boot time, before any high-level initialization of the user interface.

Prototype
```
Err HALDisplayDrawBootScreen(UInt16 x, UInt16 y,
    void *bitmapParamP)
```

Parameters

→*x*
: x-coordinate of top-left corner for bitmap. It must be a multiple of 8.

→*y*
: y-coordinate of top-left corner for bitmap.

→*bitmapParamP*
: Pointer to bitmap structure to draw (`BitmapType`).

Returns `0`
: If no error.

Comments `CTLDisplay.c`

Replaces `HwrDisplayDrawBootScreen()` function from the HAL API for Palm OS 4.0.

`HALDisplayDrawBootScreen()` is called from `SysLaunch` using the splash screen bitmap specified when building the ROM. It can potentially be called multiple times, and with different bitmaps.

For example, it's usually called only once for the initial display, but it may be called again if the user held the page down key during the early boot process, to draw the confirmation screen, and again after the user confirms a hard reset is required in order to "restore" the splash screen while the hard reset is taking place. All this occurs before the hardware is "identified" by calling `HwrIdentifyFeatures()` and `HwrModelSpecificInit()`.

Please note that the `x` and `y` arguments passed to this function could also be of the signed data type `Coord`.

HALDisplayGetAttributes Function

Purpose This function returns the various attributes of the LCD controlling hardware.

Prototype

```
Err HALDisplayGetAttributes(UInt16 iAttribute,
    UInt32 *oValue)
```

Parameters

→*iAttribute*
Constant representing the attribute you wish to retrieve.

←*oValue*
A pointer to the attribute's value.

See "Display Attribute Constants" on page 23 for more information about display attributes and their values.

Returns `0`
If no error.

Comments `CTLDisplay.c`

Replaces `HwrDisplayAttributes()` function from the HAL API for Palm OS 4.0.

HALDisplayGetPalette Function

Purpose This function gets the contents of the Color Lookup Table (CLUT), or palette, for the LCD controller hardware, which is stored in the HAL. Contents are returned in an RGB table.

Prototype
```
Err HALDisplayGetPalette(Int16 iStartIndex, UInt16
    iNumEntries, RGBColorType *oTable)
```

Parameters

→*iStartIndex*

If between 0 and 255, `iStartIndex` is the index of a CLUT entry. The *first* entry in `iTable` gets the value of that CLUT entry. The `iTable` values then get CLUT values sequentially. That is: CLUT[iStartIndex+n] sets the value of oTable[n], for n from 0 to iNumEntries-1.

Note that the `index` field of the `oTable` entries is set to the index of the CLUT entry.

→*iNumEntries*

Number of entries in the table.

←*oTable*

An array of `RGBColorType` structures. Size of array is equal to `iNumEntries`.
If `iStartIndex=-1`, the `index` field of the `RBGColorType` structures acts as an in- parameter. Otherwise, the `index` field is an out-parameter.

See "RGBColorType" on page 30 for more information about the `RGBColorType` data type.

Returns `0`

If no error. Otherwise, returns `kHALDispErrOutOfRange`, if `oTable` contains an entry whose index exceeds the maximum index for the current screen depth.

Comments `CTLDisplay.c`

Replaces `HwrDisplayPalette()` function from the HAL API for Palm OS 4.0.

See Also `HALDisplaySetPalette`

HALDisplaySetAttributes Function

Purpose This function returns the various attributes of the LCD controlling hardware.

Prototype

```
Err HALDisplayGetAttributes(UInt16 iAttribute,
   UInt32 iValue)
```

Parameters

→*`iAttribute`*
The attribute to set. Only the attributes marked as `Both` in the table are allowed to be set.

→*`iValue`*
Value to assign.

See "Display Attribute Constants" on page 23 for more information about display attributes and their values. In that table, only the attributes marked as Both are allowed to be set.

Returns

`0`
If no error.

Comments `CTLDisplay.c`

Replaces `HwrDisplayAttributes()` function from the HAL API for Palm OS 4.0.

HALDisplaySetPalette Function

Purpose This function sets the contents of the Color Lookup Table (CLUT), or palette, for the LCD controller hardware.

Prototype

```
Err HALDisplaySetPalette(Int16 iStartIndex, UInt16
   iNumEntries, RGBColorType *iTable)
```

Parameters

→*`iStartIndex`*
If between 0 and 255, `iStartIndex` is the index of a CLUT entry. The *first* entry in `iTable` sets that CLUT entry. The `iTable` values then set CLUT values sequentially. That is:
iTable[n] sets the value of CLUT[iStartIndex+n] ,
for n from 0 to iNumEntries-1.

If `iStartIndex` equals -1, then the `index` field of each `iTable` entry is used as an index into the CLUT. The CLUT entry receives the value of the `iTable` entry. Starts with the first entry in `iTable` and proceeds sequentially through the `iTable` entries.

→*`iNumEntries`*
Number of entries to set.

→*`iTable`*
An array of `RGBColorType` structures. Size of array must equal `iNumEntries`.

See "RGBColorType" on page 30 for more information about the `RGBColorType` data type.

Returns `0`
If no error. Otherwise, returns `kHALDispErrOutOfRange`, if `iTable` contains an entry whose index exceeds the maximum index for the current screen depth.

Comments `CTLDisplay.c`

Replaces `HwrDisplayPalette()` function from the HAL API for Palm OS 4.0.

See Also `HALDisplayGetAttributes`

HALDisplayDoze Function

Purpose This function handles the LCD when the device is being put into doze mode, rather than into full sleep mode. In doze mode, the processor is stopped, but the LCD still displays the last view of the user interface and refreshes the screen. This is a power-saving mode.

Prototype `Err HALDisplayDoze(Boolean doze)`

Parameters →*`doze`*
`True` if device is in wake mode and you want to put it in doze mode.
`False` if device is in sleep mode and you want to wake up the LCD controller without displaying anything on screen. This is a special situation. See Comments below.

Returns `0`
If no error.

Comments `CTLDisplay.c`

Called by the auto-lock alarm in System Manager

A special situation arises when the following conditions exist:

- the `doze` parameter is `false`.
- the `kHALDispMemAccessOK` display attribute is false. (See "Display Attribute Constants" on page 23.)
- the device is in sleep mode.
- the auto-lock alarm has been triggered by the passage of the specified time period.

Before locking the device, the alarm handler notifies Palm OS to process events such as updating the current time and the battery gauge. Both activities draw to the screen, which causes a small difficulty: since the device is asleep, the LCD controller is disabled. And since the `kHALDispMemAccessOK` attribute is false, this attempt to draw to the screen causes a bus error.

To solve this situation, the alarm handler calls `HALDisplayDoze()` with `doze` set to `false`. This wakes the LCD controller but not the LCD itself. In this way, the Palm OS can draw to the video memory without causing a crash.

You may wonder why the alarm handler doesn't just call `HALDisplayWake()`. While this would work, it would also cause the LCD to flash on and display. Since the device was asleep when the auto-lock alarm was triggered, having the screen turn on is not desirable behavior.

HALDisplayLock Function

Purpose This function reduces screen flicker and ensures smooth screen updates.This function locks the screen, returning the address of an offscreen buffer to which the blitter writes.

Prototype

```
void* HALDisplayLock(ScrGlobalsType* scrGlobalsP,
   Boolean* oAlreadyLocked, UInt32 size)
```

Parameters

→*scrGlobalsP*
Pointer to screen manager globals.

←*oAlreadyLocked*
Returns `true` if the screen is already locked.

→*size*
Size in bytes of the offscreen screen buffer.

Returns Returns a pointer to the new offscreen buffer, if allocated. Or returns NULL if the offscreen buffer is not allocated.

Comments `CTLDisplay.c`

This function "locks" the display screen of the Palm OS device by moving the existing frame buffer to a different address and then returning the address of a new, offscreen buffer. The driver continues to display the moved buffer while the blitter writes to the offscreen buffer. When the screen is "unlocked," the contents of the offscreen buffer are reflected onscreen.

To support screen locking, your Palm OS device must have enough VRAM for two frame buffers. If screen locking is not supported `HALDisplayLock()` should return NULL.

The controller supported by the sample DAL creates an offscreen buffer in VRAM.

The screen lock count represents the number of times that `HALDisplayLock()` has been called. The screen must be unlocked as many times as it was locked in order to actually update the device display screen.

When an application locks the screen, the window manager calls the screen manager which calls the display driver: `WinScreenLock()` calls `HALScreenLock()`, which calls `HALDisplayLock()`.

HALDisplaySleep Function

Purpose This function turns off the LCD. Called during the process of putting the device to sleep, this function handles the display appropriately.

Prototype
```
Err HALDisplaySleep(Boolean untilReset, Boolean
    emergency)
```

Parameters →*untilReset*
`True` indicates that once the device has been put to sleep, it will remain so until a hard reset. Usually, this parameter is `true` only when `emergency` parameter is true, too.

→*emergency*
`True` indicates that the device is being shut down faster than normal, usually in response to a low battery interrupt. The emergency shutdown happens immediately to save user data that is in the storage heap. Normal cleanup is not performed.

Returns `0`
If no error.

Comments `CTLDisplay.c`

This function must handle the case of an emergency shutdown and the case of a sleep state that can only be wakened by hard reset. For normal shutdowns and sleep states, the function is called with `false` for both parameters.

This function is called by the HAL, not by the Palm OS. See `LBC_HALPower.c`

HALDisplayUnlock Function

Purpose This function works in concert with `HALDisplayLock()` to reduce screen flicker and ensure smooth screen updates. It "unlocks" the display screen by replacing the buffer that the driver is currently displaying with the offscreen "virtual" buffer.

Prototype `Err HALDisplayUnlock (ScrGlobalsType* scrGlobalsP)`

Parameters →*scrGlobalsP*
Pointer to screen manager globals.

Returns `0`
If no error.
Returns `HALDispErrUnlockErr` if the current and new screen addresses are the same.

Comments `CTLDisplay.c`

This function sets the base address of the driver's current buffer to the base address of the offscreen frame buffer that was established by an earlier call to `HALDisplayLock()`. Consequently, the contents of the offscreen buffer are displayed onscreen.

If the DAL uses the system heap to allocate its screen buffer, it gets deallocated here. The controller supported by the sample DAL allocates its screen buffer in VRAM.

When an application unlocks the screen, the window manager calls the screen manager which calls the display driver: `WinScreenUnlock()` calls `HALScreenUnlock()`, which calls `HALDisplayUnlock()`.

HALDisplayWake Function

Purpose This function wakes up the LCD.

Prototype `Err HALDisplayWake(void)`

Parameters None.

Returns `0`

If no error.

Comments `CTLBoot.c`

Replaces `HwrDisplayWake()` function from the HAL API for Palm OS 4.0.

See Also `EvtResetAutoOffTimer`

6

Initialization

This chapter describes the API functions of the HAL that deal with the initialization process. They are described in alphabetical sequence. Some of the routines deal with general initialization. Others deal with the virtual memory map.

Setting up and initializing the virtual memory map is one of the primary goals of the initialization process. For details about the memory map and how it gets set up during the boot process, see "Memory Management" in *DAL Customization Guide.* You will also find information about memory locations of actual SB-relative globals and of low memory globals.

For details on the order in which different pieces of hardware are inialialized, see "Boot Sequence" in *DAL Customization Guide.*

HwrPostDebugInit Function

Purpose This routine is called after the debugger is installed. It installs the `HwrLowBatteryHandler()` to prevent race conditions.

Prototype `void HwrPostDebugInit(void)`

Parameters None.

Returns None.

Comments `LBC_ROMHiHardware.c`

This function is called by `InitStage2()`.

Install the `HwrLowBatteryHandler` in case battery interrupts are handled this early, as it can prevent race conditions.

See Also `InitStage2`

HwrPreDebugInit Function

Purpose This function performs necessary initialization of hardware before initializing the debugger.

Prototype `void HwrPreDebugInit(UInt32 cardHeaderAddr)`

Parameters `cardHeaderAddr` Address of card header structure.

Returns None.

Comments `LBC_ROMHardware.c`

This function is called by `InitStage1() in ROMBoot.c`.

See Also `HALDisplayDrawBootScreen`
`InitStage1`

HwrPreRAMInit Function

Purpose This function performs necessary initialization of hardware before RAM can be used for the dynamic heap and the storage heap. It sets up the final virtual memory map before the kernel starts. It initializes certain hardware devices—specifically, those that have not already been initialized by the `Reset_A` routine and those that will not be initialized by driver-specific initialization function calls later in the boot sequence.

Prototype `void HwrPreRAMInit(void)`

Parameters None.

Returns None.

Comments `LBC_ROMHardware.c`

This function is called by the C_Entry point in `ROMBoot.c`. Extremely important initialization of the memory regions takes place in this function. However, the `HwrPreRAMInit` assumes that `Reset_A` code has already performed a preliminary initialization of the RAM. Examine in-line comments in `Reset_A.s` and `LBC_ROMHardware.c` for details about what happens at each step of these two initialization stages.

See Also Reset_A

InitStage1 Function

Purpose Main initialization code for booting the Palm OS device. All basic setup starts here. This includes locating the big ROM, initializing system and hardware globals, locating the system shared library, initializing the debugger, and initializing the kernel. It can also enter the small ROM debugger.

Prototype

```
void InitStage1(UInt16 hardResetOrDebug,
   UInt32 cardHeaderAddr)
```

Parameters

→*hardResetOrDebug flag*
-1 = hard reset requested
1 = drop into debugger (for flashing)
0 = boot normally

→*cardHeaderAddr*
Address of CardHeader structure.

Returns None.

Comments `ROMBoot.c`

The `CardHeaderType` is defined in `MemoryPrv.h`.

InitStage2 Function

Purpose Secondary initialization code for Palm OS—which initializes the BigROM. This routine calls the post debugger initialization function. Then it calls functions to initialize interrupts, digitizer/pen subsystem, key subsystem, battery, and time manager.

Prototype `void InitStage2(Boolean hardReset)`

Parameters

`hardReset`
True, if the Palm OS device is undergoing a hard reset. All storage heaps will be wiped, in addition to the dynamic heap.
False, if the Palm OS device is undergoing a soft reset. Storage heaps will be preserved, while the dynamic heap alone is wiped.

Returns None

Comments `ROMBootStage2.c`

Calls the DAL API function `HwrPostDebugInit()`.

Reset_A Function

Purpose This routine is written in ARM assembly language. It is the very first code executed upon the first device boot-up, after a hard reset, or after a soft reset. It performs all the initialization required before branching to C_Entry in `ROMBoot.c`, which is the main C-language system code. The `Reset_A` code defines the ENTRY point, initializes the stack pointers for each mode, copies Read-Only and Read/Write data from ROM to RAM, and zero-initializes the ZI data areas used by the C code. It also initializes the DRAM configuration/control registers to permit the MMU translation tables to be loaded into RAM. The first initialization of the virtual memory map also takes place in `Reset_A`. The `HwrPreRAMInit` function, which will refine the virtual memory map, assumes that `Reset_A` has already done this preliminary work.

The `Reset_A` code is defined in the `Reset_A.s` file.

If you are modifying this routine, see an important note in the Comments section of `HwrPreRAMInit`.

See Also `HwrPreRAMInit`

7

Interrupt Handling

This chapter describes the API functions of the HAL that deal with the interrupts. They are described in alphabetical sequence. These routines deal with hardware interrupts. If you wish to learn specifically about software interrupts, however, see "“Writing a Software Interrupt Handler” in *DAL Customization Guide*.

Interrupt Handling Data Structures

InterruptAllStatus

The `InterruptAllStatus` data type indicates whether the interrupts as a group are disabled or enabled. The constants defined in this enumeration are used by HALInterruptAllSetStatus. The definition for `InterruptAllStatus` is found in `HALInterrupts.h`.

```
enum InterruptAllStatusTag {
  InterruptAllStatusDisabled,
  InterruptAllStatusEnabled,
};

typedef Enum16 InterruptAllStatus;
```

IRQState

The `IRQState` data type is used to indicate the state of a given interrupt: whether it is disabled or enabled. The constants defined in this enumerations are used by HALInterruptSetState. The definition for `IRQState` is found in `LBC_Interrupt.h`.

```
typedef enum {
  IRQDisabled,
  IRQEnabled,
}IRQState;
```

Interrupt Functions

HALInterruptAllSetStatus Function

Purpose Enables or disables all interrupts at the same time.

Prototype
```
InterruptAllStatus
   HALInterruptAllSetStatus(InterruptAllStatus
   newStatus)
```

Parameters → *newStatus*
State of interrupts considered as a group. This parameter indicates the state to set.

Returns The old state (all interrupts enabled or disabled).

Comments `LBC_Interrupts.c`

HALInterruptGetHandler Function

Purpose Returns the handler and the data it uses for the given interrupt number.

Prototype
```
void HALInterruptGetHandler(UInt32 irqNum,
   void **handlerP, void **handlerArgP)
```

Parameters → *irqNum*
Interrupt number

← *handlerP*
Returns pointer to handler

← *handlerArgP*
Returns pointer to handler data

Returns `0.`
If no error.

Comments `LBC_Interrupts.c`

This routine replaces the direct assignment of interrupt vectors in for Palm OS 4.0, which was the method used on Motorola 68K processors.

HALInterruptSetHandler Function

Purpose Assigns an interrupt handler to a given interrupt.

Prototype
```
void HALInterruptSetHandler(UInt32 irqNum,
   void *handlerP, void *handlerArgP)
```

Parameters

→*irqNum*
Interrupt number

→*handlerP*
Pointer to handler

→*handlerArgP*
Pointer to handler data

Returns None.

Comments `LBC_Interrupts.c`

This routines replaces the direct assignment of interrupt vectors in for Palm OS 4.0, which runs on Motorola 68K processors.

HALInterruptSetState Function

Purpose Enables or disables an interrupt line.

Prototype
```
IRQState HALInterruptSetState(UInt32 irqNum,
   IRQState newState)
```

Parameters

→*irqNum*
Interrupt number

→*newState*
State to set

Returns The old interrupt line state.

Comments `LBC_Interrupts.c`

HwrInterruptsInit Function

Purpose This routine is called from `InitStage2`. It initializes the system interrupts and installs the system timer procedure.

Prototype
```
void HwrInterruptsInit(void);
```

Parameters None.

Returns None.

Comments LBC_Interrupts.c

8

Keyboard Support

This chapter describes the API functions of the HAL that deal with the keyboard. They are described in alphabetical sequence.

The HAL implementation needs to enqueue a key-down event on the first key-down. After the key is first down, key-repeat actions should enqueue a repeating event at a rate set by `HALKeySetRates`, until the key is detected to be up. Once the key is up, the implementation needs to enqueue a key-up event.

For more information about the Key Manager, see the *Palm OS Programmer's Companion* and the *Palm OS Programmer's API Reference*.

For information about key events and how they are handled, see "Hardware Events" on page 61.

Keyboard Support Masks

Key Mask

`HALKeySetMask` and `HALKeyGetState` use a UInt32 mask. Each bit in the mask corresponds to a particular hard key or hardware feature (such as the antenna) on the device. The mask is defined in the include files `HALKey.h` and `CmnKeyTypes.h`

Bit	Value
0	Power Key
1	Page-up
2	Page-down
3	App #1
4	App #2

Bit	Value
5	App #3
6	App #4
7	Cradle
8	Antenna
9	Contrast

Keyboard Support Data Structures

None applicable.

Keyboard Support Functions

HALKeyGetRates Function

Purpose Retrieve current key repeat rate.

Prototype

```
Err HALKeyGetRates(UInt16* oInitDelay, UInt16*
    oPeriod, Boolean* oQueueAhead)
```

Parameters

←*oInitDelay*
The initial delay in milliseconds for an auto-repeat event.

←*oPeriod*
The auto-repeat rate specified as the period in milliseconds.

←*oQueueAhead*
If true, auto-repeating keeps queuing up key events if the queue has keys in it. If false, auto-repeat does not enqueue keys unless the queue is already empty.

Returns

`0`
If no error.

Comments `LBC_KeyMgr.c`

If you pass in NULL for any of the above parameters, that parameter will not return a value.

Replaces `KeyRates()` function from the HAL API for Palm OS 4.0.

See Also HALKeySetRates

HALKeyGetState Function

Purpose Get UInt32 with bits set for each key that is currently depressed.

Prototype `UInt32 HALKeyGetState(void)`

Parameters `None.`

See "Key Mask" on page 49 to discover what the bits in the key mask mean.

Returns Returns the current key state.

Comments `LBC_KeyMgr.c`

Replaces `KeyCurrentState()` function from the HAL API for Palm OS 4.0.

Some systems cannot provide current state information; so it may be necessary for the DAL to cache the state information internally.

This API is intended for use only with the basic keys defined in `HALKey.h` (i.e., power, scroll up, scroll down, and the application keys).

See Also HALKeySetMask

HALKeyResetDoubleTap Function

Purpose Resets the double-tap counter so that the next tap can be considered as the first one of a double.

Prototype `Err HALKeyResetDoubleTap(void)`

Parameters None.

Returns `0`

If no error.

Comments `LBC_KeyMgr.c`

Replaces `KeyResetDoubleTap()` function from the HAL API for Palm OS 4.0.

This function is called by `SysEvtMgr()` when it detects a pen event to reset the key manager's double-tap detection. It is called by `EvtEnqueuePenPoint()`.

See Also HALKeySetMask

HALKeySetMask Function

Purpose This function sets the key mask--a bitfield that specifies which hardware keys are allowed to generate key events and which are not. Use this to turn off key events for one or more hardware keys.

Prototype `UInt32 HALKeySetMask(UInt32 keyMask)`

Parameters →*keyMask*

The mask bitmap with bits set to 1 for active keys that must generate key events. Bits set to 0 for masked keys that should not generate key events.

See "Key Mask" on page 49 to discover what the bits in the key mask mean.

Returns Returns the old mask value if called from the big ROM. Returns zero if called from the small ROM.

Comments `LBC_KeyMgr.c`

Replaces `KeySetMask()` function from the HAL API for Palm OS 4.0.

This API is intended for use only with the basic keys defined in HALKey.h (i.e., power, scroll up, scroll down, and the application keys.)

See Also HALKeyGetState

HALKeySetRates Function

Purpose Set key repeat rate.

Prototype

```
Err HALKeySetRates(UInt16 initDelay,
   UInt16 Period, Boolean queueAhead)
```

Parameters →*initDelay*

The initial delay in milliseconds for an auto-repeat event.

→*period*
The auto-repeat rate specified as the period in milliseconds.

→*queueAhead*
If `true`, auto-repeating keeps queuing up key events if the queue has keys in it. If `false`, auto-repeat does not enqueue keys unless the queue is already empty.

Returns `0`
If no error.

Comments `LBC_KeyMgr.c`

Replaces `KeyRates()` function from the HAL API for Palm OS 4.0.

See Also HALKeyGetRates

9

Power States

This chapter describes the API functions of the HAL that deal with the different power states. They are described in alphabetical sequence.

Power management of the Palm OS® is the responsibility of DAL. The Palm OS only needs a way to set the auto-sleep time-out value, and be notified before the device really goes to sleep.

