Springboard™ Development Guide for Handspring™ Handheld Computers

Release 1.14

Information herein is preliminary and subject to change without notice.

TRADEMARK ACKNOWLEDGMENT:

Document number: 80-0091-00

Handspring, Inc.

189 Bernardo Ave.

Mountain View, CA 94043-5203

TEL: (650) 230-5000

FAX: (650) 230-2100

www.handspring.com

Change History		
Revision	**Date**	**Description**
1.14	26 July 2001	**Handspring Developer Agreement: Updated Developer License Agreement.**
1.13	28 June 2001	**Module Setup Application:** Correction. The correct creator and type for a Setup application is 'HsCd' and 'HsSu' respectively. The card parameter is retrieved with HsCardAttrGet() not HsAppEventHandlerSet().
		Read Cycle: Footnote added. Springboard read cycle timing 4 specification (CPU expecting data after CSx* asserted) specifies a maximum of 300ns. However, some Handspring handheld computers can be configured with a longer access time. Access times beyond 300ns are specific to a handheld rather than part of the Springboard specification.
		Electrical Specifications, AC Characteristics: Clarification. The Springboard read and write timing parameters are specified with a maximum 50pF capacitance. Reset timing is specified with a maximum 33µF capacitance.
1.12	16 April 2001	**Compatibility Testing:** An updated guide to Springboard compatibility is located on the Handspring website.
		Springboard Trademarks and Logos: An updated guide to Springboard logos and usage guidelines are located on the Handspring website.
1.11	18 Oct 2000	**Springboard Interface Pinout and Signal Descriptions:** The microphone polarity was still reversed in revision 1.1 of this document. This document corrects the pin assignments.
1.1	16 Oct 2000	**Product Guides**: Information specific to a Handspring product has been moved to our Product Guides. Information on the cradle interface and any implementation details that may change from product to product are located here.
		Module Design Details, **Catching Module Removals:** This section has been updated to reflect implementation differences between Palm OS revisions.
		API Calls: Several API call definitions have been added to this section. Specifically, HSPrefSet() and HSPrefGet() support custom serial library redirection.
		Electrical Specifications, AC Characteristics: The Springboard read and write timing parameters have changed to accommodate the range of Handspring handheld products. Varying CPU speeds and design differences between products have been taken into account.

Table of Contents

1. Springboard Software Development

The key factor that makes the Springboard™ Expansion Slot a compelling platform is its *plug-and-play functionality*. The Springboard Expansion Slot allows different modules to be inserted and removed from a handheld computer at *any time*. To support this functionality, Handspring has created extensions to the standard Palm OS® in order to enable new, specialized hardware and software.

These extensions allow you to:

- Detect the insertion of a module.
- Load applications and appropriate drivers stored on the module.
- Cleanly remove software when the module is removed.

This functionality correctly implements plug-and-play for the handheld computer.

Application software that resides in a module's memory is "executed in place", just like applications in the device's internal ROM or RAM. With this design, the Palm OS jumps directly to program code, rather than "loading" an application into memory and "jumping" to the appropriate code. This execute in place architecture is well suited to handheld devices in which memory and processing power are scarce resources.

The Springboard Expansion Slot builds on this architecture by directly mapping the module into the CPU memory map. Access to hardware in the handheld and on the module is conducted in exactly the same way. To support full plug-and-play functionality, the system allows for removal of a module while it is running an application. The user is automatically switched out of the module application and back to the Application Launcher, as necessary.

Because the Springboard Expansion Slot is a direct extension of the CPU's parallel bus, modules can also be designed that contain specialized hardware to address new markets not being addressed by the handheld computer today (e.g., communications, entertainment, and professional markets).

This guide describes the Springboard Expansion Slot, and provides you with the information necessary to:

- Implement "application-only" products on a Springboard-compatible memory module.
- Design Springboard-compatible modules with specialized hardware and the applications to support them.

1.1. Software Development for Handspring Handhelds

There are three general categories of software development for Handspring handhelds. Each category has different requirements, as described in greater detail in the sections that follow. The three categories are:

Generic Applications: These applications execute from the Handspring handheld's internal memory.

> *Examples:* Any of the utilities, games, and other applications that can be downloaded for use on Palm OS-based devices.

Generic Applications on a Springboard Module: These applications execute from a Springboard memory module. Inserting a module provides instant access to the application, eliminating the need to download and install software.

> *Example:* A Palm OS game that is distributed on a Springboard module to accommodate retail distribution.