The Palm OS operates in one of the three following states: run, doze, and sleep. The specific implementation of the doze and sleep states is platform dependent. For example, on existing Palm OS hardware when all Palm OS tasks block, the DAL can enter the doze mode by shutting down the CPU clock, but leaving the peripherals on. Any interrupt can wake up the CPU from doze mode. On platforms where there is less direct access to the hardware, the DAL may provide advice to the underlying hardware layer.

If the Palm OS has enabled auto-sleep, and the DAL has direct access to the hardware, the DAL can power off the device after a period of inactivity. After entering the sleep state, only certain interrupts (some hardware keys, alarms etc.) can wake up the device. As with the doze state, the DAL can provide sleep advice to the underlying hardware layer on systems with no direct access to hardware.

Wake and Sleep modes are handled in the HAL, while the Doze mode is entered by the kernel when there are no running tasks. Documentation for Doze is therefore in the Kernel Documentation.

Power States Data Structures

None applicable.

Power States Functions

HALPowerGetAutoOffEvtTime Function

Purpose Gets the scheduled time at which the system will automatically go to sleep. Time is identified in absolute terms—that is, the number of milliseconds since the last system reset.

Prototype `UInt32 HALPowerGetAutoOffEvtTime(void)`

Parameters None.

Returns Returns the scheduled time in milliseconds since the last reset.

Comments `LBC_HALPower.c`

Absolute time refers to system time in milliseconds, as returned by function HALTimeGetSystemTime.

HALPowerGetAutoOffSeconds Function

Purpose Gets the system's auto-off timeout.

Prototype `UInt16 HALGetAutoOffSeconds(void)`

Parameters None.

Returns Returns the auto-off timeout, which can be set through `HALSetAutoOffSeconds`.

Comments `LBC_HALPower.c`

HALPowerSetAutoOffEvtTime Function

Purpose Schedules the system to automatically go to sleep at the time specified. Time is identified in absolute terms—that is, the number of milliseconds since the last system reset.

Prototype `Void HALPowerGetAutoOffEvtTime(UInt32 iEvtTime)`

Parameters →*`iEvtTime`*
Number of milliseconds since reset.

Returns `0`

Comments `LBC_HALPower.c`

Absolute time is referred to as *system time* in milliseconds, as returned by function HALTimeGetSystemTime.

HALPowerSetAutoOffSeconds Function

Purpose Sets the system's auto-off timeout.

Prototype `Err HALSetAutoOffSeconds(UInt16 iSeconds)`

Parameters `iSeconds`

Number of seconds of user inactivity that must elapse before auto-off feature puts the device to sleep.

Returns None.

Comments `LBC_HALPower.c`

HALPowerSleepReady Function

Purpose Palm OS calls this function to indicate it is ready to sleep. If the DAL decides to really put the device to sleep when this function is called, this function will not return until the device is awakened. If the sleep process is aborted by the DAL because of unexpected events, returning from this function will also let the Palm OS experience a "faked" awakened process.

`Err HALPowerSleepReady(void)`

Parameters None.

Returns `0`

If no error.

Comments `LBC_HALPower.c`

Returning from this function means "awake and running" to the Palm OS. After returning from this function, the Palm OS will perform additional "wake-up" processing.

10

Miscellaneous Functions

This chapter describes the API functions of the HAL dealing with various functions that do not fit in one of the other chapters.

Random Seed

Initializing the seed for random calculation needs access to hardware facilities. It is the responsibility of the DAL to provide this access.

HALRandomInitializeSeed Function

Purpose Initialize seed to be used by system or user-defined random generators.

Prototype `Err HALRandomInitializeSeed(UInt32* randomSeed)`

Parameters → *randomSeed*
New random seed.

Returns `0`
If no error.

Comments `HALRandom.c`

According to the type of DAL implementation, this function may typically call hardware timers, read random pieces of memory (as in previous Palm OS versions), or call an underlying API.

Boot Functions

HALSetInitStage Function

This function initializes a global variable that keeps track of what stage the Palm OS initialization process has reached. That variable is set each time a manager is launched. It is used to find out when threads launched by PalmOS can go on.

Prototype `Err HALSetInitStage(UInt32 uiValue)`

Parameters →*`uiValue`*

Milestone indicating the state of managers at tat function call time.

Returns `0`

If no error.

Comments `LBC_ROMHardware.c`

The valid milestones can be found in `HALReset.h`.

PalmOSMain Function

This function is the only one called by the DAL to initialize and start PalmOS.

Prototype `Err PalmOSMain(Boolean hardReset)`

Parameters →*`hardReset`*

Boolean value that controls memory initialization. If a hard reset is requested, storage heaps will be reformatted (i.e., wiped).

Returns `0`

If no error.

Comments `PalmOSMain.c`

None.

Hardware Events

Palm OS is an event driven system. Hardware events such as Pen and Key events are added to the Palm OS queues using callbacks. The following PalmOS functions will be registered respectively to enqueue Pen and miscellaneous (including Key) events. Note that *key* events are used even for things that are not related to the keyboard; most key events are in fact virtual characters.

- `ErrEvtEnqueuePenPoint(PointType* penPoint)`
- `ErrEvtEnqueueKey(WChar ascii, UInt16 keycode, UInt16 modifiers)`

These callbacks are registered by PalmOS at boot time (in the `SysUILaunch` function) using HALEventRegisterCallback.

HALEventPost Function

Explicitly enqueues an event. Called from the Palm OS.

Prototype

```
Err HALEventPost(HALEventIDType iEventID,
    const HALEventDataType* iEventData)
```

Parameters

→*`iEventID`*
Event of the event to be posted.

→*`iEventData`*
Pointer to the event being enqueued.

Returns

`0`
If no error.

Comments

`LBC_HALEvent.c`

The Palm OS calls this function to enqueue software-generated events into the key queue. This function has to call Palm OS callbacks without being re-entered. Callbacks for custom events defined by other DAL implementers and registered with `HALEventRegisterCallback` should be handled by this function.

NOTE: Currently this function only supports posting key events.

HALEventRegisterCallback Function

This function is used by the Palm OS to register itself for all events, including non-UI events.

Prototype
```
Err HALEventRegisterCallback(HALEventIDType
    iEventID, HALEventCallBackPtrType iCallBack,
    HALEventCallBackPtrType *oPrvCallBack)
```

Parameters

→*iEventID*
ID of the event to be registered. (See the description of available events in HalEvent.h)

→*iCallBack*
A pointer to the callback function.

←*oPrvCallBack*
The old callback address, if any.

Returns

`0`
If no error.

Comments

`LBC_ROMHardware.c`

It is assumed that the function pointer has been processed to include the necessary code to switch contexts between the caller and the callback, or that the caller will handle the necessary switch.

Reset

HALReset Function

Resets and restarts the Palm OS environment.

Prototype
```
void HALReset(Boolean hardReset)
```

Parameters

`hard Reset`
True to perform a hard reset. False to perform a soft reset.

Returns None.

Comments

`LBC_ROMHardware.c`

A *hard reset* initializes the storage heaps as well as the dynamic heap. Thus, non-volatile data is lost. A *soft reset* merely wipes the dynamic heap (i.e., stack, globals, and dynamic allocation area), preserving storage heaps.

Miscellaneous Functions

HALGetHwrMiscFlags Function

Returns the system hardware flags that describe available features on the current device.

Prototype `UInt16 HALGetHwrMiscFlags(void)`

Parameters None.

Returns System hardware flags, as described in `HwrMiscFlags.h` and set by `HwrIdentifyFeatures`.

Comments `LBC_ROMHiHardware.c`

None.

HALGetHwrMiscFlagsExt Function

Returns the extended system hardware flags that describe available features on the current device.

Prototype `UInt32 HALGetHwrMiscFlagsExt(void)`

Parameters None.

Returns Extended system hardware flags, as described in `HwrMiscFlags.h`.

Comments `LBC_ROMHiHardware.c`

None.

HALGetHwrWakeUp Function

Retrieves the system wake-up state.

Prototype `UInt16 HALGetHwrWakeUp(void)`

Parameters None.

Returns System wake-up state flags.

Comments `LBC_ROMHiHardware.c`

Wake-up milestones can be found in `Hardware.h`

HALGetROMToken Function

This function retrieves a ROM token from the burned list of tokens.

Prototype

```
Err HALGetROMToken(UInt32 tokenRequested,
   UInt8** dataPP, UInt16* sizeP)
```

Parameters

→*tokenRequested*
Address of 4-byte ROM token that you are requesting.

↔ *dataPP*
If in-parameter was not NULL, out-parameter is data in ROM token. Pass in NULL, if data not needed.

↔ *sizeP*
If in-parameter was not NULL, out-parameter is size of data in ROM token. Pass in NULL, if size not needed.

Returns `Returns one of the following values:`

`errNone`
Success--token found.

`kHALErrorTkhTokenNotFound`
Token not found.

`kHALErrorTknTokenInvalid`
Invalid token.

Comments `HALTokens.c`

HALOEMGetCompanyID Function

Returns Company ID of the HAL manufacturer.

Prototype `UInt32 HALOEMGetCompanyID(void)`

Parameters None.

Returns HAL maker ID.

Comments `LBC_ROMHiHardware.c`

HALOEMGetDeviceID Function

Returns Device ID of the device on which this HAL is running.

Prototype `UInt32 HALOEMGetDeviceID(void)`

Parameters None.

Returns Device ID.

Comments `LBC_ROMHiHardware.c`

HALOEMGetHALID Function

Returns HAL ID of this HAL.

Prototype `UInt32 HALOEMGetHALID(void)`

Parameters None.

Returns HAL ID.

Comments `LBC_ROMHiHardware.c`

HALProcessorID Function

Returns Processor ID of the platform this HAL runs on.

Prototype `UInt32 HALOEMGetProcessorID(void)`

Parameters None.

Returns Processor ID.

Comments `CTLProcessor.c`

See `CmnFtrNums.h` for a listing of processor values.

HALSetHwrMiscFlags Function

Allows modification of the system hardware flags.

Prototype `Void HALSetHwrMiscFlags(UInt16 newValue)`

Parameters →*`newValue`*
The new value of the flag word.

Returns `None.`

Comments `LBC_ROMHiHardware.c`

Flags are described in `HwrMiscFlags.h`

11

Memory

This chapter describes the API functions of the HAL that deal with memory initialization and protection. They are described in alphabetical sequence. For conceptual explanations of memory, see "Memory Management" in *DAL Customization Guide*.

Memory Data Structures

HALMemoryMap

The `HALMemoryMap` data type contains a pointer to an array of `HALMemoryRegionType` structures. The definition for `HALMemoryMap` is found in `HALMemory.h`.

```
typedef struct HALMemoryMapTag {
  UInt8 numRegions,
  const HALMemoryRegionType* regions }
HALMemoryMapType;
```

Field Descriptions

`numRegions`	Number of memory regions.
`regions`	Pointer to array of `HALMemoryRegionType`. Size of array is `numRegions`.

HALMemoryRegionType

The `HALMemoryRegionType` data type is used as a memory region descriptor, storing information such as the type, base address and size of a given memory region. The definition for `HALMemoryRegionType` is found in `HALMemory.h`.

```
typedef struct HALMemoryRegionTag {
```

```
  HALMemoryType type,
  void* baseAddress,
  UInt32 size,
} HALMemoryRegionType;
```

Field Descriptions

`type`	Kind of memory region. Value is one of enum constants defined by `HALMemoryType`.
`baseAddress`	Pointer to address where the memory region starts.
`size`	Size of the memory region in bytes.

HALMemoryType

The `HALMemoryType` data type is used to indicate the kind of memory region. The constants defined in `HALMemoryTag` correspond to the four kinds of memory. The definition for `HALMemoryType` is found in `HALMemory.h`.

```
enum HALMemoryTag {
  kROM,
  kVolatileRAM,
  kNonVolatileRAM
  ksmallROM
};
typedef Enum8 HALMemoryType;
```

Value Descriptions

`kROM`	Big ROM.
`kVolatileRAM`	Dynamic heap.
`kNonVolatileRAM`	Storage heap.
`ksmallROM`	Small ROM. Used in flash upgrade. For description of flash upgrading, see *Building a ROM Upgrade Tool*.

Example

Consider the following example from the globals initialization portion of the Sample DAL. In `HALGlobals.h`, an array of memory region structures is declared as a field of the `HALGlobalsType` structure. The following code snippet shows the array and memory map declarations:

```
. . .
HALMemoryRegionType HALMemoryRegions[NUM_HAL_MEMORY_REGIONS];
HALMemoryMapType HALMemoryMap;
. . .
```

The `HALMemoryRegions` array is then accessed through the `GHALMemoryRegions` macro and initialized as shown in this snippet from `LBC_ROMHardware.c`:

```
GHALMemoryRegions[0].type = kROM;
GHALMemoryRegions[0].baseAddress = romStartAddr;// this is
                                                //not a fixed address!
GHALMemoryRegions[0].size = hwrBigROMSize;
GHALMemoryRegions[1].type = kVolatileRAM;
GHALMemoryRegions[1].baseAddress = GHwrDynamicHeapBase;
GHALMemoryRegions[1].size = GHwrDynamicHeapSize;
GHALMemoryRegions[2].type = kNonVolatileRAM;
GHALMemoryRegions[2].baseAddress = hwrVirtualStorageHeapBase;
GHALMemoryRegions[2].size = GHwrStorageHeapSize;

// Currenty the SmallROM region is simply mapped along
// with the BigROM.  This region can be mapped to
// any location, but this is the simplest way to do it
// and allows the SmallROM region to be dynamically remapped
// along with the BigROM when the widebin is loaded in
// RAM for debugging.
GHALMemoryRegions[3].type = kSmallROM;
GHALMemoryRegions[3].baseAddress = romStartAddr - hwrBigROMFlashOffset;
GHALMemoryRegions[3].size = hwrBigROMFlashOffset;

GHALMemoryMap.numRegions = NUM_HAL_MEMORY_REGIONS;
GHALMemoryMap.regions = GHALMemoryRegions;
```

Memory Map Functions

HALMemoryGetMemoryMap Function

Purpose The Palm OS calls this function once at boot time to get a map of all the available memory regions, including: the number of regions, their kind, base address, and size. This routine is also called by the flash upgrade utility to determine location of the small ROM. Once the small ROM is located, the flash driver in the small ROM can be run from RAM.

Prototype
```
const HALMemoryMapType
    *HALMemoryGetMemoryMap(void)
```

Parameters None.

Returns Returns pointer to a constant structure which defines the entire memory map of the device (all regions). Don't modify the returned memory since it might be in ROM!

Comments `LBC_ROMHiHardware.c`

Storage of the region descriptors (i.e., `HALMemoryRegionType`) is maintained by the DAL, which must set the `HALMemoryRegionType` values somewhere. This can be done in `HwrPreRAMInit`. The values must be set before `HALMemoryGetMemoryMap()` is first called.

Memory Protection Functions

Palm OS protects its data storage heaps by disabling writing to corresponding areas of physical memory most of the time. Modifications to the storage heap can be done only by a few system functions that enable writing, do the modifications, and then immediately disable writing again. Palm OS increases the protection by disabling task switching while memory protection is off.

HAL must implement functions for disabling/enabling such protections. Note that two levels of protection are required: hardware memory write protection, and a mechanism to prevent other tasks from corrupting the memory while a task has disabled the protection. This API affects the entire storage heap.

HALMemoryGetStorageAreaProtectionState Function

Returns the current protection state.

Prototype

```
Boolean HALMemoryGetStorageAreaProtectionState(
    void)
```

Parameters None.

Returns Returns `true` if memory is not protected (writing is enabled). Returns `false` if memory is protected (writing is disabled).

Comments `LBC_ROMHardware.c`

None.

HALMemorySetStorageAreaProtectionState Function

Enable or disable writing to the storage heaps.

Prototype

```
Err HALMemorySetStorageAreaProtectionState(
    Boolean enableWrites)
```

Parameters →*`enableWrites`*

If false, protects memory.

Returns None.

Comments `LBC_ROMHardware.c`

This function does not return the previous state of memory protection. For that functionality, use HALMemoryGetStorageAreaProtectionState.

12

Real Time Clock Support

This chapter describes the API functions of the HAL that deal with the Real Time Clock (RTC). They are described in alphabetical sequence.

Real Time Clock Support Data Structures

None applicable.

Real Time Clock Support Functions

HALTimeGetAlarm Function

Purpose This function gets the current alarm setting in seconds since 1 January 1904. This function is reserved for use by the Alarm Manager.

Prototype `Err HALTimeGetAlarm(UInt32 *AlarmSeconds)`

Parameters →*`AlarmSeconds`*

The current alarm setting in seconds since 1/1/1904.

Returns `0`

If no error.

Comments `CTLTimeMgr.c`

Replaces `TimGetAlarm()` function from the HAL API for Palm OS 4.0.

This function is called by the Alarm Manager.

Since `TimSetAlarm()` is reserved for Alarm Manager use, it does not require its own semaphore to assure data integrity. The Alarm Manager uses a semaphore for this purpose.

This function first masks the Real Time Clock (RTC) interrupt, then gets the number of alarm seconds, and finally unmasks the RTC interrupt.

See Also `HALTimeSetAlarm`

HALTimeGetSeconds Function

Purpose This function returns the number of seconds since 1 January 1904.

Prototype `UInt32 HALTimeGetSeconds(void)`

Parameters None.

Returns Returns number of seconds since 1/1/1904.

Comments `CTLTimeMgr.c`

Replaces `TimGetSeconds()` function from the HAL API for Palm OS 4.0.

This function is called by many HAL and Palm OS functions.

See Also `HALTimeGetSystemTime`
`HALTimeGetSystemTimerInterval`
`HALTimeSetSeconds`

HALTimeGetSystemTime Function

Purpose Returns the number of milliseconds since boot time.

Prototype `UInt32 HALTimeGetSystemTime(void)`

Parameters None.

Returns Returns the system time counter value snapshot.

Comments `CTLTimeMgr.c`

HALTimeGetSystemTimerInterval Function

Purpose Returns the effective number of system ticks per second.

Prototype `UInt32 HALTimeGetSystemTimerInterval(void)`

Parameters None.

Returns Returns number of milliseconds between system time interrupts.

Comments `CTLTimeMgr.c`

See Also `SysTicksPerSecond`

HALTimeSetAlarm Function

Sets an alarm in seconds since 1 January 1904. It is reserved for use by the Alarm Manager.

Prototype `Err HALTimeSetAlarm(UInt32 iAlarmSeconds)`

Parameters →*`iAlarmSeconds`*

Alarm in seconds since 1 January 1904, or `0` to cancel the current alarm.

Returns `0.`

If no error.

Comments `CTLTimeMgr.c`

Replaces `TimSetAlarm()` function from the HAL API for Palm OS 4.0.

This function is called by the Alarm Manager, and by `AlmEnableNotification()`, `AlmCancelAll()`, `AlmDisplayAlarm()`, and `PrvSetNextAlarm()`.

Since `TimSetAlarm()` is reserved for the Alarm Manager use, it does not require its own semaphore to assure data integrity. The Alarm Manager uses a semaphore for this purpose.

This function first masks the Real Time Clock (RTC) interrupt, then calculates the number of alarm seconds, and finally unmasks the RTC interrupt.

Palm OS permits only one RTC alarm to exist at any one time. When calling this function, any existing alarm set by previous calls to `HALTimeSetAlarm()` is overridden.

The DAL implementation needs to enqueue and return an RTC (alarm) event to the Palm OS when an alarm set by `HALTimeSetAlarm()` expires. If the device is in sleep mode when the alarm expires, it is the responsibility of the DAL to wake up the device before posting the RTC event.

See Also `AlmEnableNotification`
`AlmCancelAll`
`AlmDisplayAlarm`
`PrvSetNextAlarm`

HALTimeSetSeconds Function

Purpose This function sets the time of the device.

Prototype `Err TimSetSeconds(UInt32 iSeconds)`

Parameters → *`iSeconds`*
Number of seconds since 1 January 1904.

Returns `0`
If no error.

Comments `CTLTimeMgr.c`

Replaces `TimSetSeconds()` function from the HAL API for Palm OS 4.0.

This function is called by many HAL and system functions.

13

Serial Drivers

This chapter describes the API functions of the HAL that deal with drivers for serial communications. There are two main sections. The Virtual Driver section starts with "Virtual Driver Data Structures" on page 78. This section applies to all ports that emulate serial communications. The USB Driver section is much shorter and starts with "USB Data Structures" on page 95. This section contains material specific to a virtual driver that controls a USB port.

For details on writing a driver, consult *Serial Communications Driver Design Guide*.

Brief Overview of Virtual Drivers

Palm OS 4.0 introduced the concept of a *virtual driver*, which has the following characteristics:

- It can manage multiple serial ports at once.
- It can transmit and receive data in blocks or bytes.
- It may control a different kind of port, such as IR or USB, that is emulating a serial connection.

The serial port abstraction is defined to be the line between the Serial Manager and its drivers. This means that the serial drivers are entirely implemented in the HAL.

These drivers are shared libraries, and many of them can exist simultaneously. So, the API presented below is not really the HAL interface to serial communications. It is more the API of the functions that a serial driver must implement.

When the DAL runs using the same runtime model as the Palm OS, a driver implementation is really straightforward because it is easy to call inside the driver from Palm OS, as well as let the driver call back to the Palm OS.

But, if the DAL runtime is different than that of Palm OS, special care must be taken. Indeed, in that case, the driver-exported functions will be initially called using the Palm OS runtime, but the driver itself will have been built to use the DAL runtime. Some wrapper capable of switching runtimes will therefore be needed in the driver.

As it is the case for all shared libraries, a driver will return its entry point function pointer when it is loaded. This entry point is called `HALSerialEntrypoint` and is described below. The Serial Manager will call this entry point and get back the function pointers that will let it use the port. Prototypes for these functions are described below. Understand that these are only templates and that each serial driver must implement its own set.

In Palm OS, the following functions are implemented using serial drivers:

- RS232
- IrCOMM
- Serial emulation over USB.

A driver for a USB port will provide almost all the same functions as a driver for regular serial port. In the current implementation, only one special function is needed specifically for USB support.

Virtual Driver Data Structures

DrvEntryOpCodeEnum Enum

Purpose The serial manager can call this entry point with three different opcodes. The driver must return the requested data in the space pointed to by oData. The three opcodes are defined as:

Constants

```
typedef enum DrvrEntryOpCodeTag
   DrvrEntryOpCodeEnum;
enum DrvrEntryOpCodeTag
{
   drvrEntryGetPortCount,
   drvrEntryGetDrvrFuncts,
   drvrEntryGetPortFtrsNEntries
};
```

Fields `Value Descriptions`

`drvrEntryGetPortCount`	Used to request that the driver return the number of ports it handles. In this case, `oData` points to a `UInt16`.
`drvrEntryGetDrvrFuncts`	Used to request that the driver initialize `oData` with pointers to its functions. This is used when a client opens the port. In this case, `oData` points to a `VdrvAPIType`.
`drvrEntryGetPortFtrsNEntries`	Used to request information on the port. `HalSerialEntryPoint` must be called with this opcode, once for each port. In this case, `oData` points to a `DrvrInfoType`. The `portNumber` field of the `DrvrInfoType` must be set to the index of the currently requested port (first port is number 0). The driver will set the remaining fields in the `DrvrInfoType.`

DrvInfoType Struct

Purpose The `DrvInfoType` is a structure that provides `HALSerialEntryPoint` a place to return various characteristics of a given port. The first field specifies the port in question.

A pointer to this type is passed back by `HalSerialEntryPoint` when the `drvEntryGetPortFtrsNEntries` opcode is passed in.

Prototype

```
typedef struct DrvrInfoTag DrvrInfoType
struct DrvrInfoTag
{
   UInt8 portNumber;
   UInt32 drvrID;
   UInt32 drvrVersion;
   UInt32 maxBaudRate;
   UInt32 portFlags;
   const Char *portDesc;
Uint32 dbCreator;
};
```

Fields `Field Descriptions`

`portNumber`	Must not be changed. On entry to this function, it will contain the port number (zero based) for which the serial manager currently requests information.
`drvrID`	Will be set to the port Id the application will need to specify to select a driver. Must be set to something unique for each port amongst all drivers (e.g. COM1, COM2, etc.). Possibly that it will not be the same as `dbCreator`.
`drvrVersion`	Must be set to the `kDrvrVersion` constant, which represents the SDK current version.
`maxBaudRate`	Field will be set to a numeric value equal to the maximum speed this port can handle. For drivers using a transport for which setting a speed has no sense (IR, TCP routing, etc.), just specify 230400L.
`portFlags`	Specifies the port's features and capabilities (see `SerialVdrvr.h`). This is where the driver says if it handles an RS232 port, an IR transceiver, a USB connection etc. See "Port Flags Constants" on page 81.

`portDesc`	Can be set to zero. In this case, it will be initialized later by the serial manager to point to the port name, whose string must be stored in a resource. The resource type is `'tSTL'` (string list) and resource id is 1000. The port number serves as an index in the string list to get the port name. If you choose to use this resource, the port list has to be fixed and statically ordered. If the port name cannot be known in advance, the driver can allocate a memory block to store the port name and store a pointer to this block in the portDesc field. This block must be assigned to the system (OwnerID=0) and will become its property.
`dbCreator`	Will be set to the driver's code database creator. This is the PRC containing the driver code.

Port Flags Constants

The `portFlags` field of the `DrvInfoTag` structure mentioned above contains a bitfield value. Flags can be turned off or on, using constants defined in `SerialVdrv.h`. Serial driver programming deals mainly with the following constants.

Constant	Value	Description
`portBkgndModeSupported`	0x00000002	Denotes that this port can be used for background mode.Background mode support is implied on physical drivers.
`portRS232Capable`	0x00000004	Driver supports communications through an RS-232 port. Usually OR'ed with `portCradlePort`. May also be OR'ed with `portIRDACapable`.

Constant	Value	Description
`portIRDACapable`	0x00000008	Driver supports communications via an IR port, using IRDA mode. Usually OR'ed with `portCradlePort. May also be OR'ed with portRS232Capable`.
`portCradlePort`	0x00000010	Driver assumes that the port is the device's cradle port. Usually, this value is OR'ed with one or more other values from the table.
`portExternalPort`	0x00000020	Denotes this SerialHW's port is external or on a memory card.
`portModemPort`	0x00000040	Denotes this SerialHW communicates with a modem.
`portCncMgrVisible`	0x00000080	Denotes this serial port's name is to be displayed in the Connection panel.
portPrivateUse	0x00001000	Set if this driver is for special software and NOT general applications in the system.
`portUSBCapable`	0x00000200	Driver supports a USB connection. Usually OR'ed with `portCradlePort`. In the default drivers shipped to licensees, USB support has its own driver, and the value is not OR'ed with `port232Capable`.

DrvRcvQTag Struct

This structure contains pointers to callback functions provided by the serial manager for accessing its receive queue. These callbacks are invoked by the driver in order to write incoming data to the serial manager. See prototypes of the function pointers directly following the table.

Prototype

```
typedef struct DrvrRcvQTag {
    void *rcvQ;
    WriteByteProcPtr qWriteByte;
    WriteBlockProcPtr qWriteBlock;
    GetSizeProcPtr qGetSize;
    GetSpaceProcPtr qGetSpace;
  } DrvrRcvQType;

  typedef DrvrRcvQType *DrvrRcvQPtr;
```

Fields

Field	Descriptions
`void *rcvQ;`	Pointer to the serial manager's receive queue, where your driver will write incoming data.
`qWriteByte`	Function pointer to function that writes one incoming byte to the serial manager's receive queue.
`qWriteBlock`	Function pointer to function that writes one block of incoming bytes to the serial manager's receive queue from the receive queue.
`qGetSize`	Function pointer to function returning the total size, in bytes, of the serial manager's receive queue.
`qGetSpace`	Function pointer to function returning the available space, in bytes, of the serial manager's receive queue.

Receive Queue Callbacks

The serial manager provides callbacks that give access to its receive queue. The prototypes of these function pointers appear below. For detailed descriptions, see "Serial Manager Queue Functions" in the *Palm OS Programmer's API Reference*.

Prototypes of Callback Functions

```
typedef Err(*WriteByteProcPtr)(void *theQ, UInt8
theByte, UInt16 lineErrs);
```

```
typedef Err(*WriteBlockProcPtr)(void *theQ,
UInt8 *bufP, UInt16 size, UInt16 lineErrs);

typedef UInt32(*GetSizeProcPtr)(void *theQ);

typedef UInt32(*GetSpaceProcPtr)(void *theQ);
```

Parameters to Callbacks

-> *theQ	Pointer to the serial manager's receive queue.
-> theByte	One byte of incoming data that driver is writing to the serial manager's receive queue.
-> *bufP	Pointer to a buffer of incoming data that driver is writing to the serial manager's receive queue.
-> size	Size of the buffer pointed to by `*bufP`.
-> lineErrs	Any serial line errors should be reported here.

VdrvAPIType

The `VdrvAPIType` is a structure whose members are function pointers. The functions in question are defined in the driver.

A pointer to this type is passed back by `HalSerialEntryPoint` when the `drvrEntryGetDrvrFuncts` opcode is passed in.

```
typedef struct VdrvAPITag VdrvAPIType
struct VdrvAPITag
{
   VdrvOpenProcPtr HALSerialOpen;
   VdrvCloseProcPtr HALSerialClose;
   VdrvControlProcPtr HALSerialControl;
   VdrvStatusProcPtr HALSerialStatus;
   VdrvWriteProcPtr HALSerialWrite;
   VdrvControlCustomProcPtr
   HALSerialControlCustom;
```

```
};
```

VdrvConfigType

This structure is used to hold configuration information about the connection. Values are set by the call to the `HALSerialOpen()` routine—which may have a different name, since it is always called via a function pointer. For instance, the routine is sometimes named `VdrvOpen`().

```
typedef struct VdrvConfigType {
   UInt32 baud;
   UInt32 drvrId;
   UInt32 function;
   MemPtr drvrDataP;
   UInt16 drvrDataSize;
} VdrvConfigType;
typedef VdrvConfigType *VdrvConfigPtr;
```

Fields

`UInt32 baud;`	Baud rate at which to connect.
`UInt32 drvrId`	Creator ID of the driver that is handling the port that was opened.
`UInt32 function`	Function Id of the connection. Used only by USB connections. Identifies which application on the device opened the port. This information is used by the desktop's USB driver to locate the corresponding application on the desktop.
`MemPtr drvrDataP`	Pointer to driver-specific data. Needed by blueTooth; ignored otherwise.
`UInt16 drvrDataSize`	Size of the driver specific data block. Needed by blueTooth, ignored otherwise.

VdrvDataPtr

This is a generic pointer, used as an out-parameter from the `HALSerialOpen()` routine. Private data particular to the opened port is stored at this location.

```
typedef void *VdrvDataPtr;
```

Virtual Driver Functions

HALSerialClose Function

Purpose This function is called to close a previously opened port.

Prototype `Err HALSerialClose (VdrvDataPtr idrvrData )`

Parameters Port information related to the port to close

Returns `0`

If no error.

Comments `lbcSerialDriver.c`

The driver must deallocate everything that it allocated in `HALSerialOpen`.

HALSerialControl Function

Purpose This function is used to access low level driver functions, like setting the baud rate.

Prototype

```
Err HALSerialControl(VdrvDataPtr idrvrData,
    VdrvCtlOpCodeEnum iControlCode,
    void* ioControlData, UInt16* ioControlDataLen
    );
```

Parameters `idrvrData`

The driver’s private data pointer.

`iControlCode`

A control function opCode. One of the opCodes listed in the VdrvCtlOpCodeEnum type.

`ioControlData`
: A pointer to data for the specified control function. Depending on the function, this can be a pointer to input data, or a pointer to space for returned data.

`ioControlDataLen`
: A pointer to the length of data being passed in or out.

Returns `0`
: If no error.

Comments `lbcSerialDriver.c`

This function needs to support the opcodes listed in the `VdrvCtlOpCodeEnum` type. If an opcode is unsupported, the call must return the `serErrNotSupported` error code for that opcode. The following table lists the constants defined by the `VdrvCtlOpCodeEnum` type and describes the corresponding `ioControlDataP` and `ioControlDataLenP` parameters. Some control codes just perform an action but do not input nor output any data.

Constant	Description
`vdrvOpCodeSetBaudRate`	`IoControlData` points to a `UInt32` specifying the new desired baud rate.
`vdrvOpCodeSetSettingsFlags`	`IoControlData` points to a `UInt32` specifying the new desired settings.
`vdrvOpCodeSetCtsTimeout`	`IoControlData` points to a `UInt32` specifying the CTS timeout in ticks. This timeout is the amount of time the function `HALSerialWrite` should wait when CTS is held down before returning a serErrTimeout.
`vdrvOpCodeClearErr`	This opcode instructs the driver that the serial manager has cleared its sticky hardware error, and that the driver should now clear them in the hardware if necessary.
`vdrvOpCodeSetSleepMode`	The device is now going to sleep, so the driver should suspend the underlying hardware.

Constant	Description
vdrvOpCodeSetWakeupMode	The device is now waking up. The driver should restore what it suspended when the port was put to sleep.
vdrvOpCodeTxFIFOCount	`IoControlData` points to a `UInt32` into which the driver must store the number of bytes that are in the send queue waiting to be sent out.
vdrvOpCodeStartBreak	Start a break signal.
vdrvOpCodeStopBreak	Stop a break signal
vdrvOpCodeStartLoopback	Start loopback mode.
vdrvOpCodeStopLoopback	Stop loopback mode.
vdrvOpCodeFlushTxFIFO	Kill all data waiting in the send queue
vdrvOpCodeFlushRxFIFO	Kill all data that has arrived in the device but that has not been read yet by the driver input handler (ISP, read thread or other).
vdrvOpCodeSendBufferedData	Waits until all data that is still in the send queue be sent out. If it is not possible to send all the data without blocking for more than the current CTS timeout, return serErrTimeout.
vdrvOpCodeGetOptTransmitSize	`IoControlData` points to a `UInt32` where the driver must store the optimum buffer size for sending data. If the driver does not buffer transmit data, it must return 0.
vdrvOpCodeGetRcvTheshold	`IoControlData` points to a `UInt32` where the driver must store the number of free bytes that must be available in the receive queue before it will store more data in it. The serial manager will use this to calculate the largest block that is guaranteed to be delivered by the driver before it holds the input.

Constant	Description
`vdrvOpCodeSetWakeupMode`	The device is now waking up. The driver should restore what it suspended when the port was put to sleep.
`vdrvOpCodeTxFIFOCount`	`IoControlData` points to a `UInt32` into which the driver must store the number of bytes that are in the send queue waiting to be sent out.
`vdrvOpCodeStartBreak`	Start a break signal.
`vdrvOpCodeStopBreak`	Stop a break signal
`vdrvOpCodeStartLoopback`	Start loopback mode.
`vdrvOpCodeStopLoopback`	Stop loopback mode.
`vdrvOpCodeFlushTxFIFO`	Kill all data waiting in the send queue
`vdrvOpCodeFlushRxFIFO`	Kill all data that has arrived in the device but that has not been read yet by the driver input handler (ISP, read thread or other).
`vdrvOpCodeSendBufferedData`	Waits until all data that is still in the send queue be sent out. If it is not possible to send all the data without blocking for more than the current CTS timeout, return serErrTimeout.
`vdrvOpCodeGetOptTransmitSize`	`IoControlData` points to a `UInt32` where the driver must store the optimum buffer size for sending data. If the driver does not buffer transmit data, it must return 0.
`vdrvOpCodeGetRcvTheshold`	`IoControlData` points to a `UInt32` where the driver must store the number of free bytes that must be available in the receive queue before it will store more data in it. The serial manager will use this to calculate the largest block that is guaranteed to be delivered by the driver before it holds the input.

Constant	Description
`vdrvOpCodeNotifyBytesReadFromQ`	This notifies the driver that some space has been made in the serial receive queue. If the driver receive handler was suspended waiting for space to push more bytes in the receive queue, it can resume now.
`vdrvOpCodeSetDTRAsserted`	`IoControlData` points to a Boolean specifying if the DTR line must be set to a high (if true) or low (if false) level.
`vdrvOpCodeGetDTRAsserted`	`IoControlData` points to a Boolean where the driver must store true if the DTR line is currently at a high level, or false otherwise.
`vdrvOpCodeWaitForConfiguration`	The serial manager uses this opcode before it calls `SrmSend` or `SrmReceive`, in order to let the driver finish any lengthy initialization it would have started in `HALSerialOpen`.
`vdrvOpCodeGetUSBDeviceDescriptor`	Query driver for device descriptor for USB.
`vdrvOpCodeGetUSBConfigDescriptor`	Query driver for configuration descriptor for USB.

HALSerialControlCustom Function

Purpose This function enables a driver to make sure that a custom control opcode is really supported by this driver (and that it is not just the same id as a custom opcode from a different driver). This is accomplished by adding the driver creator to the prototype.

Purpose

```
Err HALSerialControlCustom (VdrvDataPtr idrvrData,
    UInt16 iControlCode, UInt32 iCreator,
    void* ioControlData,
    UInt16* ioControlDataLen);
```

Parameters

`idrvrData`
The driver's private data.

`iControlCode`
A control function opcode.

`iCreator`
The application asking for this opcode also passes the expected driver creator here. The driver must make sure this call can be handled appropriately.

`ioControlData`
A pointer to data for the specified control function. Depending on the function, this can be input data, or a pointer to space for returned data.

`ioControlDataLen`
A pointer to the length of data being passed in or out.

Returns `0`
If no error.

Compatibility `lbcSerialDriver.c`

There are no opcodes currently defined, as each driver will define their own. Most drivers do not implement specific opcodes and should just return a `serErrNotSupported`.

HALSerialEntryPoint Function

Purpose This function is the one returned by `RALLoadModule` when a driver is loaded (therefore, this function must be exported from its module). All drivers are loaded at boot time when the serial manager initializes.

Prototype

```
Err HalSerialEntryPoint
    (DrvrEntryOpCodeEnum iOpCode, void *oData)
```

Parameters `iOpCode`
Specify the function and the data type asked to the entry point.

`oData`
A pointer to a `UInt16`, `DrvrInfoType`, or `VdrvAPIType` structure, depending on the opCode. This pointer is used to return the data.

Returns Returns 0 if call succeeded, otherwise the return value is non-zero (the port will not show up at all in Palm OS).

Comments `lbcSerialDriver.c`

If some hardware or software must be available to use a port, this function can check for these requirements and return an error any of them are not available. In this case, the port will simply be ignored and hidden. Since the needed resources, however, may not be checkable at boot time, an alternative is to return zero anyway. Then you can wait until some application opens the port and can report an error if the needed resource is still not available.

NOTE: The functions described in this section can have whatever names you assign them in the serial driver. They are always called using a function pointer. Elsewhere in the documentation, for instance, these may be called VdrvOpen(), VdrvClose(), etc.

HALSerialOpen Function

Purpose This function initializes the port so that it is ready to send and receive data. This can mean for example initializing a physical UART or create an IrCOMM instance.

Prototype

```
Err HALSerialOpen (VdrvDataPtr oDrvrData,
   VdrvConfigPtr iConfig, DrvrRcvQPtr iRcvQ );
```

Parameters

`oDrvrData`
: A pointer to a `VdrvDataType` field where the driver can store some private port-dependant data. This data will be passed back to the driver with every `HALSerialXXX` call.

`iConfig`
: The structure pointed by `iConfig` contains the initial baud rate. drvrId specifies which port the client wants to open. Parameters `drvrData` and `drvrDataSize` will be used only by drivers that need special data passed in from the client opening the port (e.g. the Bluetooth driver passes some channel information here). Drivers that don't have this need should just ignore these fields.
The yield related parameters are passed to the driver so that it can eventually forward them to an underlying driver that it uses these parameters.

`iRcvQ`

The `iRcvQ` parameter contains a pointer to a structure containing function pointers. Each one points to a callback provided by the serial manager so that drivers can access its receive queue. The driver invokes these callbacks when it has incoming data to send to the serial manager.

NOTE: The serial driver provides no API routine to perform synchronous receive. Instead, the driver directly accesses the serial manager's receive queue by means of the `iRcvQ` field.

Returns `0`

If no error.

Comments `lbcSerialDriver.c`

Type definitions used by this function include:

```
typedef Err (*WriteByteProcPtr)(void *theQ, UInt8
   theByte,UInt16 lineErrs);
typedef Err (*WriteBlockProcPtr)(void *theQ, UInt8 *bufP,
   UInt16 size, UInt16 lineErrs);
typedef UInt32 (*GetSizeProcPtr)(void *theQ);
typedef UInt32 (*GetSpaceProcPtr)(void *theQ);
typedef void (*SignalCheckPtr)(void *theQ, UInt16 lineErrs);

typedef struct DrvrRcvQTag DrvrRcvQType;
struct DrvrRcvQTag
{
   void*rcvQ;
   WriteByteProcPtrqWriteByte;
   WriteBlockProcPtrqWriteBlock;
   GetSizeProcPtrqGetSize;
   GetSpaceProcPtrqGetSpace;
};

typedef struct VdrvConfigTag VdrvConfigType;
struct VdrvConfigTag
{
   UInt32baud;
   UInt32drvrId;
   UInt32function;
   MemPtrdrvrDataP;
   UInt16drvrDataSize;
   SrmYieldPortProcPtryieldPortCallBackP;
   UInt32yieldPortRefCon;
```

```
};

typedef void* VdrvDataType;
```

HALSerialStatus Function

Purpose Returns the port status. The information returned includes the state of the following lines: RTS, CTS, and DSR. It also indicates if a break condition has been detected.

Prototype

```
Err HALSerialStatus(VdrvDataType idrvrData,
   UInt16* oStatus);
```

Parameters

`idrvrData`
A pointer to the driver's private data.

`oStatus`
The port status is returned in this parameter

Returns

`0`
If no error.

Comments `lbcSerialDriver.c.`

The port status is made from the mask values defined in `DrvrStatusEnum`.

```
typedef enum DrvrStatusTag DrvrStatusEnum;
enum DrvrStatusTag
{
   drvrStatusCtsOn= 0x0001,
   drvrStatusRtsOn= 0x0002,
   drvrStatusDsrOn= 0x0004,
   drvrStatusBreakAsserted= 0x0020
};
```

HALSerialWrite Function

Purpose This function sends data out through the port.

Prototype

```
UInt32 HALSerialWrite( VdrvDataPtr idrvrData,
Const void* iBuffer,
UInt32 ioLength,
Err *errP )
```

Parameters

`idrvrData`
The driver's private data.

`iBuffer`
A pointer to the buffer containing the data to be written to the virtual device.

`ioLength`
The number of bytes to be written is passed into the function. The number of bytes actually written is passed back.

`errP`
Pointer to error code returned

Returns Returns the number of bytes actually written

Comments `lbcSerialDriver.c`

This function will block until all the bytes are written, or an error occurs. The only possible error is a `serErrTimeout`, which means the CTS line was held down for more than the current CTS timeout.

Even if the implementation allows for asynchronous sending, the driver should still block until data has all been sent out, because the caller is expecting this behavior.

USB Data Structures

UsbDeviceRequestType Struct

Purpose This structure is used by USBRequestGetExtConnectionInfo.

Prototype

```
typedef struct {
   UInt8   bmRequestType;
   UInt8   bRequest;
   UInt16   wValue;
```

```
    UInt16   wIndex;
    UInt16   wLength;
} UsbDeviceRequestType,* UsbDeviceRequestPtr;
```

Fields

`bmRequestType`

`bRequest`

`wValue`

`wIndex`

`wLength`

USB Driver Functions

UsbConnect Function

Purpose This function initializes the USB hardware and starts the enumeration.

Prototype `void UsbConnect (void);`

Parameters None.

Returns None.

Comments `ctlUsbIO.c`

UsbDisconnect Function

Purpose This function de-enumerates.

Prototype `void UsbDisconnect (void);`

Parameters None.

Returns None.

Comments `ctlUsbIO.c`

UsbHwrInit Function

Purpose This routine sets up the USB chip. It is called by `HwrPostDebugInit`.

Prototype
```
void UsbHwrInit (Boolean reset);
```

Parameters → *`reset`*

If `true`, function initializes USB chip and puts it in low power.

If `false`, function performs the minimum initialization of the USB chip and puts it in low power.

Returns None.

Comments `ctlUsbHwrInit.c`

UsbRequestGetExtConnectionInfo Function

Purpose This routine handles the vendor-defined GetExtConnectionInfo request. This request is sent by the host during enumeration to get information about the nature of the connections. This function supersedes `UsbRequestGetConnectionInfo`.

Prototype
```
static void UsbRequestGetExtConnectionInfo
    (VdrvDataPtr idrvrData,
    UsbDeviceRequestPtr requestP);
```

Parameters → *`idrvrData`*

The driver's private data.

→ *`requestP`*

Vendor Device Request

Returns None.

Comments `ctlUsbRequest.c`

The `UsbDeviceRequestPtr` is defined in `ctlUsbRequest.h`.

14

Screen

Drawing functionality is provided by four software components that work in concert: the window manager, the screen manager, the blitter, and the display driver. The window manager is part of the Palm OS® and cannot be modified. The other three components are part of the HAL and can be modified by licensees. For a discussion of how the components divide up the work of producing graphic display, see "Display Architecture" in *Display Driver Design Guide*.

This chapter describes the screen manager and the blitter. Data structures and constants applying to both are presented first. Then Screen manager functions are presented, followed by blitter functions.

The bulk of the material describes the blitter, which is the low-level and behind-the-scenes component responsible for drawing graphic primitives. It generates the drawing primitives, initializes the blitting state variables, and applies logical operations on the source data while writing pixel values to the destination bitmap. This destination bitmap can be either the LCD display, or an offscreen window.

The destination bitmap is part of something called a *canvas*, which is the DAL equivalent of a *graphics port*. A canvas also contains information about the drawing state, such as object colors (e.g. foreground, background, text), transfer modes, and patterns. Drawing functions are context free, i.e. all drawing functions require that a pointer to a `CanvasType` structure be passed as a parameter.

Each primitive drawing operation can lead to one of two possibilities:

- Drawing to the device's display (that is, the video memory)
- Drawing to a Palm OS format off-screen buffer

Note that the blitter is the graphics component closest to the video display hardware. It is, therefore, the component that licensees will modify to incorporate support for hardware video acceleration.

All these issues are discussed in more detail in *Display Driver Design Guide*.

Blitter Supports High Density

The blitter code that ships with the current DAL provides native support for double-density and one-and-a-half density, in addition to single-density.

NOTE: A previous release of the PDK (Palm OS 5 release 5.0) had glue functions to provide single-density support. They are not part of the current release. Since the glue functions and the blitter functions have very similar names, be sure you are editing the correct functions. For instance, the previous release had a glue function named `HALDrawLine`, whereas the current release has a blitter routine named `HAL_Drawline`.

Intermediate Buffer Not Needed

In the previous release(s) of Palm OS PDK, you had to create an intermediate buffer for unusual combinations of processor and video controller endianness. The standard combination in Palm OS 5 is a little-endian ARM processor and a little-pixel-endian video controller. (For more details about little-pixel-endian graphics, refer to the section “Pixel Arrangements” in the *Display Driver Design Guide*). The current version of the DAL has built-in support for most combinations of processor and video controller. Consequently, you will probably not need to create your own intermediate buffer.

Screen Data Structures

These are the data structures used by the screen manager and the blitter.

AbsRectType

```
typedef struct AbsRectType {
  Coord left;
  Coord top;
  Coord right;
  Coord bottom;
} AbsRectType;
```

Field Descriptions

left	Left coordinate of the rectangle.
top	Top coordinate of the rectangle.
right	Right coordinate of the rectangle.
bottom	Bottom coordinate of the rectangle.

BitmapCompressionType

The `BitmapCompressionType` enum specifies possible bitmap compression types. These are the possible values for the `compressionType` field of `BitmapTypeV3` and `BltBitmapType` data structures.

```
typedef enum {
   BitmapCompressionTypeScanLine = 0,
   BitmapCompressionTypeRLE,
   BitmapCompressionTypePackBits,
   BitmapCompressionTypeEnd,
   BitmapCompressionTypeBest = 0x64,
   BitmapCompressionTypeNone = 0xFF
   } BitmapCompressionType;
```

Value Descriptions

BitmapCompressionTypeScanLine	Use scan line compression. Scan line compression is compatible with Palm OS® 2.0 and higher.
BitmapCompressionTypeRLE	Use RLE compression. RLE compression is supported in Palm OS 3.5 and higher.
BitmapCompressionTypePackBits	Use PackBits compression.PackBits compression is supported in Palm OS 4.0 and higher.
BitmapCompressionTypeEnd	For internal use only.
BitmapCompressionTypeBest	For internal use only.
BitmapCompressionTypeNone	No compression is used. This value should only be used as an argument to BmpCompress.

BitmapFlagsType

The `BitmapFlagsType` bitmap defines the `flags` field of `BitmapTypeV3`. Defined in `CmnBitmapTypes.h`, the `BitmapFlagsType` specifies the attributes of a bitmap. (In this context, a *bitmap* is a graphic image.)

```
typedef struct BitmapFlagsType {
UInt16 compressed:1;
UInt16 hasColorTable:1;
UInt16 hasTransparency:1;
UInt16 indirect:1;
UInt16 forScreen:1;
UInt16 direct Color:1;
Uint16 indirectColorTable:1;
UInt16 noDither:1;
UInt16 reserved:8;
} BitmapFlagsType;
```

Field Descriptions

compressed	If `true`, the bitmap is compressed and the compression type field specifies the compression used. If `false`, the bitmap is uncompressed. The `BmpCompress` function sets this field.
hasColorTable	If `true`, the bitmap has its own color table. If `false`, the bitmap uses the system color table. You specify whether the bitmap has its own color table when you create the bitmap.
hasTransparency	If `true`, the OS will not draw pixels that have a value equal to the `transparentIndex`. If `false`, the bitmap has no transparency value. You specify the transparent color when you create the bitmap, using the Palm OS Constructor user interface or calling the Palm OS API `BmpSetTransparentValue`.

indirect	If `true`, the address to the bitmap's data is stored where the bitmap itself would normally be stored. The actual bitmap data is stored elsewhere. If `false`, the bitmap data is stored directly following the bitmap header or directly following the bitmap's color table if it has one. Never set this flag.
forScreen	If `true`, the bitmap is the bitmap for the display (screen) window. Never set this flag.
directColor	If true, bitmap contains direct RGB data, not palette indexes.
indirectColorTable	If true, a pointer to the color table for the bitmap is stored in place of the color table. This allows bitmaps to share color tables, thus saving memory.
noDither	If true, blitter does not dither bitmaps when imaging.
reserved	Reserved for future use.

BitmapTypeV3

The `BitmapTypeV3` type is used extensively by the window manager APIs of the Palm OS, by private drawing utilities in the DAL, and by such DAL APIs as `HALScreenInit`. Although usually declared as `BitmapType` in function prototypes, the actual structure is of type `BitmapTypeV3`. For more information, see BitmapType in the *Palm OS Programmer's API Reference*.

NOTE: This definition corresponds to the 'Tbmp' and 'tAIB' resource types.

```
typedef struct BitmapTypeV3
{
  Int16            width;
  Int16            height;
  UInt16           rowBytes;
  BitmapFlagsType  flags;
  UInt8            pixelSize;
  UInt8            version;
//version 3 fields
  UInt8           size;
  UInt8            pixelFormat;
  UInt8            unused;
  UInt8            compressionType;
 UInt16           density;
 UInt32           transparentValue
 UInt32           nextBitmapOffset

  // if (flags.hasColorTable)
  //    {
  //    if (flags.indirectColorTable)
  //       ColorTableType *colorTableP
  //    else
  //       ColorTableType  colorTable;
  //    }
  // if (flags.indirect)
  //    void*  bitsP;
  // else
  //    UInt8  bits[];
}
BitmapTypeV3;
```

Field Descriptions

width	The width of the bitmap in pixels. You specify this value when you create the bitmap.
height	The height of the bitmap in pixels. You specify this value when you create the bitmap.
rowBytes	The number of bytes stored for each row of the bitmap where height is the number of rows.

flags	The bitmap's attributes. See BitmapFlagsType.
pixelSize	The bits per pixel. Currently supported pixel depths are 1-, 2-, 4-, and 8-bit index color and 16-bit direct color. You specify this value when you create the bitmap.
version	Version of the bitmap. This is version 3.
size	Size of this structure in bytes. (0x16)
pixelFormat	Format of the pixel data. See `PixelFormatType`.
unused	Reserved for future use.
compressionType	See `BitmapCompressionType`
density	Used by blitter to scale bitmaps.
transparentValue	The index or RGB value of the transparent color.
nextBitmapOffset	Byte offset to next bitmap in bitmap family.
colorTableP	Pointer to color table.
colorTable	Color table, which could have 0 entries. Value is 2 bytes long.
bitsP	Pointer to actual bits
bits	Actual bits.

BltBitmapType

The double-density window manager uses the `BltBitmapType` to communicate bitmap information to the blitter. This structure is designed for compatibility with the Palm OS BitmapType data structure. The `BltBitmapType` is very similar to a `BitmapTypeV3` (types 3 bitmaps). The notable difference is that the `BltBitmapType` requires three fields that are optional on `BitmapTypeV3`. These fields are `colorTableP`, `bitmapDataP` and `compressedSize`.

```
typedef struct BltBitmapType
   {
   // version 3 BitmapType
   Int16 width;
   Int16 height;
   UInt16 rowBytes;
   BitmapFlagsTyp flags;
   UInt8 pixelSize;
   UInt8 version;
   UInt8 size;
   UInt8 pixelFormat;
   UInt8 unused;
   UInt8 compressionType;
   UInt16 density;
   UInt32 transparentValue;
   UInt32 nextBitmapOffset;
   // blitter fields
   ColorTableType* colorTableP;
   void* bitmapDataP;
   UInt32 compressedSize;
   }
BltBitmapType;
```

Field Descriptions

width	Width of bitmap image in pixels.
height	Height of bitmap image in pixels.
rowBytes	Number of bytes it takes to store a single row of pixels: `width * bitdepth / 8`
flags	See BitmapFlagsType.
pixelSize	Bits per pixel.
version	Data structure version 3.
size	Size of this structure in bytes (0x16).
pixelFormat	Format of the pixel data. See `pixelFormatType` in `CmnBitmapTypes.h`.
unused	Reserved for future use.

compressionType	See BitmapCompressionType .
density	Used by the blitter to scale bitmaps.
transparentValue	The index or RGB value of the transparent color, filling UInt32.
nextBitmapOffset	Byte offset to next bitmap in bitmap family.
colorTableP	Pointer to the bitmap's color table. NULL is possible value.
bitmapDataP	Pointer to bits of the image or the bits of the compressed image.
compressedSize	Size of compressed data.

CanvasType

This double-density version of the `CanvasType` is used by the blitter. It is the first argument in all calls to the blitter. This structure encapsulates the context of the window state needed by the blitter, while avoiding dependence on the `WindowType` data structure. It is defined as follows:

```
typedef struct CanvasType {
  RectangleType     clippingRect;
  DrawStateType*    drawStateP;
  BltBitmapType*       bitmapP;
} CanvasType;
```

Field Descriptions

clippingRect	Clipping rectangle of the `DrawWindow`. This represents the clipping bounds of the destination bitmap.

drawStateP	Pointer to the `DrawWindow`'s graphic state (refer to the `DrawStateType` structure defined below). Defined in `Window.h`, it contains the transfer mode, color, pattern, and font definitions
bitmapP	A pointer to the destination bitmap. Note that the destination bitmap is a `BltBitmapType`.

The new blitter version of CanvasType does not have a `viewOrigin` field. Instead, all drawing coordinates are relative to the upper-left corner of the bitmap.

ColorTableType

Used by `HALDraw_FindIndexes`, and `HALScreenGetColorTable`.

```
typedef struct ColorTableType
{
    // high bits (numEntries > 256) reserved
   UInt16         numEntries;
   UInt16 reserved;
   RGBColorType   entry[];
}
ColorTableType;
```

Field Descriptions

numEntries	Number of entries in table.
entry[]	Variable-sized array of colors (`0` to `numEntries-1`).

CustomPatternType

```
typedef UInt8 CustomPatternType [8];
//8x8 1-bit deep pattern
```

DrawStateType

Used by HALDraw_GetPixel.

```
typedef struct DrawStateType
{
   WinDrawOperation    transferMode;
   PatternType         pattern;
   UnderlineModeType   underlineMode;
   FontID              fontId;
   FontPtr             font;
   CustomPatternType   patternData;

   IndexedColorType    foreColor;
   IndexedColorType    backColor;
   IndexedColorType    textColor;
   UInt8               reserved;

   RGBColorType       foreColorRGB;
   RGBColorType       backColorRGB;
   RGBColorType       textColorRGB;

   UInt16             coordinateSystem;
   DrawStateFlagsType flags;
   Fixed              scale;
   Fixed              ntvToActiveScale;
   Fixed              stdToActiveScale;
   Fixed              activeToStdScale;
}
```

Field Descriptions

transferMode	The current transfer mode for color drawing.
pattern	The ID of the current pattern. If set to `customPattern`, the `patternData` field contains the actual pattern.
underlineMode	The ID of the current underline mode.
fontId	The ID of the current font.
font	A pointer to the current font.

patternData	The current pattern being used by the `WinFill` functions if `pattern` is `customPattern`.

The following are only valid for indexed color bitmaps:

foreColor	Index of the current color used for the foreground.
backColor	Index of the current color used for the background.
textColor	Index of the current color used for text.
reserved	Reserved for future use.

The following are only valid for direct color bitmaps:

foreColorRGB	RGB value of the current color used for the foreground. Only valid for Palm OS 4.0 and 4.1.
backColorRGB	RGB value of the current color used for the background. Only valid for Palm OS 4.0 and 4.1.
textColorRGB	RGB value of the current color used for text. Only valid for Palm OS 4.0 and 4.1.

These fields are used when drawing most graphic primitives:

coordinateSystem	The active coordinate system. Valid values are described in "Window Coordinate System Constants" on page 114.
flags	Flags that control how bitmaps and text are scaled.
scale	A fixed point value used to convert from the draw window's active coordinate system to native coordinates.

ntvToActiveScale	A fixed point value used to convert from the native coordinate system to the draw window's active coordinate system; the inverse of `scale`.
stdToActiveScale	A fixed point value used to convert from the standard coordinate system to the draw window's active coordinate system. This field is used internally to convert font metrics, which are stored as standard coordinates.
activeToStdScale	A fixed point value used to convert from the active coordinate system to the standard coordinate system; the inverse of `stdToActive`.

IndexedColorType

```
typedef UInt8 IndexedColorType; //1-, 2-, 4-, or 8-bit index
```

PointType

Used by `HALDrawSetPixels()`.

```
typedef struct PointType
{
   Coord           x;
   Coord              y;
} PointType
```

RectangleType

Used by `HALDraw_Rectangle`.

See also the macros `AbsToRect` and `RectToAbs`, defined in `CmnRectTypes.h`.

```
typedef struct RectangleType {
  PointType  topLeft;
```

```
  PointType  extent;
} RectangleType;
```

Field Descriptions

topLeft	Top left coordinate
extent	Width and height “co-ordinate.”

RGBColorType

Used by `HALDraw_FindIndexes` and `HALScreenLock` and many other functions.

```
typedef struct RGBColorType
{
   UInt8        index;
   UInt8        r;
   UInt8        g;
   UInt8        b;
}
RGBColorType;
```

Field Descriptions

index	Index of color or best match to current CLUT. May be unused, if color-matching is not performed.
r	Amount of red, 0->255.
g	Amount of green, 0->255.
b	Amount of blue, 0->255.

WinLockInitType

Used by `HALScreenLock`.

```
typedef enum
{
  winLockCopy, winLockErase, winLockDontCare
```

```
}
WinLockInitType;
```

Window Constants

Window Coordinate System Constants

These constants, defined in `Window.h`, specify the coordinate system to be used when drawing within a given window:

Constant	Value	Description
kCoordinatesNative	0	Use the bitmap's native coordinate system; this enables a 1-to-1 correspondence between coordinates and pixels.
kCoordinatesStandard	72	The coordinate system used by most handhelds running Palm OS 4.0 and earlier. On a single-density handheld, there is one screen pixel per standard coordinate. On a high-density screen, there is more than one screen pixel per standard coordinate.
kCoordinatesOneAndAHalf	108	One and a half times the standard coordinate system.
kCoordinatesDouble	144	Twice the standard coordinate system.

WinDrawOperation Enumeration

The `WinDrawOperation` constants are also known as *transfer mode* constants, since they specify how pixels are written to the screen during drawing operations. The new blitter modifies the operation of the original transfer modes slightly to make them more consistent. The operations defined in this table apply to double-density display.

This same set of constants is used by the Palm OS and by the DAL. The third-party developer passes a constant to the Palm OS API function, which passes it to the blitter function. Because modifications to the transfer mode operations have merely simplified DAL-level coding, existing third-party applications will not be adversely affected.

```
enum WinDrawOperationTag {winPaint, winErase,
winMask, winInvert, winOverlay, winPaintInverse,
winSwap} ;

typedef Enum8 WinDrawOperation;
```

Value Descriptions

winPaint	Write color-matched source pixel to destination; if *hasTransparency* flag is set, `winPaint` behaves like `winOverlay` instead.
winErase	Write backColor, if the source pixel is transparent.
winMask	Write backColor, if the source pixel is not transparent.
winInvert	Bitwise XOR the color-matched source pixel onto the destination. This mode does not honor the transparent color in any way.
winOverlay	Write color-matched source pixel to the destination, if the source pixel is not transparent.
winPaintInverse	Invert the source pixel color and then proceed as with `winPaint`.
winSwap	The `backColor` and `foreColor` destination colors are swapped if the source is a pattern (the type of pattern is disregarded). If the source is a bitmap, then the bitmap is transferred using `winPaint` mode instead.

The Transparent Color

As with Palm OS 4.X, a bitmap may designate a *transparent color* and set a *hasTransparency* flag. These concepts are augmented somewhat in Palm OS 5 to make the transfer modes more consistent.

When the `hasTransparency` flag is set and the transfer mode is `winPaint`, only the non-transparent pixels are copied to the destination. With bitonal data such as text and patterns, we can safely assume that the off bits are the ones designated as transparent and that the `hasTransparency` flag is always false. This assumption retains backwards compatibility.

When drawing text using the `winOverlay` mode, the non-transparent pixels are copied to the destination and the transparent pixels are skipped over. This pixel-based definition of the operation makes it suitable for 1- or multi-bit displays. With 1-bit display, the off bits are considered to be the transparent color. Note that this definition of `winOverlay` is new to Palm OS 5.

Color Defaults

The following default assumptions are made about color tables and transparent colors:

- 2-bit, 4-bit, and 8-bit source bitmaps that don't have a color table inherit the system default color table for their given bit-depth.
- 1-bit sources (bitmaps, text, and patterns) that don't have a color table are given a color table where entry 0 is the `backColor` and entry 1 is the `foreColor` (`textColor` for text)
- Bitmaps that don't specify any transparent color (text, patterns, and version 0 bitmaps) are assumed to have a transparent color of index 0 and the `hasTransparency` bit turned off.

Screen Manager Functions

HALRedrawInputArea Function

Purpose This function draws the input area. It is called by the display driver when the screen base address, the screen depth, or the hardware palette changes. It is also called when a user taps a button in the input area to draw the button inverted, and by live ink implementations to erase the ink.

Prototype
```
Err HALRedrawInputArea(const RectangleType* rectP,
    Boolean selected)
```

Parameters

→*rectP*
Bounds of a rectangle within the input area that should be redrawn. Set to NULL to redraw the whole input area. This rectangle is specified in native coordinates, relative to the input area window.

→*selected*
If true, redraws using the "selected" version of the input area bitmap. This bitmap is similar to the regular version, except that the buttons are drawn in their inverted (selected) states.

Returns `0`
If no error.

Comments HALScreenMgr.c

When the user taps in a button in the input area, and the OS supports an active input area, then the `SysHandleEvent` routine calls `HALRedrawInputArea` with the *selected* parameter set to true, in order to invert the button on the screen. It then tracks the pen, redrawing the button as necessary, until the pen goes up.

Compatibility Implemented in OS 5

HALScreenDefaultPalette Function

Purpose This function determines whether the screen palette is the default palette.

Prototype `Boolean ScrUpdateScreenBitmap(void)`

Parameters None.

Returns Returns `true`, if screen palette is the default palette. Returns `false` otherwise.

Comments HALScreenMgr.c

HALScreenDrawNotify Function

Purpose This function is used in special circumstances to notify the screen manager that the display has been modified. See Comments below.

Prototype
```
void HALScreenDrawNotify(Int16 updLeft, Int16
    updTop, Int16 updWidth, Int16 updHeight)
```

Parameters

`updLeft`
: Left coordinate of update rectangle.

`updTop`
: Top coordinate of update rectangle.

`updWidth`
: Width of update rectangle.

`updHeight`
: Height of update rectangle.

Returns None.

Comments HALScreenMgr.c

Calls `HALScreenUpdateArea()`.

This function is called in two circumstances. It is called by all blitting routines after modifying the screen display and is passed the bounds rectangle of the drawing operation. It is also used when there is an intermediate buffer. In the latter case, this function calls `HALScreenUpdateArea()` in order to transfer the intermediate buffer to the display hardware buffer. In the current release of the

PDK, an intermediate buffer should not be required. See "Intermediate Buffer Not Needed" on page 100.

See Also `HALScreenSendUpdateArea`

HALScreenGetColortable Function

Purpose This function returns the screen palette.

Prototype `ColorTableType *HALScreenGetColortable(void)`

Parameters None.

See "ColorTableType" on page 109 for more information about the `ColorTableType` data type.

Returns Screen color table.

Comments HALScreenMgr.c

This function is used by `WinRGBToIndex()` and `WinIndexToRGB()`.

See Also WinRGBToIndex
WinIndexToRGB

HALScreenInit Function

Purpose This function initializes the screen bitmap and the display hardware's palette, and returns a pointer to the screen bitmap. The screen bitmap is a `BitmapType` data structure (actually, a `BitmapTypeV3`) that the Palm OS uses to hold information about the contents currently being displayed onscreen. One of the fields of the structure is a pointer to the bits in the frame buffer.

Prototype

```
Err HALScreenInit(BitmapType** screenBitmapP,
   ColorTableType* defaultPaletteP)
```

Parameters

←*`screenBitmapP`*
Out-parameter for returning the address of the initialized screen bitmap.

→*`defaultPaletteP`*
In-parameter for pointer to a default palette, which is used to initialize the display hardware's palette and the screen

bitmap's color table. Color table entries are stored in `colorEntries[]`, which is a field of the screen globals structure (`GScrGlobals`).

See "BitmapTypeV3" on page 104 for more information about the data type.

See "ColorTableType" on page 109 for more information about the data type.

Returns Address of initialized bitmap in `*screenBitmapP`.
Color table entries stored in the screen globals structure. See Parameters above.

Comments HALScreenMgr.c

This function is called by `WinScreenInit()` when the system boots.

See Also WinScreenInit

HALScreenLock Function

Purpose This function reduces screen flicker and ensures smooth screen updates.This function locks the screen, returning the address of a new offscreen buffer to which the blitter writes.

Prototype `UInt8 *HALScreenLock(WinLockInitType iMode)`

Parameters →*`iMode`*

WinLockCopy—copy old screen to new.

WinLockErase—erase new screen to white.

WinLockDontCare—don't do anything.

Returns 0If no error.

Comments HALScreenMgr.c

Replaces `ScrScreenLock()` function from the HAL API for Palm OS 4.0.

This function "locks" the display screen of the Palm OS device by moving the existing frame buffer to a different address and then returning the address of a new, offscreen buffer. The driver continues to display the moved buffer while the blitter writes to the offscreen buffer. When the screen is "unlocked," the contents of the offscreen buffer are reflected onscreen.

To support screen locking, your Palm OS device must have enough VRAM for two frame buffers. If screen locking is not supported, the HAL, via `HALDisplayLock()`, returns NULL to `HALScreenLock()`.

The controller supported by the sample DAL creates an offscreen buffer in VRAM.

The screen lock count represents the number of times that `HALScreenLock()` has been called. The screen must be unlocked as many times as it was locked in order to actually update the device display screen.

When an application locks the screen, the window manager calls the screen manager which calls the display driver: `WinScreenLock()` calls `HALScreenLock()`, which calls `HALDisplayLock()`.

See Also HALScreenUnlock
WinScreenLock
HALDisplayLock

HALScreenPalette Function

Purpose This function sets the globals screen palette, and programs the hardware palette.

Prototype

```
Err ScrPalette(Int16 startIndex,
    UInt16 numEntries, ColorTableType *tableP,
    ColorTableType **palettes)
```

Parameters

`startindex`
Starting palette entry for operation.

`numEntries`
Number of palette entries to operate on.

`tableP`
Source color table.

`palettes` Array of default system palettes.

See "ColorTableType" on page 109 for more information about the data type.

Returns sysErrNoFreeResourceThere is a memory allocation error.

errNone Success.

Comments HALScreenMgr.c

This function is called by `WinPalette()` when the screen palette is changed. See `WinPalette` for a description of the arguments.

Update the `GScrGlobalsP->colorTranslateP` array when setting the palette.

See Also HALScreenUpdateBitmap
WinPalette

HALScreenSendUpdateArea Function

Purpose This function calls the display transfer function defined by the display driver. The display transfer function sends the contents of the intermediate buffer, if any, to the display hardware buffer. The updated bounds of the screen rectangle are then accessed by the display driver.

Prototype `void HALScreenSendUpdateArea(Boolean force)`

Parameters `force`

If true, send update area regardless of last time it was sent.

If false, send update area only if the time threshold from the last update has passed.

Returns None.

Comments HALScreenMgr.c

Called by `HALScreenDrawNotify()`.

This function is called periodically to send an updated region of the blitter's intermediate buffer to the hardware for display controllers that do not match the blitter's standard format. You will probably not need to use an intermediate buffer. See "Intermediate Buffer Not Needed" on page 100.

See Also `HALScreenDrawNotify`.

HALScreenUnlock Function

Purpose This function works in concert with `HALScreenLock()` to reduce screen flicker and ensure smooth screen updates.It “unlocks” the screen by replacing the buffer that the driver is currently displaying with the offscreen “virtual” buffer.

Prototype `Err HALScreenUnlock(void)`

Parameters None.

Returns `0.`

If no error.

Comments HALScreenMgr.c

Replaces the `ScrScreenUnlock()` function from the HAL API for Palm OS 4.0.

This function sets the base address of the driver’s current buffer to the base address of the offscreen frame buffer that was established by an earlier call to `HALScreenLock()`. Consequently, the contents of the offscreen buffer are displayed onscreen.

If the DAL uses the system heap to allocate its screen buffer, it gets deallocated here. The controller supported by the sample DAL, however, allocates its screen buffer in VRAM.

When an application unlocks the screen, the window manager calls the screen manager which calls the display driver: `WinScreenUnlock()` calls `HALScreenUnlock()`, which calls `HALDisplayUnlock()`.

See Also HALScreenLock
WinScreenUnlock
HALDisplayUnlock

HALScreenUpdateBitmap Function

Purpose This function updates the screen bitmap when an application changes the screen depth. It sets the bitmap's geometry, depth, and color attributes. The screen bitmap is a BitmapTypeV3.

Prototype `Err ScrUpdateScreenBitmap(UInt16 depth)`

Parameters `depth`
Depth in bits per pixel or 0 to preserve.

Returns `0`
If no error.

Comments HALScreenMgr.c

The screen `colorTable` is not initialized here, but rather is initialized in a separate call to HAL`ScreenPalette()`.

This function is called by `WinScreenMode()`.

See Also HALScreenPalette
WinScreenMode

Blitter Functions

The blitter for the current DAL fully supports double-density screen display. By *double-density*, we mean a screen display that is 320 x 320 pixels, which is double the 160 x 160 pixel screen of the original Palm OS devices. The functions are presented in alphabetical order.

HALDraw_Bitmap Function

Purpose Function for copying bits from a source bitmap to a target bitmap. This generic function is used to transfer the contents of a rectangular area of a source bitmap to another rectangular area in a target bitmap.

Prototype

```
Err HALDraw_Bitmap ( CanvasType *canvasP,
    BltBitmapType *srcBitmapP,
    RectangleType *dstClippedP, Int16 offsetX,
    Int16 offsetY )
```

Parameters

→*canvasP*
: The canvas contains the graphics state of the windows, and is passed to all blitter functions.

→*srcBitmapP*
: The bitmap of the source window passed to WinCopyRectangle. Bitmap to be copied to the target specified in `canvasP`. If the compressed bit is set in the source window flags, this routine will automatically decompress the source bitmap as it copies it to the destination.

→*dstClippedP*
: The clipping bounds of the destination bitmap. Do not write outside this destination rectangle. If the compressed bit is set in the destination window flags, this routine will automatically compress the source bitmap as it copies it to the destination.

→*offsetX*
: Offset each of the source's scaled pixels by this much in the x direction.

→*offsetY*
: Offset each of the source's scaled pixels by this much in the y direction.

Returns

`0`
: If no error.

Comments

`HALDrawing.c`

This routine can decompress, scale, color match, depth convert, offset, and clip pixels while moving them to a destination with one of 5 transfer modes and optional halftoning.

The blitter assumes that the rectangle that it is being asked to display is readable. In other words, the blitter honors the clipping defined in the source window's data structure.

Scaling of the source bitmap is independent of destination location.

See Also

`HALDraw_Rectangle`

HALDraw_Chars Function

Purpose This function is the font blitting routine.

Prototype

```
void HALDraw_Chars (const CanvasType* canvasP,
   Coord toX, Coord toY, const Char* charsP_in,
   Int16 len, FontPtr fontP, FontMapPtr fontMap,
   DrawCharCheckPro charCheckProc)
```

Parameters

→*canvasP*
The canvas contains the graphics state of the windows, and is passed to all blitter functions. It indicates where and how the characters will be rendered.

→*toX*
The x coordinate of the upper left corner of the first character to blit.

→*toY*
The y coordinate of the upper left corner of the first character to blit.

→*charsP*
The characters to blit. (May be multi-byte.)

→*len*
Number of bytes. (Characters may be multi-byte.)

→*fontP*
Font to use.

→*fontMap*
Font metrics.

→*charCheckPro*
Callback function used to verify potentially invalid characters.

Returns None.

Comments `HALDrawing.c`

There is no need to pre-clip the input characters, as this routine can do it as efficiently as any other.

See Also HALDraw_Chars

HALDraw_FindIndexes Function

This function will go through `numEntries` (starting from 0) of the `matchColorsP` table, matching the RGB values to the closest index in the `refColorTableP` table (which is the current color lookup table, or CLUT). For each specified entry of the match color table, this function sets the `index` field to the index of the entry in the reference color table that constitutes the best fit value.

Prototype

```
Err HALDrawFindIndexes(UInt16 numEntries,
    RGBColorType *matchColorsP,
    const ColorTableType *refColorTableP)
```

Parameters

→*`numEntries`*
The number of entries in the table to be matched.

↔ *`matchColorsP`*
Color entries to find matches for.

→*`refColorTableP`*
The CLUT. It is the Reference color table to match colors against. If NULL, use the table stored in the screen globals.

See "RGBColorType" on page 113 for more information about the `RGBColorType` data type.

See "ColorTableType" on page 109 for more information about the `ColorTableType` and `RGBColorType`data types.

Returns

`0`
If no error.

Comments

HALDrawing.c

Replaces `BltFindIndexes()` function from the HAL API for Palm OS 4.0.

This function is called by `WinRGBToIndex()` in blitter. In the source code comments, you may see the term "screen globals." Keep in mind that the screen globals represent the hardware palette.

See Also

HALDraw_FindIndexes

HALDraw_GetPixel Function

Purpose This function returns the pixel value of the specified x,y coordinate in the given bitmap.

Prototype
```
UInt32 HALDraw_GetPixel (const CanvasType* canvas,
    Coord x, Coord y, Boolean asIndex)
```

Parameters

→*canvas*
A pointer to a Palm OS structure defining the location of the bitmap containing the pixel.

→*x*
x-coordinate of pixel.

→*y*
y-coordinate of pixel.

→*asIndex*
`True` or `false`. See Results section below.

Returns Value of pixel. If `asIndex` is `true`, return value is an index into the CLUT (color lookup table). If `asIndex` is `false`, return is a 16-bit pixel value, encoded as a 5-6-5 RGB.

Comments `HALDrawing.c`

See Also HALDraw_GetPixel

HALDraw_Line Function

Purpose Function for drawing lines.

Prototype
```
Err HALDraw_Line(const CanvasType *CanvasP,
    Int16 x1, Int16 y1, Int16 x2, Int16 y2,
    Int16 PenWidth)
```

Parameters

→*iCanvas*
The canvas contains the graphics state of the windows, and is passed to all blitter functions.

→*x1*
x-coordinate of the start of the line.

→*y1*
y-coordinate of the start of the line.

→*x2*
x-coordinate of the end of the line.

→*y2*
y-coordinate of the end of the line.

→*penWidth*
Width (or height) of pen. For lines moving in a horizontal or mostly-horizontal direction, this gives the height of the line--pixels extend below the pen location to add fullness. For lines moving in a vertical, or mostly-vertical, or a 45-degree direction, this gives the width of the line—pixels extend to the right of the pen location to add fullness.
A value of 1 is consistent with the older (single-density) blitters and produces a thin line.
A value of 2 allows low-density applications to draw appropriately-thick lines on a double-density display.

See "CanvasType" on page 108 for more information about the `CanvasType` data type. It contains the graphics state of the windows, and is passed to all blitter functions.

Returns None.

Comments `HALDrawing.c`

Coordinate points are inclusive.

The clipping in Palm OS 5 has changed. Briefly put, clipping determines if a given pixel is drawn or not. Clipping does not have any effect on which pixels are chosen to represent a line.

See Also HALDraw_Line

HALDraw_Pixels Function

Purpose Function for drawing a pixel.

Prototype
```
void HALDraw_Pixels ( const CanvasType* canvasP,
    Int16 numPoints, const PointType* pts,
    Int16 penWidth)
```

Parameters

→*canvasP*
The canvas contains the graphics state of the windows, and is passed to all blitter functions. It indicates where and how the pixel(s) will be rendered.

→*numPoints*
: # of points in the array

→*pts*
: Constant pointer to an array of `PointType`

→*penWidth*
: When `penWidth` is more than 1, the routine actually draws little squares that extend down and to the right.

Returns None.

Comments `HALDrawing.c`

This function is used to draw pixels. A single pixel can be drawn using a numPoints value of 1.   Internally, the blitter may use different algorithms when the number of pixels to draw is large.Drawing multiple pixels at once with a single call is always faster than calling the blitter repeatedly.

See Also [HALDraw_Pixels]

HALDraw_Rectangle Function

Purpose Function for rendering filled or framed rectangles, including rectangles with rounded corners.

Prototype

```
void HALDraw_DrawRectangle (
    const CanvasType* canvasP,
    const RectangleType* rectP, Coord radius,
    Coord penWidth)
```

Parameters

→*canvasP*
: The canvas contains the graphics state of the windows, and is passed to all blitter functions. It indicates where and how the rectangle will be rendered.

→*rectP*
: Rectangle to be drawn

→*radius*
: Radius of curvature for the rectangle's corners, in pixels.

→*penWidth*
: If this value is 0, the rectangle will be filled. If this value is a positive integer, the rectangle will have a frame whose width is that value. Outsetting the rectangle parameter by the

penWidth will cause the framed rectangle to be drawn completely and precisely outside the filled rectangle version, a feature used when drawing push buttons. The foreColor, backColor, transferMode, and pattern are all used.

Returns None.

Comments `HALDrawing.c`

This routine never draws pixels outside the input rectangle.

The current blitter never draws a pixel more than once while blitting a rounded rectangle.

See Also HALDraw_Rectangle

HALDrawInit Function

Purpose This function initializes blitter globals.

Prototype

```
Err HALDrawFindIndexes(UInt16 numEntries,
   RGBColorType *matchColorsP,
   const ColorTableType *refColorTableP)
```

Parameters None.

Returns None.

Comments HALDrawing.c

Called by `HALScreenInit()`.

See Also HALScreenInit

15

Sound Support

This chapter describes the HAL sound functions. These functions are called by the Palm Sound Manager.

The most important part of the HAL sound implementation is the sound mixer. This is software that's expected to accept as many as 16 streams of stereo sampled data, and mix them into a single (stereo) output signal that can be sent to the device's sound hardware (speakers, headphone jack, line-out jacks, etc.). The HAL is also expected to produce a single stereo input signal by reading data from a microphone, line-in jack, or other input device.

Each of the output streams (created through `HALSoundOpen`) supply data to the Sound Manager as a series of buffers that are retrieved through callback functions. More specifically, the Sound Manager calls into application code, passing a buffer that the callback is expected to fill with sound data. The Sound Manager then passes these buffers to the HAL by calling `HALSoundWrite`, which function is responsible for scooping out the sound data and dumping it into a given stream.

NOTE: The HAL isn't required to always be able to supply 16 output streams. In the sample implementation for Palm OS 5, for example, there are sufficient system resources to only create 15 streams. Sound Manager users will be warned that the number of available output streams isn't guaranteed.

On the input side, the HAL must produce the "next" buffer of sound data when the Sound Manager calls `HALSoundRead`. The function is passed a buffer into which it writes data that was read from an input device.

The HAL also includes functions that support legacy square wave sound generation: `HALSoundPlay`, `HALSoundOff`

Palm OS 5 provides a sample implementation of the HAL sound functions, including a sound mixer, in `HALSound.cpp`.

All elements described here are declared in `HALSound.h`. A sample implementation is given in. `Samples\LubbockRef\Src\HALSound.cpp`.

HAL Sound Structures and Constants

HALSndIoctlCmds Enum

Purpose These enumerated constants represent the commands that the `HALSoundIoctl` function is expected to handle.

Prototype

```
enum HALSndIoctlCmds
    {
      CMD_SETFORMAT,
      CMD_SETVOLUME,
      CMD_GETVOLUME,
      CMD_SETPAN,
      CMD_GETPAN,
      CMD_ALLOCSTREAMBUFFERS,
      CMD_STOP
    };
```

Comments See `HALSoundIoctl` for information on the commands that these constants represent.

HALSoundAllocStreamBufType Typedef

Purpose Structure that contains the data buffers used in sampled sound streams.

Prototype

```
typedef struct
    {
      Int32  size;
      char *buf[2];
    } HALSoundAllocStreamBufType;
```

Fields

`size`
Size of the data buffers (all buffers must be the same size).

`buf`
: Array of pointers to the data buffers. Currently, only double-buffering is allowed.

Comments When executing the CMD_ALLOCSTREAMBUFFERS command, the `HALSoundIoctl` function expects its final argument to be a pointer to a `HALSoundAllocStreamBufType` structure.

HALSoundInitStreamType Typedef

Purpose Describes the format of a sound stream; used by the

Prototype

```
typedef struct
{
  UInt32  samplerate;
  SndSampleType  type;
  SndStreamWidth  width;
} HALSoundInitStreamType;
```

Fields

`samplerate`
: Sampling rate in frames-per-second. The Sound Manager allows sampling rates up to 96000.

`type`
: A constant that represents the sample quantization and endianness. The `SndSampleType` constants, defined in `SoundMgr.h`, are described in the *Sound Manager* chapter of the Palm OS Reference manual.

`width`
: A constant that represents the number of audio channels; either sndMono (one channel) or sndStereo (two channels).

Comments When executing the `CMD_SETFORMAT` command, the `HALSoundIoctl` function expects its final argument to be a pointer to a `HALSoundInitStreamType` structure.

HAL Sound Support Functions

HALPlaySmf Function

Purpose Generates a series of simple square wave tones whose frequencies, amplitudes, and (implied) durations are defined in a Standard MIDI File.

Prototype
```
Err HALPlaySmf ( void *chanP, SndSmfCmdEnum cmd,
    UInt8 *smfP, SndSmfOptionsType *optP,
    SndSmfChanRangeType *chanRangeP,
    SndSmfCallbacksType *callbacksP,
    Boolean bNoWait )
```

Returns Returns 0 if the request was handled, and non-zero otherwise. See the **Comments**, below, for more information.

Comments This function is an implementation of the `SndPlaySmf` function. For descriptions of what the parameters mean and how the function is expected to behave see `SndPlaySmf` in the “Sound Manager” chapter of *Palm OS Programmer’s API Reference*.

With regard to return values, `HALPlaySmf` can either handle the request and return 0, or it can punt by returning any other value. In the latter case, the Sound Manager opens and decodes the MIDI file itself and issues a series of `HALSoundPlay`/`HALSoundStop` calls to play the MIDI data.

If the caller requested the SMF duration and your implementation returns non-zero, the Sound Manager handles the entire operation without calling into the HAL.

HALSoundClose Function

Purpose Closes a sampled sound stream.

Prototype `Err HALSoundClose ( Int32 streamRef )`

Parameters

→*streamRef*
  Stream identifier for the stream that wants to be closed.

Returns Returns 0 upon success; otherwise non-zero.

See Also [HALSoundOpen](#)

HALSoundDispose Function

Purpose Called by the system to shut down and clean up the sound facilities.

Prototype `Err HALSoundDispose (void)`

Returns The function returns 0 upon success, and non-zero otherwise.

Comments The sample implementation uses this function to shut down and destroy the sound mixer, and unload the sound driver.

See Also `HALSoundInitialize`

HALSoundInitialize Function

Purpose Called at startup to initialize the sound facilities.

Prototype `Err HALSoundInitialize (void)`

Returns The function returns 0 upon success, and non-zero otherwise.

Comments The sample implementation uses this function to load the sound driver, initialize the sound mixer, and create the task in which the mixer runs.

See Also `HALSoundDispose`

HALSoundIoctl Function

Purpose Sets/gets attributes of a sound stream.

Prototype

```
Err HALSoundIoctl ( Int32 streamRef,
   Int32 command, void *data );
```

Parameters

→*streamRef*
Cookie that identifies the sound stream, as returned by `HALSoundOpen`.

→*command*
Constant that represents the requested command. One of the `HALSndIoctlCmds` *values.*

↔ *data*
Command-specific data

Returns

`errNone`
Success.

`sndErrMemory`
: Not enough memory to allocate the data buffers (CMD_ALLOCSTREAMBUFFERS).

`sndErrBadParam`
: Invalid format (CMD_SETFORMAT) or setting value (CMD_SETVOLUME and CMD_SETPAN).

Comments The `HALSndIoctlCmds` constants and their associated data values are described somewhere near here:

- `CMD_SETFORMAT`. Sets the stream's sound format. The `data` is a pointer to a `HALSoundInitStreamType` structure that describes the desired format.
- `CMD_SETVOLUME`. Sets the stream's volume in the range [0, 1024], where 0 is inaudible and 1024 is full volume. The `data` is the requested volume as a `UInt32`.
- `CMD_GETVOLUME`. Returns, by reference in `data`, the stream's current volume as a `UInt32`.
- `CMD_SETPAN`. Sets the stream's stereo placement in the range [-1024, 1024] from hard left to hard right. The `data` is the requested pan setting as a `UInt32`.
- `CMD_GETPAN`. Returns, by reference in `data`, the stream's current stereo pan setting as a `UInt32`.
- `CMD_ALLOCSTREAMBUFFERS`. Allocates the stream's sound data buffers. The `data` is a pointer to a `HALSoundAllocStreamBufType` structure that indicates the desired buffer size (in its `size` field), and passes an array of (unallocated) buffer pointers. The function allocates memory for the buffers, using the size indication as a suggestion (all buffers in a single stream must be the same size). The function then resets `data->size` to the allocated size of a single buffer.
- `CMD_STOP`. This command is issued whenever `SndStreamStop` is called. There is no parameter block for this comand; the value of `data` is undefined.

HALSoundOff Function

Purpose Stops playing the current sound.

IMPORTANT: This function must never block.

Prototype `Err HALSoundOff (void)`

Returns Returns 0 upon success; otherwise non-zero.

Comments Any sound previously started through `HALSoundPlay` is stopped. Stream sounds (generated through HALSoundWrite) aren't affected.

Compatibility Replaces the `HwrSoundOff()` function from the HAL API for Palm OS 4.0.

See Also `HALSoundPlay`

HALSoundOpen Function

Purpose Opens a new sampled sound stream (input or output) and returns an identifier that represents the open stream.

Prototype
```
Int32 HALSoundOpen ( Char *device, int flags,
   Err *error )
```

Parameters

→*device*
Specifies the stream's direction. Either halSoundADC (input) or halSoundMixer (output).

→*flags*
Currently unused.

←*error*
Used to return the function's status.

Returns The function itself returns a stream identifier, where a valid identifier is greater than zero. If the function returns 0 (or a negative number), the stream was not opened. The error argument is set to a indicative code, typically one of the following.

`errNone`
Success.

`sndErrMemory`
Couldn't allocate required resources.

See Also `HALSoundClose`

HALSoundPlay Function

Purpose Generates a tone with a given frequency, amplitude, and duration.

Prototype
```
Err HALSoundPlay ( UInt32 frequency,
   UInt16 amplitude, UInt32 duration )
```

Parameters

→*frequency*
Frequency in Hz.

→*amplitude*
Amplitude in the range [0, `sndMaxAmp`].

→*duration*
Duration in milliseconds.

Returns Returns 0 upon success; otherwise non-zero.

Comments The sample implementation generates a square wave tone, in emulation of a traditional small-device hardware tone generator. Only one tone can be produced at a time.

Compatibility Replaces the `HwrSoundOn()` function in the HAL API for Palm OS 4.0.

See Also `HALSoundOff`

HALSoundRead Function

Purpose This is the sound recording function: It reads data from an open input sound stream and places it in a caller-defined buffer.

Prototype
```
Int32 HALSoundRead ( Int32 streamRef,
   void *buffer, Int32 bufferSize, Err *error )
```

Parameters

→*streamRef*
Stream identifier for the stream that wants to be read from.

←*buffer*
Buffer into which the data is placed.

→*bufferSize* *Size of buffer, in bytes.*

←*`error`*
Error code.

Returns The function returns the number of bytes read; the error status is returned in `error`. A positive return indicates success, and `error` is set to `errNone`. A direct return of 0 means nothing was read, and `error` is set to an error code, including:

`sndErrInvalidStream`
`streamRef` isn't open, or is otherwise invalid.

`sndErrBadParam`
The buffer size is incompatible with the size of the stream's data buffers.

See Also `HALSoundOpen`, `HALSoundWrite`

HALSoundSleep Function

Comments Currently unused.

HALSoundWake Function

Comments Currently unused.

HALSoundWrite Function

Purpose This is the sound playback function: It takes sound data from a caller-supplied buffer and writes it into an open output sound stream.

Prototype
```
Int32 HALSoundWrite ( Int32 streamRef,
   void *buffer, Int32 bufferSize, Err *error )
```

Parameters

→*`streamRef`*
Stream identifier for the stream that wants to be written to.

→*`buffer`*
Buffer from which the data is taken.

→*`bufferSize`*
Size of `buffer`, in bytes.

←*error*
: Error code.

Returns The function returns the number of bytes written; the error status is returned in `error`. A positive return indicates success, and `error` is set to `errNone`. A direct return of 0 means nothing was written, and `error` is set to an error code, including:

`sndErrInvalidStream`
: `streamRef` isn't open, or is otherwise invalid.

See Also `HALSoundOpen`, `HALSoundRead`

16

Timer Support

This chapter describes the API functions of the HAL that deal with the timer. They are described in alphabetical sequence.

For more information about the Time Manager, see the *Palm OS Programmer's Companion* and the *Palm OS Programmer's API Reference*.

Timer Support Data Structures

None applicable.

Timer Support Functions

HALDelay Function

Purpose Waits for the given amount of time.

Prototype `void HALDelay(UInt32 microseconds)`

Parameters `microseconds`
The number of microseconds to wait.

Returns None.

Comments `CTLTimer.c`

This function is called by various hardware routines and may be called from within an interrupt handler.

Replaces `HwrDelay()` function from the HAL API for Palm OS 4.0.

Part II
Kernel Hardware Abstraction Layer (kHAL)

17

kHAL Functions

The kHAL functions are called by the kernel to initialize registers, set up tasks, check system state, and so on. Many of these functions are platform-dependent: When Palm OS 5 is ported to a new ARM platform, the kHAL functions must be re-visited to see if their implementations need to be altered (or completely rewritten) to match the new hardware.

The kHAL portion of the sample implementation is in:

```
Development Kit\Samples\LubbockRef\kHAL\Src\
```

Overview

The kHAL functions fall into three groups:

- The **CPU-specific functions** exhibit direct control over the CPU:
  - `kHAL_DisableInt` disables IRQ interrupts.
  - `kHAL_EnableInt` re-enables IRQ interrupts.
  - `kHAL_CPULock` “locks” the CPU, presumably for uninterrupted execution of a critical section. (Currently a no-op.)
  - `kHAL_CPUUnlock` is the antidote to `kHAL_CPULock`. (Currently a no-op.)
  - `kHAL_Doze` conserves power by halting the CPU until the next interrupt.
- The **system initialization functions** are called during the boot sequence (only). In the order that they’re called, they are:
  - `kHAL_Init` initializes the CPU vectors.

- [kHAL_RegisterInterruptHandler]() installs a software interrupt handler for a particular device. (This may be called more than once during the boot sequence.)
- [kHAL_SwitchToFirstTask]() performs a context switch to the first task (the main kernel task).

- The **multi-tasking functions** are used to manage a context switch:
  - [kHAL_CreateInitialTaskContext]() sets up the context for a freshly created task (*any* freshly created task, not just the system's first task).
  - [kHAL_SetTaskReturnValue]() sets the return value for a task that has timed out.

NOTE: If you poke around in the sample kHAL implementation, you'll also see `kHAL_CheckIdle` and `kHAL_CheckContextSwitchNeeded`. These aren't actual kHAL functions; they're part of the SWI handler implementation that's provided with the sample. There's no reason to reimplement these functions.

Only the CPU-specific functions *need* to be (wholly) reimplemented for each CPU.

The other functions are "kernel-specific": Their implementations are highly dependent on the structure of the kernel (Palm OS Kernel 1.0). Since you can't change the kernel, you should always include the sample code for the kernel-specific kHAL functions in your own implementations.

The individual kHAL functions are described below, listed in recognizably alphabetical order.

kHAL Functions

kHAL_CPULock Function

IMPORTANT: `kHAL_CPULock` is provided for completeness, but is currently a no-op. Don't bother re-implementing it.

Purpose Locks the CPU so that the current task can't be switched out. The lock is released by `kHAL_CPUUnlock`.

Declared In `(none)`

`Samples\LubbockRef\kHAL\Src\KernelCall.c`

Prototype `ER kHAL_CPULock ( UInt32 *cpsr )`

Parameters `cpsr`
A pointer to the value of the caller's `CPSR`.

kHAL_CPUUnlock Function

IMPORTANT: `kHAL_CPUUnlock` is provided for completeness, but is currently a no-op. Don't bother re-implementing it.

Purpose Undoes the effect of a previous `kHAL_CPULock`, allowing the currently locked-in task to be switched out.

Declared In `(none)`

Example `Samples\LubbockRef\kHAL\Src\KernelCall.c`

Prototype `ER kHAL_CPUUnlock ( UInt32 *cpsr )`

Parameters `cpsr`
A pointer to the value of the caller's `CPSR`.

Result The return value is ignored.

kHAL_CreateInitialTaskContext Function

Purpose Creates the context for a freshly minted task. Called every time a new task is created.

Declared In `Palm_OS_DAL_Support\Kernel\MCK\KernelPrv.h`

Example. `Samples\LubbockRef\kHAL\Src\KHAL.c`

Prototype

```
void kHAL_CreateInitialTaskContext ( TCB *task,
    Int32 startCode )
```

Parameters

`task`
A pointer to the Task Control Block for the new task.

`startCode`
Argument data that's passed to the task's entry function.

Comments The sample implementation pushes function call data onto the task's stack, and sets up the task's "saved context" area. You must include the sample implementation code in your own implementation of this function.

kHAL_DisableInt Function

Purpose Disables interrupts. Interrupts remain disabled until `kHAL_EnableInt` is called.

Declared In `Palm_OS_DAL_Support\Kernel\MCK\KernelPrv.h`

Example `Samples\LubbockRef\kHAL\Src\IRQ_A.s`

Prototype `void kHAL_DisableInt ( void )`

Comments The sample implementation disables IRQ interrupts, but not FIQ interrupts.

kHAL_Doze Function

Purpose Halts the CPU until the next interrupt occurs. The sole purpose of this function is to conserve power.

Declared In `Palm_OS_DAL_Support\Kernel\MCK\KernelPrv.h`

Example `Samples\LubbockRef\kHAL\Src\kHAL_A.s`

Prototype `void kHAL_Doze ( void )`

kHAL_EnableInt Function

Purpose Re-enables interrupts, undoing the effect of a previous `kHAL_DisableInt`.

Declared In `Palm_OS_DAL_Support\Kernel\MCK\KernelPrv.h`

Example `Samples\LubbockRef\kHAL\Src\IRQ_A.s`

Prototype `void kHAL_EnableInt ( void )`

kHAL_Init Function

Purpose Initializes the CPU's interrupt vectors. Called once during the boot sequence.

Declared In `Palm_OS_DAL_Support\Kernel\MCK\KernelPrv.h`

Example `Samples\LubbockRef\kHAL\Src\KHAL.c`

Prototype `void kHAL_Init ( void )`

Comments If you want to base your version on the sample implementation, note the following: For convenience, the sample DAL includes a `kHAL_Init` helper function, `PrvInstallHandler`, that does the actual vector initialization. If you use `PrvInstallHandler`, you should modify the code so that it *always* takes the branch that looks like this:

```
GHwrExceptionHandlers[vector] = routine;
```

kHAL_RegisterInterruptHandler Function

Purpose Adds a hardware interrupt handler to the dispatch table.

Declared In `Palm_OS_DAL_Support\Kernel\MCK\KernelPrv.h`

Example `Samples\LubbockRef\kHAL\Src\IRQ.c`

Prototype

```
ER kHAL_RegisterInterruptHandler (
    UInt32 interruptID,
    FP interruptHandler, void *handlerArg )
```

Parameters

`interruptID`
A number that uniquely identifies the device that you want this handler to handle.

`interruptHandler`
The address of the (APCS-compatible) interrupt routine.

`handlerArg`
The address of a data area for the interrupt routine.

Result The sample implementation returns the following; your implementation should follow suit:

`E_OK`
Success.

`E_PAR`
Bad `interruptID` value. The sample implementation pre-installs the first three interrupt handlers (for `FIQ`, `IRQ`, and `ABORT` interrupts); an attempt to overwrite these handlers is thwarted, and `E_PAR` is returned.

Comments You should only need to implement this function if you're redesigning the dispatch table.

kHAL_SetTaskReturnValue Function

Purpose Sets the return value for a task that has timed out.

Declared In `Palm_OS_DAL_Support\Kernel\MCK\KernelPrv.h`

Example `Samples\LubbockRef\kHAL\Src\KHAL.c`

Prototype

```
void kHAL_SetTaskReturnValue (UInt32 *savedContext,
Int32 returnValue);
```

Parameters

`savedContext`
A pointer to the task's context array.

`returnValue`
The return value suggested by the kernel.

Comments The sample implementation stuffs the `returnValue` into the task's return register (`r0`—this is the second element in the `savedContext` array. The entire implementation looks like this:

```
savedContext[1] = returnValue;
```

It's recommended that you use this implementation without modification. In this implementation, the returnValue is always `E_TMOUT`, the error code that indicates that the task has timed out.

kHAL_SwitchToFirstTask Function

Purpose Jump starts the task switching mechanism by stuffing data into the context area, and then switching to that context.

Declared In `Palm_OS_DAL_Support\Kernel\MCK\KernelPrv.h`

Example `Samples\LubbockRef\kHAL\Src\KHAL_A.s`

Prototype

```
void kHAL_SwitchToFirstTask (
    UInt32 *savedContext )
```

Parameters

`savedContext`
: A pointer to the context of the task that's being switched to. This context is created by the kernel; you probably don't want to mess with it.

Comments The sample implementation moves the argument into `r2`, and then calls a kernel routine (`RestoreTaskContext`) that performs the actual context switch.

`kHAL_SwitchToFirstTask` is always called near the end of the boot process; it's passed a context that represents the main kernel task.

Part III
Kernel Abstraction Layer (KAL)

18

The KAL

The Kernel Abstraction Layer (KAL) is a set of functions that let you create and manage resources (or, as we call them here, "objects") that represent fundamental kernel functionality, such as tasks, mutual exclusion locks, inter-process communication, and so on.

You invoke the KAL functions in your HAL and kHAL implementations. The KAL functions can't be reimplemented.

Kernel Object Types

There are six kernel object types: tasks, mutexes, semaphores, event groups, mailboxes, and timers:

- An additional KAL chapter, Chapter 19, "KAL Generic API," on page 161, lists and explains the KAL error codes and other constants that apply to more than one kernel object.
- A task is an independent thread of execution. See Chapter 20, "Tasks," on page 165.
- A mutex is "one task at a time" lock that's typically used to protect non-reentrant code. See Chapter 22, "Mutexes," on page 191.
- A semaphore is a task synchronization object; it's similar to a mutex, but is much more flexible. Semaphores are typically used to coordinate dependent tasks. See Chapter 21, "Semaphores," on page 185.
- An event group is a "conditional" task synchronization object. It lets you define an arbitrary set of conditions that must be met before a task is allowed to continue execution. See Chapter 23, "Event Groups," on page 197.
- A mailbox is an inter-process message queue. See Chapter 24, "Mailboxes," on page 205.

- A timer object lets you ask the the kernel to invoke functions on your behalf. You typically use timers to perform periodic checks on the system. See Chapter 25, "Timers," on page 211.

Object Count Limits

The number of (simultaneous) kernel objects that can be created across the entire system is limited by the values set in the `KernelConfig.h` file. These limits are:

- Tasks: 32
- Mutexes: 64
- Semaphores: 64
- Event groups:16
- Mailboxes:16
- Timers: 24

Note that the system itself creates and uses some number of these objects. For example, you code won't be able to create 31 tasks (32 minus the calling task) because of the handful of tasks—such as the timer task and the kernel task itself—that are hardwired into the system.

The limits defined in `KernelConfig.h` are currently unmodifiable.

Object ID Numbers

Each kernel object is identified by an ID number that's *contemporaneously* unique for the object's type. As long as the object exists, it's ID number won't collide with any other object of the same type.

However, ID numbers aren't *persistently* unique. After you delete an object, some other newly created object (of the same type) may take on the old object's ID. Because of this, you must be careful when you cache a kernel object ID number.

Object ID number values have no significance by themselves. The value doesn't even signify the object's type: For example, a task object, semaphore object, mutex object (and so on), can all have the same ID value.

For all objects, value 0 is invalid.

19

KAL Generic API

This section lists the KAL constants that are used by (potentially) all the KAL object types. It also describes the two KAL startup functions (KALInit and KALStart) that are called by the system. Note that these functions are described here for information only; you should never call these functions yourself.

KAL Generic Constants

KAL Error Constants

Purpose Error codes that are returned by the KAL functions. For all KAL functions, a return value of 0 indicates success. Not all of these constants are unique to the KAL functions—`errNone` and `kDALTimeout` are used by other DAL functions.

Declared In `Kernel.h, Common/CmnErrors.h`

Constants

`errNone`
: Success.

`kKALErrBadParam`
: Argument value not in range.

`kKALErrNoFreeResource`
: No more available objects: All allocated objects of this type are being used.

`kKALErrNoFreeRAM`
: Not enough memory to create the requested object.

`kKALErrInvalidContext`
: The function can't be called because task switching is disabled.

`kKALErrSemInUse`
: Currently unused.

`kKALErrInvalidID`
: The (argument) ID doesn't identify a valid object of the appropriate type. This is essentially the same error as kKALErrObjectNotExist.

`kKALErrObjectDeleted`
: An object that was blocking the function (a semaphore or mutex, as examples) was deleted.

`kKALErrObjectInvalid`
: The requested operation is illegal because the object (which does exist) is in the "wrong state." This applies, primarily, to tasks; for example, a task can't call `KALTaskTerminate` on itself.

`kKALErrQueueOverflow`
: The object has (already) reached a queue or nesting limit.

`kKALErrWaitReleased`
: Currently unused.

`kKALErrObjectNotExist`
: The object upon which this function is operating doesn't exist. This is essentially the same error as kKALErrInvalidID.

`kKALErrNotOwner`
: The requested operation can only be performed by the object's owner. Currently, this is used by `KALMutexRelease` only.

`kDALTimeout`
: The function has returned because a timeout limit has expired.

KAL Timeout Constants

Purpose These constants represent edge-case timeout values, and provide a convenient "wait a moment" constant. You can pass these constants to functions that take timeout arguments.

Declared In `Kernel.h`

Constants `kTimeoutWaitForever`
: Never timeout—wait forever.

`kTimeoutPoll`
Immediate timeout; the calling function returns immediately. You use this if you need a resource but you aren't willing to wait for it.

`kTimeout1Second`
Timeout for one second. Convenient—and only 11 keystrokes longer than typing "1000".

KAL Startup Functions

The following functions are invoked during the boot sequence to initialize and start the kernel. As mentioned above (and as we'll constantly remind you, below), you never call these functions yourself; and, like the rest of the KAL, you can't reimplement them. They're described here in the interest of satisfying the curious.

KALInit Function

Purpose Initializes the kernel. The other KAL functions are no-ops until `KALInit` has been called. Called during the boot sequence—you never invoke this function yourself.

Declared In `Kernel.h`

Prototype `void KALInit ( void )`

KALStart Function

Purpose Starts the main kernel task running (which enables multi-tasking), and then starts an additional task. Called during the boot sequence—you never invoke this function yourself.

Declared In `Kernel.h`

Prototype `void KALStart ( void *additionalTaskProc )`

Parameters `additionalTaskProc`
Entry point for the "additional task." In its invocation during the boot sequence, the entry point is `PalmOSMain`—this is the function that start the Palm OS running (and which function never returns).

20

Tasks

A task is a thread of execution. Every task has its own stack and execution context. All tasks run pseudo-concurrently; the kernel schedules the execution of a task based on its numeric **priority** (as compared to other tasks) and its **task state**. The task state signifies whether the state is ready to run, sleeping, waiting for some other object (such as a semaphore or mutex), and so on.

Creating, Starting, and Stopping a Task

New tasks can be created through the `KALTaskCreate` function. After it's created, the task is "dormant" until told to run through `KALTaskStart`. The task executes its "entry point function" and we're off to the races.

If the task reaches the end of its entry point function (i.e. if the entry point function returns) the task exits and returns to the dormant state. The task can then be restarted through another `KALTaskStart` call, or you can delete it through `KALTaskDelete`. While it's running, you can force a task to exit by calling `KALTaskExit`; as with "natural" exiting, forcing a task to exit places it in the dormant state, from which it can be restarted or deleted.

Synchronizing Tasks

If you want your task to play with other tasks, you'll need to synchronize their operations. For example, if you have a task that writes a buffer of data, and another that reads the buffer, you'll need to make the reader task wait until the writer task has filled the buffer, and then make the writer wait until the reader has "emptied" it.

Most of the other KAL objects let you perform this sort of synchronization. In particular, mutexes and semaphores are designed specifically to synchronize tasks. In general, if you have a synchronization issue you should try to use mutexes and semaphores to solve it. (Event groups and mailboxes can also be used, although these are slightly more complicated objects.)

Nonetheless, the task API provides functions that can manipulate a task's state directly. These are described below.

Delaying

You can delay a (running) task's operation by calling `KALTaskDelay`. This causes the task to pause for some number of milliseconds before proceeding. When it's finished delaying, the task resumes where it left off. This can be useful if your task is running in a tight polling loop, although you should be careful not to abuse the technique: Most polling loops are much more efficient when controlled by a semaphore.

Wait and Wake

You can tell a task to wait (`KALTaskWait`) until it's signalled by some other task to wake up (`KALTaskWake`). The wait-and-wake system is a sort of "cheap semaphore." There are two essential features in this system:

- A task can be told to wake up before it's told to wait; waking a "non-waiting" task causes the task to increment its **wakeup count**. If a task has a positive wakeup count when it's told to wait, it continues without waiting (and decrements the wakeup count).
- A calling task can only tell itself to wait—you can't call `KALTaskWait` on some other task. This guarantees that the waiting task isn't already waiting.

As an example of how you might use wait-and-wake, let's say you have two tasks: task **J** and task **K**. You want to synchronize them such that task **J** waits at point **X** until task **K** has gotten to point **Y** (and vice versa). A simple way to do this is to have each task tell the other to wake up, and then immediately (the task itself) goes to sleep. The code would look something like this (without error checking and other details):

```
/* Task J code */
// we're running and running and running...
// and then we get to 'Point X':
KALTaskWake(taskK);
KALTaskWait(...);
```

The task **K** code looks exactly the same, modulo the function argument:

```
/* Task K code */
// we're running and running and running...
// and then we get to 'Point Y':
KALTaskWake(taskJ);
KALTaskWait(...);
```

Let's assume task **J** gets to point **X** first (i.e. before **K** gets to **Y**). Here's what happens:

1. **J** calls `KALTaskWake` to tell **K** to wake up; since **K** isn't waiting, the call increments **K**'s wakeup count.
2. **J** then calls `KALTaskWait`, and so stops its own execution.
3. **K** finally gets to point **Y** and calls `KALTaskWake` on **J**, which causes **J** to return from `KALTaskWait`.
4. **K** uses its positive wakeup count to "side step" its own `KALTaskWait` (the call doesn't go away, **K** enters it and then immediately returns).

Of course, it's possible that task **K** will get to point **Y** between steps 1 and 2. This isn't a problem: Both tasks will have positive wakeup counts by the time they get to their respective `KALTaskWait` calls.

A simpler example, in which task **J** regulates task **K** without reciprocity, is possible—but the example above points out an important caveat: If you're using a reciprocal wait-and-wake (as in the example), you *must* call wake before calling wait.

Another important point to keep in mind is that a task can't increment it's own wakeup count: A task can't call `KALTaskWake` on itself.

Suspending and Resuming

A final task manipulation mechanism is "suspend and resume." You can interrupt a task's operation by calling `KALTaskSuspend`, and then tell it to resume from where it was suspended by calling `KALTaskResume`. This sounds similar to wait-and-wake, but there's an important difference:

- A task can't suspend itself, thus there's no guarantee that the suspended task won't already be waiting (or suspended).
- Resumptions can't be "pre-issued." If you call `KALTaskResume` on a task that's isn't suspended, the call is (essentially) ignored.

Suspending and resuming can be useful when you're prototyping, debugging, or profiling code. You should never use suspend-and-resume to simulate a semaphore, or to otherwise control "real" code.

Tasks Lists

At any time, a task is either running, or it's on one of three **task lists**:

- The **ready list** contains tasks that are waiting for their turn to run. The list is ordered by task priority.
- The **wait list** contains tasks that are blocked in a function call, waiting for a "signal" in order to continue. The signal can be the release of a semaphore or mutex, the arrival of a mailbox message, the invocation of a `KALTaskWake` call, and so on. When the signal arrives, the task is moved to the ready list.
- The **suspended list** contains tasks that have been suspended through a `KALTaskSuspend` call. As mentioned above, a suspended task is "unsuspended" through a `KALTaskResume` call. When a suspended task is resumed, it's moved back to whichever list it was on when it was suspended.

The task lists are created and managed by the kernel. Moving tasks between these lists is the kernel's business; nonetheless, you should be aware of the lists as concepts and terms.

Priorities and Scheduling

A task's numeric priority, set when the task is created, is a measure of the task's "urgency" as compared to all other tasks. The "most urgent" task in the ready list is the one that the kernel scheduler, when it needs to choose a new task, will promote to running status. (If the currently running task has a higher priority than the tasks in the ready list, that task will continue to run.)

Priority numbers fall in the range [1, 255], where 1 is the "most urgent" priority, and 255 is the "least urgent." For the purposes of this documentation, a "higher priority" means greater urgency—which means a numerically smaller priority value.

The priority values themselves are meaningful only in (boolean) comparison to other tasks' priorities. Thus, the difference between priority value 1 and priority value 255 has the same significance as the difference between, say, 254 and 255: That one priority is "higher" (i.e. more urgent) than another is the only thing that matters.

Because the kernel only cares about finding the highest priority value, it's possible for a high priority task to "starve" lower priority tasks. The scheduler isn't "fair"—it doesn't auto-degrade a task's priority as the task receives cycles, nor does it dole out cycles in proportion to the tasks' priorities. It's up to the task creator to be responsible about setting an appropriate priority value:

- The default priority is 100. This is used for applications or other high level modules for which tasks are created automatically (in other words, in situations where the module doesn't get to specify a priority).
- Some of the Palm OS 5 system task priorities are:
  - Hot Sync: 90
  - Telephony: 80
  - Net: 60
  - Infrared: 50
  - System timer: 5
- Priority 0 is reserved for use by the system.

If in doubt, you should stick with priority 100. For background tasks, choose a lower priority (i.e. > 100). If you're creating a server, you may want a higher priority (< 100).

Task Structures, Constants, and Types

Task State Constants Function

Purpose Constants that represent the various states a task can be in.

Declared In `Kernel.h`

Constants

`kTaskStateRunning`
The task is currently executing.

`kTaskStateReady`
The task is "running" in the sense that it isn't waiting, suspended, or dormant, but it isn't currently executing. Instead, it's queued on the ready list, waiting for its turn to execute.

`kTaskStateWaiting`
The task is blocked in a function, waiting for some condition such as timer expiration, semaphore or mutex release, and so on. The task is currently on the wait list; when the condition holds, it will be moved from the wait list to the ready list.

`kTaskStateSuspended`
The task was running (or on the ready list) when it was suspended through a `KALTaskSuspend` call. The only way to "unsuspend" it is to call `KALTaskResume`. When it's resumed, the task is moved back to the ready list.

`kTaskStateWaitSuspend`
The task was suspended (through `KALTaskSuspend`) while it was on the wait list. When it's resumed (through `KALTaskResume`), it's moved back to the wait list or, if the wait condition was met while the task was suspended, it's moved to the ready list.

`kTaskStateDormant`
The task is freshly created and is waiting to be told to run (`KALTaskStart`), or it has already run and has been told to

exit (KALTaskExit). While it's dormant the task does nothing.

Comments A task's state is recorded in the `taskState` field of its `KALTaskInfoType` structure, which you can retrieve through the `KALTaskGetInfo` function. You can look at `taskState` for debugging or profiling purposes, but you should *never* predicate "real" code based on a task's state. The state is constantly changing; by the time you return from `KALTaskGetInfo`, the task may have changed state.

Task Wait Cause Constants Function

Purpose Constants that represent the various reasons that a task is waiting (i.e. its state is `kTaskStateWaiting` or `kTaskStateWaitSuspend`).

Declared In `Kernel.h`

Constants

`kTaskWaitCauseWait`
The task was told to wait by a `KALTaskWait` call.

`kTaskWaitCauseDelay`
The task is waiting for a delay time limit to expire. See `KALTaskDelay`.

`kTaskWaitCauseEventGroup`
The task is waiting for an event group to match the task's event pattern. See Chapter 23, "Event Groups."

`kTaskWaitCauseSemaphore`
The task is waiting for a semaphore to be released. See Chapter 21, "Semaphores."

`kTaskWaitCauseMailbox`
The task is trying to write to a full mailbox, or read from an empty mailbox. See Chapter 24, "Mailboxes."

`kTaskWaitCauseMutex`
The task is waiting for a mutex to be released.See Chapter 22, "Mutexes."

KALTaskCreateParamType Function

Purpose Repository for information that you supply when you create a new task.

Declared In `Kernel.h`

Prototype

```
typedef struct KALTaskCreateParamType {
     void  *exinf;
     KALTaskProcPtr  taskProc;
     UInt32  stackSize;
     UInt32  priority;
     UInt32  tag;
   } KALTaskCreateParamType;
```

Fields

`exinf`
: A pointer to "extended information" that can be supplied to the task. The data that exinf points to is passed as an argument to the task's entry point function.

`taskProc`
: The task's entry point function. This is the function that's called when the task is told to start ([`KALTaskStart`]).

`stackSize`
: The desired size of the task's stack, in bytes.

`priority`
: The desired priority of the task, in the range [1, 255], where 1 is the highest (most urgent) priority, and 255 is the lowest. For more information on priorities, see "[Priorities and Scheduling]."

`tag`
: A caller-defined identifier for the task. The tag value is recorded in the task's info structure ([`KALTaskInfoType`]) but is otherwise unused by the system.

KALTaskInfoType Function

Purpose Structure that contains everything that's known about a task.

Declared In `Kernel.h`

Prototype

```
typedef struct KALTaskInfoType
   {
     void  *exinf;
     KALTaskProcPtr  taskProc;
     void  *stack;
     UInt32  stackSize;
     UInt32  priority;
     UInt32  tag;
     KernelID  waitID;
     UInt16  waitCause;
     UInt8  taskState;
     UInt8  wakeupCount;
     UInt8  suspendCount;
   }  KALTaskInfoType;
```

Fields

`exinf`
: Pointer to “extended information.” This is arbitrary data that was (allocated and) defined by the task creator when the task was created. See `KALTaskCreate` for more information.

`taskProc`
: The task’s entry point function. This is the code that the task executes when it starts running.

`stack`
: A pointer to the task’s stack. Stacks grow down, so this is (numerically) the highest address value in the stack.

`stackSize`
: The size of the stack, in bytes.

`priority`
: The task’s execution priority, relative to other tasks. See “Priorities and Scheduling,” above, for more information on priority values.

`tag`
: The task’s ID number.

`waitID`
: If the task is waiting (for a timer, mutex, semaphore, etc.) this is the ID of the object it’s waiting on.

`waitCause`
: A constant that represents the type of condition the thread is waiting on.

`taskState`
: A constant that represent's the task's state—running, sleeping, waiting, etc. See "Task State Constants" for a list of constants.

`wakeupCount`
: The number of "wakeup" requests that are queued for this task. A wakeup undoes the effect of `KALTaskWait` (which see for details).

`suspendCount`
: The number of suspend requests that are queued for this task. Each suspend (`KALTaskSuspend`) must be balanced by a resume (`KALTaskResume`

Comments Every task has a KALTaskInfoType structure associated with it. The structure is created and maintained by the kernel. Although you can retrieve a task's info structure through `KALTaskGetInfo`, you can't change the data in the structure, and you should rarely need to use the information except for purposes of debugging or profiling.

The values for those few fields that are defined by the task creator—priority, extended info, entry point function—are all supplied through the `KALTaskCreateParamType` structure that's passed to the `KALTaskCreate` function.

KALTaskProcPtr Function

Purpose Protocol for task entry point functions.

Declared In `Kernel.h`

Prototype `typedef void (*KALTaskProcPtr) (void *);`

Comments The data for the argument is supplied in the `KALTaskCreate` call.

Task Functions

KALTaskCreate Function

Purpose Creates a new task.

Declared In `Kernel.h`

Prototype
```
Err KALTaskCreate (KernelID *taskID,
    const KALTaskCreateParamType *params)
```

Parameters

← *taskID*
: Integer that uniquely identifies the task.

→ *params*
: Caller-supplied information that's used to create the task. See `KALTaskCreateParamType` for details.

Returns

`errNone`
: Success.

`kKALErrNoFreeResource`
: All task objects are currently being used.

`kKALErrNoFreeRAM`
: Not enough memory to allocate the task's resources.

`kKALErrBadParam`
: Non-existent `params` argument, the priority value specification is 0, or the stack size specification is 0.

Comments When the function (successfully) returns, the new task will be in the dormant state. To tell the task to start operating, call `KALTaskStart`.

KALTaskDelay Function

Purpose Blocks for some amount of time, thus delaying the calling task.

Declared In `Kernel.h`

Prototype
```
Err KALTaskDelay (Int32 delayInMs)
```

Parameters

→ *delayInMs*
: The amount of time to block, in milliseconds.

Returns

`errNone`
Success.

`kKALErrBadParam`
delayInMs is less than 0.

`kKALErrInvalidContext`
The task is currently locked into the CPU (see `KALTaskSwitching`).

`kKALErrInvalidID`
The taskID value is out-of-bounds.

`kKALErrObjectNotExist`
taskID doesn't identify a task object.

Comments The countdown begins immediately, and continues even if the task is suspended. When the countdown reaches 0, the task continues execution, or waits to be resumed (if it's still suspended).

Delaying a task can be useful if it's not abused. See "Synchronizing Tasks" on page 165 for more information on delaying (and alternatives).

KALTaskDelete Function

Purpose Exits a running task (if necessary) and then destroys it.

Declared In `Kernel.h`

Prototype `Err KALTaskDelete (KernelID taskID)`

Parameters

→ *taskID*
The ID of the task that you want to destroy.

Returns

`errNone`
Success.

`kKALErrInvalidID`
The taskID value is out-of-bounds.

`kKALErrObjectNotExist`
taskID doesn't identify an extant object.

Comments You can call this function on any task, regardless of its current state. It's legal for a task to call this function on itself. In this case, the function doesn't return.

In general, you should avoid deleting tasks that aren't already dormant. It's the caller's responsibility to ensure that the targeted task is in a "clean" state before deleting it. For example, if the task is holding a mutex, `KALTaskDelete` does *not* release the mutex.

KALTaskExit Function

Purpose Stops a task, but doesn't dispose of it entirely. An exited task is in the dormant state.

Declared In `Kernel.h`

Prototype `Err KALTaskExit (KernelID taskID)`

Parameters →*`taskID`*
: The ID of the task that you want to stop.

Returns `errNone`
: Success.

`kKALErrObjectInvalid`
: The task is already dormant.

`kKALErrInvalidID`
: The taskID value is out-of-bounds.

`kKALErrObjectNotExist`
: taskID doesn't identify an extant object.

Comments After successfully exiting the task, you can restart it (`KALTaskStart`) or throw it away (`KALTaskDelete`).

A task can call `KALTaskExit` on itself. In this case, the function never returns.

Except for already-dormant tasks, any task (i.e. in any state) can be told to exit. The task's current state is "forgiven." For example, if the task is blocked, waiting for a semaphore, the task is removed from the semaphore's task list queue. If you then restart the task, the task *isn't* re-inserted into the semaphore's queue. Similarly for suspended tasks.

In general, tasks should be allowed to exit naturally rather than by being told to exit through this function. A task's entry point function should be designed to return on its own (which puts the task into the dormant state).

KALTaskGetCurrentID Function

Purpose Returns the ID of the calling task.

Declared In `Kernel.h`

Prototype
```
Err KALTaskGetCurrentID (KernelID *taskID)
```

Parameters

← *`taskID`*
Reference argument that returns the calling task's ID number.

Returns The function always returns `errNone`. If you call this function from a "task-independent" task, such as the system timer task, `taskID` is set to 0 (which is always an invalid kernel ID number). You're not allowed to access such tasks.

KALTaskGetInfo Function

Purpose Returns information about a specific task. Provided for debugging and profiling only.

Declared In `Kernel.h`

Prototype
```
Err KALTaskGetInfo (KernelID taskID,
   KALTaskInfoType *taskInfo)
```

Parameters

→ *`taskID`*
The ID of the task that you want information about.

← *`taskInfo`*
Structure that returns the information. See `KALTaskInfoType` for details.

Returns

`errNone`
Success.

`kKALErrInvalidID`
The taskID value is out-of-bounds.

`kKALErrObjectNotExist`
taskID doesn't identify a task object.

KALTaskResume Function

Purpose Wakes up a task that was suspended through `KALTaskSuspend`.

Declared In `Kernel.h`

Prototype `Err KALTaskResume (KernelID taskID)`

Parameters → *`taskID`*
The ID of the (suspended) task that you want to resume.

Returns `errNone`
Success.

`kKALErrObjectInvalid`
The task isn't currently suspended.

`kKALErrInvalidID`
The taskID value is out-of-bounds.

`kKALErrObjectNotExist`
taskID doesn't identify a task object.

Comments The task is resumed only if its suspension count has dropped to 0. (Nested calls to `KALTaskSuspend` must be balanced by an equal number of calls to `KALTaskResume`.)

KALTaskStart Function

Purpose Starts a dormant task running.

Declared In `Kernel.h`

Prototype
```
Err KALTaskStart (KernelID taskID,
   void *taskProcArg)
```

Parameters → *`taskID`*
The ID of the task that you want to shove into action.

← *`taskProcArg`*
Variable-length data that's passed to the task's entry point function.

Returns `errNone`
Success.

`kKALErrObjectInvalid`
The task isn't dormant.

`kKALErrInvalidID`
: The taskID value is out-of-bounds.

`kKALErrObjectNotExist`
: taskID doesn't identify an extant object.

Comments If successful, this function moves the task to the ready list. When it's chosen to be executed (by the kernel scheduler), the task begins operation by executing the entry point function that was specified in the `KALTaskCreate` function. When (and if) the entry point function returns, the task is put back into the dormant state, whence it can be restarted or deleted.

Note that if you're restarting a task that has already run and (was forcibly) exited, the task's previous state (when it exited) is forgotten. Specifically, if the task was waiting or suspended when it was forced to exit, the task is *not* returned to the wait list or suspended list when it's told to restart.

KALTaskSuspend Function

Purpose Suspends a (non-dormant) task's execution. The task remains suspended until `KALTaskResume` is called.

Declared In `Kernel.h`

Prototype `Err KALTaskSuspend (KernelID taskID)`

Parameters →*`taskID`*
: The ID of the task you want to suspend.

Returns `errNone`
: Success.

`kKALErrObjectInvalid`
: The task attempted to suspend itself, or the task is dormant.

`kKALErrQueueOverflow`
: The task's "suspension count" is already at the maximum value.

`kKALErrInvalidID`
: The taskID value is out-of-bounds.

`kKALErrObjectNotExist`
: taskID doesn't identify a task object.

Comments Suspending a task causes it to drop whatever it's doing—even if it's just waiting—and move over to the suspended list until it's told to resume through `KALTaskResume`. The only tasks you can't suspend are dormant tasks, and the calling task.

You can suspend an already-suspended task; each suspension must be balanced by an equal number of resumptions before the task is returned to the land of the animated.

Suspending tasks is discouraged. It's useful while you're protoyping an application, but it should very rarely be used in real code. See" Suspending and Resuming," above, for more information.

KALTaskSwitching Function

Purpose Enables and disables the calling task's ability to be context switched.

Declared In `Kernel.h`

Prototype `Err KALTaskSwitching ( UInt8 enable )`

Parameters →*`enable`*

If `false`, the calling task is locked into the CPU (it can't be switched out); if `true`, it's unlocked (it can be switched out).

Returns Currently, the function always return 0.

Comments By disabling switching, a task (the calling task) "locks" itself into the CPU. No other task can interrupt the locked task's execution until it (the locked task) enables switching. Provided for the foolhardy, `KALTaskSwitching` should only be invoked when executing extremely critical sections, such as PDA-controlled laser surgery.

The locked task can't call any function, such as `KALTaskDelay`, or `KALSemaphoreWait`, that could put the task into a wait state. Such calls will return a `kKALErrInvalidContext` error code.

Note that interrupts are *not* disabled. If an interrupt handler tries to dispatch a system call that would context switch the locked task, the call is deferred until after the locked task is unlocked.

The locked task can make any number of `KALTaskSwitching( false )` calls while it's locked. These additional calls are *not* counted: A single subsequent

`KALTaskSwitching( true )` invocation will undo all of the locking calls.

KALTaskWait Function

Purpose Moves the calling task to the wait list. The task remains blocked in `KALTaskWait` until some other task wakes it through `KALTaskWake`.

Declared In `Kernel.h`

Prototype `Err KALTaskWait (Int32 timeoutInMs)`

Parameters →*timeoutInMs*

The maximum amount of time to wait for a `KALTaskWake` call. If the call doesn't arrive within `timeoutInMs` milliseconds, the task will wake up anyway.

Returns `errNone`

Success—some other task told this task to wake up.

`kDALTimeout`

The timeout expired.

Comments The task only blocks if its wakeup count is 0. A task that has a positive wakeup count isn't stopped by `KALTaskWait`, although the wakeup count is decremented.

Because `KALTaskWait` blocks until the task is awakened, the function (i.e. `KALTaskWait`) can't be nested.

See "Wait and Wake" on page 166 for more information about this function.

KALTaskWaitClr Function

Purpose Sets the calling task's wakeup count to 0, thus throwing away all pending wake-up calls.

Declared In `Kernel.h`

Prototype `Err KALTaskWaitClr ( void )`

Returns `errNone`

Success.

Comments See KALTaskWait, KALTaskWake, and "Wait and Wake" on page 166 for more information.

KALTaskWake Function

Purpose Wakes up a waiting task.

Declared In `Kernel.h`

Prototype `Err KALTaskWake (KernelID taskID)`

Parameters → *`taskID`*
The ID of the task you want to wake up.

Returns `errNone`
Success.

`kKALErrObjectInvalid`
`taskID` identifies the calling task, or a system "task-independent" task. Neither of these tasks is a valid target for this call.

`kKALErrInvalidID`
The taskID value is out-of-bounds.

`kKALErrObjectNotExist`
taskID doesn't identify a task object.

Comments If the targeted task isn't waiting, `KALTaskWake` increments the task's "wakeup count." See `KALTaskWait` and "Wait and Wake" on page 166 for more information about this function.

21
Semaphores

A semaphore acts as a key that a task must acquire in order to continue execution. Any task can attempt to "acquire" a semaphore through the KALSemaphoreWait function. The function blocks until the semaphore is actually acquired (or until it times out).

When a task acquires a semaphore, that semaphore (typically) becomes unavailable for acquisition by other tasks. The semaphore remains unavailable until it's "released" through a call to KALSemaphoreSignal.

A task that attempts to acquire an unavailable semaphore is placed at the tail of the semaphore's task wait queue where it sits blocked in the KALSemaphoreWait call. Each call to KALSemaphoreSignal unblocks the task at the head of that semaphore's queue. Tasks in the wait queue are sorted according to their task priorities.

The Semaphore Count

To assess "acquirability" during a KALSemaphoreWait call, a semaphore looks at its semaphore count. This is a counting variable that's initialized when the semaphore is created. Ostensibly, the semaphore count's initial value is the number of tasks that can acquire the semaphore at a time. (As we'll see later, this isn't the entire story, but it's good enough for now.) For example, a semaphore that's used as a mutually exclusive lock takes an initial semaphore count of 1—in other words, only one task can acquire the semaphore at a time.

Calls to KALSemaphoreWait and KALSemaphoreSignal alter the semaphore's count: KALSemaphoreWait decrements the count, and KALSemaphoreSignal increments it.

When you call KALSemaphoreWait, the function looks at the semaphore count (before decrementing it) to determine if the semaphore is available:

- If the count is greater than zero, the semaphore is available for acquisition, so the function decrements the count and returns immediately.
- If the count is zero, the semaphore is unavailable, and the task is placed in the semaphore's task wait queue.

The initial semaphore count isn't an inviolable limit on the number of tasks that can acquire a given semaphore—it's simply the initial value for the semaphore's semaphore count variable. For example, if you create a semaphore with an initial semaphore count of 1 and then immediately call `KALSemaphoreSignal` five times, the semaphore's semaphore count will increase to 6. Furthermore, although you can't initialize the semaphore count to less-than-zero, an initial value of zero itself is common—it's an integral part of using semaphores to impose an execution order.

Although it's possible to retrieve the value of a semaphore's semaphore count (by calling `KALSemaphoreGetInfo`), you should only do so for amusement—while you're debugging, for example.

Semaphore Structures and Constants

KALSemaphoreInfoType Function

Purpose Structure that describes a specific semaphore.

Declared In `Kernel.h`

Prototype

```
typedef struct KALSemaphoreInfoType {
      KernelID  waitTask;
      UInt16   count;
      UInt16   initialCount;
    } KALSemaphoreInfoType;
```

Fields

`waitTask`
: The task that's at the head of the semaphore's task wait queue (or 0 if none).

`count`
: The semaphore's current count.

`initialCount`
The semaphore count that was used to initialize the semaphore.

Semaphore Functions

KALSemaphoreCreate Function

Purpose Creates a new semaphore

Declared In `Kernel.h`

Prototype
```
Err KALSemaphoreCreate (KernelID *semaphoreID,
    UInt32 tag, UInt32 initialCount);
```

Parameters

←*semaphoreID*
Unique identifier that's created by the function and returned by reference to the caller

→*tag*
User-defined identifier; currently unused.

→*initialCount*
Initial semaphore count. This is the number of tasks that can acquire the (freshly minted) semaphore without blocking. An initial count of 0 means the semaphore must be released before it can be acquired. Valid count values are in the range [0, 0xffff].

Returns

`errNone`
Success.

`kKALErrNoFreeResource`
All semaphore objects are currently being used.

`kKALErrNoFreeRAM`
Not enough memory to allocate the semaphore's resources.

`kKALErrBadParam`
`initialCount` value is out-of-bounds.

KALSemaphoreDelete Function

Purpose Deletes a semaphore.

Declared In `Kernel.h`

Prototype `Err KALSemaphoreDelete (KernelID semaphoreID);`

Parameters → *`semaphoreID`*
The ID of the semaphore that you want to delete.

Returns `errNone`
Success.

`kKALErrInvalidID`
The semaphoreID value is out-of-bounds.

`kKALErrObjectNotExist`
semaphoreID doesn't identify an extant object.

Comments Any tasks that are in this semaphore's task wait queue are immediately released, and return with an error code of `kKALErrObjectDeleted`.

KALSemaphoreGetInfo Function

Purpose Returns information about a semaphore

Declared In `Kernel.h`

Prototype
```
Err KALSemaphoreGetInfo (KernelID semaphoreID,
   KALSemaphoreInfoType *semaphoreInfo)
```

Parameters → *`semaphoreID`*
The ID of the semaphore for which you want to retrieve information.

← *`semaphoreInfo`*
A structure that contains the information. See `KALSemaphoreInfoType` for details.

Returns `errNone`
Success.

`kKALErrInvalidID`
The semaphoreID value is out-of-bounds.

`kKALErrObjectNotExist`

`semaphoreID doesn't identify a semaphoreobject.`

Comments Provided for debugging and profiling information only. You should never predicate "real" code on the information in the `KALSemaphoreInfoType` structure.

KALSemaphoreSignal Function

Purpose "Releases" a semaphore, thus allowing it to be acquired by the highest priority task in the semaphore's task queue.

Declared In `Kernel.h`

Prototype `Err KALSemaphoreSignal (KernelID semaphoreID)`

Parameters →*`semaphoreID`*
The ID of the semaphore you want to release.

Returns `errNone`
Success.

`kKALErrQueueOverflow`
The semaphore count is already at the maximum.

`kKALErrInvalidID`
The semaphoreID value is out-of-bounds.

`kKALErrObjectNotExist`
semaphoreID doesn't identify an extant object.

Comments If there are any tasks waiting to acquire this semaphore, the highest priority task is released (it returns from its `KALSemaphoreWait` call with the value `errNone`). If there are no waiting tasks, the semaphore's semaphore count is incremented.

KALSemaphoreWait Function

Purpose Attempts to "acquire" a semaphore.

Declared In `Kernel.h`

Prototype
```
Err KALSemaphoreWait (KernelID semaphoreID,
    Int32 msTimeout)
```

Parameters →*`semaphoreID`*
The ID of the semaphore that you're attempting to acquire.

→*`msTimeout`*
The amount of time to wait for the acquisition, in milliseconds. See "KAL Timeout Constants" on page 162 for timeout constants that you can use (in addition to millisecond values).

Returns

`errNone`
Success.

`kDALTimeout`
The timeout expired.

`kKALErrBadParam`
Bad `msTimeout` value; only positive values and timeout constants are allowed.

`kKALErrInvalidContext`
You called this function from a task that has context switching disabled.

`kKALErrInvalidID`
The semaphoreID value is out-of-bounds.

`kKALErrObjectNotExist`
semaphoreID doesn't identify an extant object.

Comments

If the semaphore's count is greater than 0, the semaphore is immediately available: The count is decremented, and the task returns immediately. If the count is 0, the task is moved onto the semaphore's task wait queue.

22

Mutexes

A mutex is a “mutual exclusion” lock. It provides a simple means to protect non-reentrant code (or other critical code areas) from being executed by more than one task at a time.

A mutex is locked and unlocked through calls to `KALMutexReserve` and `KALMutexRelease`. A single mutex can be reserved by (or “held by”) only one task at a time; while a task holds a mutex, it’s called the mutex’s “owner.” A mutex can only be released by its owner.

While it holds a mutex, the owner can make additional `KALMutexReserve` calls. This increments the mutex’s **lock count**. To release the mutex, the owner must decrement the lock count to 0 through complementary calls to `KALMutexRelease`.

Calls made to `KALMutexReserve` by non-owning tasks block while a mutex is being held, and the wanna-be reservers are placed in the mutex’s **task wait queue**. The queue is sorted by task priority. When the mutex is released, the task with the most urgent priority is given ownership of the mutex.

Mutex Structures and Constants

KALMutexInfoType Function

Purpose Structure that contains information about a mutex.

Declared In `Kernel.h`

Prototype

```
typedef struct KALMutexInfoType {
      KernelID  holdingTask;
      KernelID waitTask;
    } KALMutexInfoType;
```

Fields

`holdingTask`
The task that currently owns the mutex (if any). This is the only task that's allowed to release the mutex.

`waitTask`
The highest priority task in the mutex's task queue. When the mutex is released by its present owner, this is the next task that will reserve the mutex.

Comments You can retrieve this structure through `KALMutexGetInfo`. The information in this structure is provided for debugging and profiling purposes only; never use this information to predicate "real" code.

Mutex Functions

KALMutexCreate Function

Purpose Creates a new, unowned mutex.

Declared In `Kernel.h`

Prototype `Err KALMutexCreate (KernelID *mutexID, UInt32 tag)`

Parameters

←*mutexID*
System-wide unique ID of the new mutex.

→*tag*
Caller-defined identifier for the mutex. Currently unused.

Returns `0`
Success.

`kKALErrNoFreeResource`
All mutex objects are currently being used.

`kKALErrNoFreeRAM`
Not enough memory to create another mutex.

KALMutexDelete Function

Purpose Deletes a mutex.

Declared In `Kernel.h`

Prototype `Err KALMutexDelete (KernelID mutexID)`

Parameters →*`mutexID`*
The ID of the mutex that you want to delete.

Returns `0`
Success.

`kKALErrInvalidID`
The mutexID value is out-of-bounds.

`kKALErrObjectNotExist`
mutexID doesn't identify an extant object.

Comments If there are any tasks waiting to reserve this mutex, they're immediately unblocked and return `kKALErrObjectDeleted`.

KALMutexGetInfo Function

Purpose Returns information about a mutex.

Declared In `Kernel.h`

Prototype
```
Err KALMutexGetInfo (KernelID mutexID,
   KALMutexInfoType *mutexInfo)
```

Parameters →*`mutexID`*
The ID of the mutex you want information about.

←*`mutexInfo`*
The dope.

Returns `errNone`
Success.

`kKALErrInvalidID`
The mutexID value is out-of-bounds.

`kKALErrObjectNotExist`
mutexID doesn't identify an extant object.

Comments Provided for debugging and profiling information only. You should never predicate "real" code on the information in the `KALMutexInfoType` structure.

KALMutexRelease Function

Purpose Releases a mutex. Only the mutx's owner can release a mutex.

Declared In `Kernel.h`

Prototype `Err KALMutexRelease (KernelID mutexID)`

Parameters →*`mutexID`*
The ID of the mutex you want to release.

Returns `errNone`
Success.

`kKALErrNotOwner`
The caller doesn't own the mutex.

`kKALErrInvalidID`
The mutexID value is out-of-bounds.

`kKALErrObjectNotExist`
mutexID doesn't identify an extant object.

Comments Each successful KALMutexRelease call decrements the mutex's lock count. If the mutex's lock count is decremented to 0 as a result of this call, the caller (owner) releases the mutex and ownership is passed to the highest priority task in the mutex's task wait queue. If there are no waiting tasks, the mutex goes unowned until the next `KALMutexReserve` call.

If the lock count remains greater than 0 (because the owner has made additional calls to `KALMutexReserve`), the caller retains ownership until it has made enough `KALMutexRelease` calls to decrement the lock count to 0.

KALMutexReserve Function

Purpose Tries to reserve the mutex.

Declared In `Kernel.h`

Prototype
```
Err KALMutexReserve (KernelID mutexID,
    Int32 msTimeout)
```

Parameters

→*mutexID*
The ID of the mutex that you want to reserve.

→*msTimeout*
The amount of time, in milliseconds, that you're willing to wait for the mutex to become available. See "KAL Timeout Constants" on page 162 for timeout constants that you can use (in addition to millisecond values).

Returns

`0`
Success

`kDALTimeout`
The timeout has expired.

`kKALErrInvalidContext`
You called this function from a task that has context switching disabled.

`kKALErrObjectDeleted`
The mutex was deleted while the caller was waiting for it.

`kKALErrBadParam`
Bad `msTimeout` value; only positive values and timeout constants are allowed.

`kKALErrQueueOverflow`
The mutex's lock count has hit the maximum allowed value.

`kKALErrInvalidID`
The mutexID value is out-of-bounds.

`kKALErrObjectNotExist`
mutexID doesn't identify an extant object.

Comments If the mutex is unowned, the caller becomes the owner and returns immediately.

If the caller is already the owner, the mutex's lock count is incremented (and the function returns immediately). To release the mutex, the owner must make an equal number of calls to

`KALMutexRelease` (i.e. equal to the number of `KALMutexReserve` calls it made).

If the mutex is owned, but not by the caller, the calling task is placed on the mutex's task queue. The task queue is sorted by task priority; when the mutex becomes available, the task with the highest priority becomes the mutex's new owner and returns from the `KALMutexReserve` call.

23

Event Groups

An **event group** is a set of conditions that a task can monitor. Each condition is represented, within the event group, as a single (ordered) **event bit**, where a value of 1 means the condition obtains (or "is set"). As the condition changes state, an agent must toggle the corresponding event bit by calling `KALEventGroupSignal` and `KALEventGroupClear`.

In the meantime, a task can block on the event group by calling `KALEventGroupWait`. When it calls `KALEventGroupWait`, the task specifies which events it's interested in (this is called the task's **wait pattern**); when these events become set, the task is unblocked. More than one task can wait on the same event group. Waiting tasks aren't queued; the only predicate for releasing a waiting task is whether the task's wait pattern is satisfied.

Each event group can contain as many as 32 events. The correspondences between "real world" conditions and (the order of) event bits is arbitrary. Creating an agent that will keep the event group up-to-date with regard to the state of the conditions is up to the caller.

Event Group Structures and Constants

Event Group Wait Mode Constants

Purpose Constants that help define a wait pattern.

Declared In `Kernel.h`

Constants `kEventGroupAll`

Used to specify that the wait pattern is satisfied only if *all* the events in the wait pattern are set.

`kEventGroupAny`
: Used to specify that the wait pattern is satisfied if *any* of the events in the wait pattern is set.

KALEventGroupInfoType Struct

Purpose Structure that contains information about an event group.

Declared In `Kernel.h`

Prototype

```
typedef struct KALEventGroupInfoType
   {
     KernelID  waitTask;
     UInt32  events;
   }  KALEventGroupInfoType;
```

Fields

`waitTask`
: The ID of the "first" task that's waiting on the event group. There may be more than one waiting task, but, currently, you can only retrieve the ID of the first one.

`events`
: The event group's event bits.

Comments You can retrieve an event group's info structure through `KALEventGroupGetInfo`. However, you can't change the info in the structure (directly), nor should you predicate your code based on the structure's values. The info structure is provided, primarily, for debugging and profiling purposes.

KALEventGroupWaitParamType Struct

Purpose Structure that encapsulates a waiting task's wait pattern. The structure is used as an argument to `KALEventGroupWait`.

Declared In `Kernel.h`

Prototype
```
typedef struct KALEventGroupWaitParamType
   {
      UInt32  waitPattern;
      UInt32  matchType;
      Int32  timeout;
   } KALEventGroupWaitParamType;
```

Fields

`waitPattern`
: Bitfield that specifies a task's wait pattern. These are the events that the task is interested in (and will block on until they're set).

`matchType`
: Either kEventGroupAll or kEventGroupAny; the former means that *all* the events in waitPattern must be set before the task is unblocked. The latter means that if *any* of the events in waitPattern is set, the task will be unblocked.

`timeout`
: The amount of time to wait for the wait pattern to hold, in milliseconds. Also see the timeout constants in "KAL Timeout Constants" on page 162.

Event Group Functions

KALEventGroupClear Function

Purpose "Unsets" (sets to 0) one or more event bits.

Declared In `Kernel.h`

Prototype
```
Err KALEventGroupClear (KernelID eventGroupID,
   UInt32 events)
```

Parameters

→*eventGroupID*
: The ID of the event group whose event bits you want to clear.

→*`events`*
: Bitmask that's `AND`'d onto the event bits.

Returns

`0`
: Success.

`kKALErrInvalidID`
: The eventGroupID value is out-of-bounds.

`kKALErrObjectNotExist`
: eventGroupID doesn't identify an extant object.

Comments

Since `events` is AND'd with the event group's event bits, any bits that aren't set in `events` will be cleared in the event bits.

Clearing bits won't cause waiting tasks to be released—tasks wait for event bits to be set, not for them to be cleared.

To set the event bits, use `KALEventGroupSignal`.

KALEventGroupCreate Function

Purpose Creates a new event group object and sets its initial state.

Declared In `Kernel.h`

Prototype

```
Err KALEventGroupCreate (KernelID *eventGroupID,
   UInt32 tag, UInt32 initialState)
```

Parameters

←*`eventGroupID`*
: System-wide unique ID that identifies the new event group.

→*`tag`*
: Caller-supplied identifier; currently unused.

→*`initialState`*
: Bitfield that sets the initial states of the event group's event bits.

Returns

`0`
: Success

`kKALErrNoFreeResource`
: All event group objects are currently being used.

`kKALErrNoFreeRAM`
: Not enough memory to create another event group.

KALEventGroupDelete Function

Purpose Deletes an event group, releasing all tasks that are blocked on the object.

Declared In `Kernel.h`

Prototype `Err KALEventGroupDelete (KernelID eventGroupID)`

Parameters →*`eventGroupID`*
ID of the event group you want to delete.

Returns `0`
Success.

`kKALErrInvalidID`
The eventGroupID value is out-of-bounds.

`kKALErrObjectNotExist`
eventGroupID doesn't identify an extant object.

Comments When an event group is deleted, the tasks that it's blocking are immediately unblocked and return with the error code `kKALErrObjectDeleted`.

KALEventGroupGetInfo Function

Purpose Returns a structure that describes the state of the event group.

Declared In `Kernel.h`

Prototype
```
Err KALEventGroupGetInfo ( KernelID eventGroupID,
   KALEventGroupInfoType *eventGroupInfo )
```

Parameters →*`eventGroupID`*
The ID of the event group you want information on.

←*`eventGroupInfo`*
Structure that encapsulates the event group info. See `KALEventGroupInfoType` for more info.

Returns `0`
Success.

`kKALErrInvalidID`
The eventGroupID value is out-of-bounds.

`kKALErrObjectNotExist`
eventGroupID doesn't identify an extant object.

Comments Provided for debugging and profiling information only. You should never predicate "real" code on the information in the `KALEventGroupInfoType` structure.

KALEventGroupRead Function

Purpose Returns the current state of the event group's event bits.

Declared In `Kernel.h`

Prototype
```
Err KALEventGroupRead ( KernelID eventGroupID,
    UInt32 *events )
```

Parameters

→*eventGroupID*
The ID of the event group whose event bits you want "read."

←*events*
Value that returns (a copy of) the event bits.

Returns

`0`
Success.

`kKALErrInvalidID`
The eventGroupID value is out-of-bounds.

`kKALErrObjectNotExist`
eventGroupID doesn't identify an extant object.

KALEventGroupSignal Function

Purpose Sets one or more event bits. This may cause waiting tasks to be released.

Declared In `Kernel.h`

Prototype
```
Err KALEventGroupSignal ( KernelID eventGroupID,
    UInt32 events )
```

Parameters

→*eventGroupID*
ID of the event group you want to modify.

→*events*
Bitmask that's `OR`'d onto the current event bits.

Returns

`0`
Success.

`kKALErrInvalidID`
The eventGroupID value is out-of-bounds.

`kKALErrObjectNotExist`
eventGroupID doesn't identify an extant object.

Comments Since the new events mask is `OR`'d onto the existing event bits, any event bits that are *already* set *remain* set—to "unset" a bit, use `KALEventGroupClear`.

After the new event mask is applied, all tasks that are currently waiting on this event group are examined to see if the change satisfies their (the waiting tasks') wait patterns. Satisfied tasks are unblocked.

KALEventGroupWait Function

Purpose Blocks on the event group until the specified wait pattern is satisfied.

Declared In `Kernel.h`

Prototype

```
Err KALEventGroupWait (KernelID eventGroupID,
   const KALEventGroupWaitParamType *patternSpec,
   UInt32 *events)
```

Parameters

→*`eventGroupID`*
ID of the event group you want (this task) to block on.

→*`patternSpec`*
Description of the wait pattern that must be satisfied before this task is unblocked (including a timeout).

←*`events`*
The state of the event group's event bits as the function returns (provided for debugging and profiling purposes).

Returns

`0`
Success.

`kKALErrObjectDeleted`
The event group was deleted while this task was waiting.

`kKALErrBadParam`
The `patternSpec->matchType` value is invalid (see `KALEventGroupWaitParamType` for valid values).

`kDALTimeout`
: The timeout specified by `patternSpec->timeout` expired.

`kKALErrInvalidID`
: The eventGroupID value is out-of-bounds.

`kKALErrObjectNotExist`
: eventGroupID doesn't identify an extant object.

24

Mailboxes

A mailbox is a task-independent message queue (FIFO) that tasks can use to pass data (or “messages”) to each other.

You place a message in a mailbox by calling `KALMailboxSend`. The message is placed at the end of the mailbox’s message queue. `KALMailboxWait` does the opposite: It pops the first message off queue and returns it to the caller.

If the message queue is empty when you call `KALMailboxWait`, the calling task is placed in the mailbox’s **task wait queue** where it waits until a message shows up. Tasks in the queue are ordered according to their priorities. When a message arrives, the highest priority (waiting) task gets the message. There’s no limit to the number of tasks that can wait on a mailbox.

Mailbox Messages

Mailbox messages are 32-bit numbers that, typically, are used to point to caller-managed data. Note that the mailbox doesn’t copy, free, or otherwise manage pointed-to data.

You can cast a mailbox message (value) to hold a `KernelID` number; this is a convenient way to broadcast a semaphore ID (or other object identifier) between tasks.

Mailbox Structures and Constants

KALMailboxInfoType Struct

Purpose Structure that contains information about a mailbox.

Declared In `Kernel.h`

Prototype

```
typedef struct KALMailboxInfoType
{
  KernelID  waitTask;
     void   *msg;
   } KALMailboxInfoType;
```

Fields

`waitTask`
: The ID of the task that's at the top of the mailbox's task wait queue. If there are no waiting tasks, this value is 0.

`msg`
: The message that's at the top of the message queue. This is the message that will be returned by the next `KALMailboxWait` call. If there are no messages in the queue, the field points to NULL.

Comments The KALMailboxInfoType structure is returned by the `KALMailboxGetInfo` function.

Mailbox Functions

KALMailboxCreate Function

Purpose Creates a new, empty mailbox.

Declared In `Kernel.h`

Prototype

```
Err KALMailboxCreate (KernelID *mailboxID,
   UInt32 tag, UInt32 depth)
```

Parameters

←*`mailboxID`*
: The system-wide unique ID of the new mailbox.

→*tag*
A caller-defined identifier for the mailbox.

→*depth*
The depth of the mailbox's message queue. This is the maximum number of (concurrent) messages the mailbox can hold. Must be at least 1.

Returns `errNone`
Success.

`kKALErrNoFreeResource`
All mailbox objects are currently being used.

`kKALErrNoFreeRAM`
Not enough memory to create another mailbox.

`kKALErrBadParam`
You specified a depth of 0. Mailbox depths must be greater than 0.

KALMailboxDelete Function

Purpose Deletes the mailbox and releases any tasks that are blocked on the mailbox.

Declared In `Kernel.h`

Prototype `Err KALMailboxDelete ( KernelID mailboxID )`

Parameters `mailboxID`
The ID of the mailbox you want to delete.

Returns `errNone`
Success.

`kKALErrInvalidID`
The mailboxID value is out-of-bounds.

`kKALErrObjectNotExist`
mailboxID doesn't identify an extant object.

Comments If there are any tasks waiting for a message to show up in this mailbox, they're immediately unblocked and return `kKALErrObjectDeleted`.

This function doesn't delete the data that mailbox's messages point to (keep in mind that the messages are usually pointers).

KALMailboxGetInfo Function

Purpose Returns information about the mailbox.

Declared In `Kernel.h`

Prototype
```
Err KALMailboxGetInfo ( KernelID mailboxID,
   KALMailboxInfoType *mailboxInfo )
```

→*mailboxID*
: The ID of the mailbox you want information on.

←*eventGroupInfo*
: Structure that encapsulates the mailbox info. See `KALMailboxInfoType` for more info.

Returns

`errNone`
: Success.

`kKALErrInvalidID`
: The mailboxID value is out-of-bounds.

`kKALErrObjectNotExist`
: mailboxID doesn't identify an extant object.

Comments Provided for debugging and profiling information only. You should never predicate "real" code on the information in the `KALMailboxInfoType` structure.

KALMailboxSend Function

Purpose Places a new messages at the tail of the mailbox's message queue.

Declared In `Kernel.h`

Prototype
```
Err KALMailboxSend ( KernelID mailboxID,
   const void *message )
```

Parameters

→*mailboxID*
: The ID of the mailbox into which you wish to gently deposit your message.

→*message*
: The message. Ostensibly, this is a pointer to some other data, although, since message is never de-referenced by the mailbox, you can use it to pass an actual 32-bit value. If you're passing a pointer, the pointed-to data will not be freed

by the mailbox. Determining who frees the data (sender or recipient—or nobody) is up to the mailbox users.

Returns
`errNone`
Success.

`kKALErrQueueOverflow`
The mailbox is full.

`kKALErrInvalidID`
The mailboxID value is out-of-bounds.

`kKALErrObjectNotExist`
mailboxID doesn't identify an extant object.

Comments If there are any tasks waiting on this mailbox, the task with the highest priority is given the newly-sent message.

KALMailboxWait Function

Purpose Retrieves a message from a mailbox. If there are no messages, the calling task is placed in the mailbox's priority-sorted task queue.

Declared In `Kernel.h`

Prototype
```
Err KALMailboxWait (KernelID mailboxID,
   void **message, Int32 msTimeout )
```

Parameters
→*mailboxID*
The ID of the mailbox into which you wish to gently deposit your message.

←*message*
A pointer to a buffer into which the message is copied.

→*msTimeout*
The amount of time to wait for a message to arrive, in milliseconds. See "KAL Timeout Constants" on page 162 for timeout constants that you can use (in addition to millisecond values).

Returns
`errNone`
Success

`kDALTimeout`
A message didn't show up within the specified timeout.

`kKALErrInvalidContext`
: You called this function from a task that has context switching disabled.

`kKALErrObjectDeleted`
: The mailbox was deleted while the caller was waiting

`kKALErrBadParam`
: Bad `msTimeout` value; only positive values and timeout constants are allowed.

`kKALErrInvalidID`
: The mailboxID value is out-of-bounds.

`kKALErrObjectNotExist`
: mailboxID doesn't identify an extant object.

25

Timers

The kernel creates, owns, and runs a **timer task** that's used to perform short operations. The timer task does this by stepping through its list of **timer objects**. Each timer contains a timestamp and a function pointer: The timestamp signifies when (in the future) the function will run: When the timestamp expires, the function is executed.

You augment the timer task's list of timers by creating a timer object (`KALTimerCreate`) and telling it when to run (`KALTimerSet`). Timers are non-periodic—after a timer's function is invoked, the timer sits and waits for another `KALTimerSet` invocation. You can simulate a perioidic timer by calling `KALTimerSet` from the timer's function. Thus, every time the function is executed, the timer is rescheduled.

Keep in mind that all timer functions are executed in the timer task's context. While it's executing a timer function, the timer task can't be context switched, and interrupts are disabled. Because of this, timer functions must execute in as few cycles as possible, and they can't make any calls to the timer task itself (function calls on the timer task are blocked).

Timer Structures, Constants, and Types

Timer State Constants

Purpose Constants that represent the two timer states (running and stopped).

Declared In `Kernel.h`

Constants `kTimerStopped`
The timer is stopped. This means that the either the timer is waiting for a `KALTimerSet` call, or the timer's function has already run.

`kTimerRunning`
The timer is running.

KALTimerInfoType Struct

Purpose Structure that contains information about a particular timer.

Declared In `Kernel.h`

Prototype

```
typedef struct KALTimerInfoType {
  UInt32  timeLeft;
  UInt8  timerState;
} KALTimerInfoType;
```

Fields `timeLeft`
The amount of time left in the function's timestamp, measured in milliseconds. In other words, this is the amount of time before the timer's function is invoked.

`timerState`
The state of the timer, either running (`kTimerRunning`) or not (`kTimerStopped`).

KALTimerProcPtr Typedef

Purpose Protocol for timer functions.

Declared In `Kernel.h`

Prototype `typedef void (*KALTimerProcPtr) (void *);`

Comments The data for the argument is supplied in the `KALTimerCreate` call.

Timer Functions

KALTimerCreate Function

Purpose Creates a new timer object.

Declared In `Kernel.h`

Prototype
```
Err KALTimerCreate (KernelID *timerID, UInt32 tag,
   KALTimerProcPtr timerProc, void *timerProcArg)
```

Parameters

← *timerID*
Returns the system-wide unique ID of the newly created timer. You use this value as a cookie for the other timer functions.

→ *tag*
Currently unused.

→ *timerProc*
A pointer to the timer's function. See `KALTimerProcPtr` for the prototype for this function.

→ *timerProcArg*
A pointer to data that will be passed as an argument to `timerProc`.

Returns

`0`
Success

`kKALErrNoFreeResource`
All timer objects are currently being used.

`kKALErrNoFreeRAM`
Not enough memory to create another timer.

Comments This creates the timer object, defines its function (and argument data), and adds the object to the timer task's list of timers. What it doesn't do is set the timer's timestamp or tell it to start running: To do that, you must call `KALTimerSet`.

The timer persists until you delete it through `KALTimerDelete`.

KALTimerDelete Function

Purpose Stops a timer, removes it from the kernel's timer list, and deletes it.

Declared In `Kernel.h`

Prototype `Err KALTimerDelete (KernelID timerID)`

Parameters →*`timerID`*
The ID of the timer you want to delete.

Returns `0`
Success.

`kKALErrInvalidID`
`timerID` doesn't identify a valid timer object.

`kKALErrObjectNotExist`
`timerID` doesn't identify an extant object.

Comments You can include this call in an implementation of a timer's function: A timer's function *is* allowed to delete the timer.

Note that timer ID values are recycled: When you delete a timer, its ID is available for use by a newly created timer. Thus, you should take care when caching time IDs.

KALTimerGetInfo Function

Purpose Returns information about a timer. Provided for debugging and profiling purposes; you should never predicate “real” code on the information returned here.

Declared In `Kernel.h`

Prototype
```
Err KALTimerGetInfo (KernelID timerID,
   KALTimerInfoType *timerInfo)
```

Parameters

→*timerID*
: The ID of the timer you want information about.

←*timerInfo*
: A structure that contains the information. See `KALTimerInfoType` for details.

Returns

`0`
: Success.

`kKALErrInvalidID`
: `timerID` doesn’t identify a valid timer object.

`kKALErrObjectNotExist`
: `timerID` doesn’t identify an extant object.

KALTimerSet Function

Purpose Starts or stops the timer.

Declared In `Kernel.h`

Prototype
```
Err KALTimerSet (KernelID timerID,
   UInt32 timestamp)
```

Parameters

→*timerID*
: The ID of the timer you want to affect.

→*timestamp*
: If greater than zero, the timer is started and the argument is the amount of time, in milliseconds, that the timer task waits before executing this timer’s function. If `timestamp` <= 0, the timer is immediately stopped.

Returns

`0`
: Success.

`kKALErrInvalidID`
: `timerID` doesn't identify a valid timer object.

`kKALErrObjectNotExist`
: `timerID` doesn't identify an extant object.

Comments If the timer is already running (and `timestamp` > 0), the timer is "stopped" before it's restarted.

You can include this call in an implementation of a timer's function. In other words, a timer's function is allowed to delete its timer.

Index

E

F

H

M

O

P