Special Purpose Applications: These applications access specialized hardware on the Springboard module. Additionally, a "special purpose" application may also install interrupt handlers and other system modifications in support of the module hardware. All the software necessary to operate the module is resident on the module itself, eliminating the need to download and install software and drivers.

> *Example*: Imaging software and drivers to operate a Springboard camera module.

1.2. Generic Applications

There are various development environments for Palm OS-based systems. The two primary tools available are the Palm OS GNU Tools and Metrowerks' CodeWarrior. Following is an overview of these tools. For more detailed information, or to download these tools, go to the Handspring website at:

http://www.handspring.com/developers/sw_dev.jhtml

1.2.1. Handspring Palm OS GNU Tools

GNU Tools are based on the UNIX environment and are based on a Bash shell command line driven system. If you're accustomed to UNIX development, this will be very familiar. To develop for Handspring handhelds you must use Handspring's GNU Tools, which use the Windows-compatible Cygnus Cygwin shell. Handspring has also developed a Microsoft Visual C++ project file that calls the GNU Tools. With this approach, you combine the ease of using an IDE with the flexibility of open source development tools. A complete description of the tools is included with the download. These tools are currently available for download from Handspring for use in the Windows environment.

1.2.2. Metrowerks CodeWarrior

Metrowerks CodeWarrior is an integrated development environment with a graphical interface that allows for easier generation of forms. It also provides a utility for automatically managing files and resources. The "lite" version -- which doesn't allow for software distribution -- is free to download. The full version can be purchased from Metrowerks. These tools are currently available from Palm for use in both the Windows and Macintosh environments. The Handspring Extension header files are available from the Handspring website and are included in the R6 version of CodeWarrior.

1.2.3. Other Development Environments

The Palm developers' tool web site (http://www.palmos.com/dev/tech/tools/) contains information on other development environments available for the Palm OS.

Developing software for Handspring handhelds is the same as developing for other Palm OS-based systems. For example, Visor™ is based on Palm OS 3.1, so documentation covering standard Palm OS development is applicable. The Palm developers' documentation web site (http://www.palmos.com/dev/tech/docs/contains a variety of references covering standard Palm OS development.

1.3. Generic Applications on a Springboard Module

If you are an application developer who wants to transfer your application to a non-volatile Springboard memory module, you simply need the Palm-MakeROM tool as described in the *Handspring Development Tools Guide* to build a ROM image. Third-party suppliers can use your ROM image to program memory modules in quantity.

For development purposes, you can also use Handspring's 8MB Flash Module. This module, available through our web site, is a run-time read-only memory-based module that can be re-programmed using the Palm Debugger. Details about the Palm Debugger can be found in the *Handspring Development Tools Guide*. Handspring includes an application (FileMover) with the module that enables users to transfer any application between internal and module memory. Handspring has also developed the `CardUpdaterMakerSDK` that provides an easy way for developers to generate a utility that customers can use for updating a Flash module.

Since the Springboard memory modules can be removed at any time during execution, there are some considerations to take into account when designing your application. Specifically, software that uses shared libraries or installs system modifications (e.g., interrupt handlers) should be configured properly to work with the plug-and-play features of the Springboard Expansion Slot. Refer to the section entitled *Application development to support plug-and-play* for a more detailed description.

You might want to consider designating a Welcome application on your module. This application would be launched automatically when your Springboard module is inserted in a handheld. Refer to the section entitled *Module Welcome application* for a description of the Welcome application.

Finally, applications that execute from a Springboard memory module (i.e., masked ROM, Flash, and OTP) are usually based on read-only memory and should be designed appropriately.

1.4. Special Purpose Applications

If you are building a Springboard module with special hardware, you must use the new Handspring API standard. This API (along with other necessary information to build Springboard-compatible modules) is explained in the next sections.

The Developer section of Handspring's web site contains source code examples for Springboard module applications that have been developed using the Handspring Palm OS GNU Tools. These examples show how to develop more sophisticated applications that install interrupt handlers or OS patches when the associated module is plugged into the Springboard Expansion Slot. These examples have been fully tested at Handspring and can be used as a baseline for application development.

1.5. Application Development To Support Plug-and-Play

The Springboard Expansion Slot supports true hot plug-and-play of removable modules. You can insert or remove a module at any time, regardless of the current state of the machine and regardless of which application is currently running. When a module is inserted, software resident on the module becomes immediately available. To enable this functionality, all modules must have a valid header generated by the Palm-makeROM utility, as described in the *Handspring Development Tools Guide*.

To support custom hardware on a Springboard module (e.g., the UART in a modem module), shared libraries and other system extensions are typically required. Handspring provides a mechanism for specifying a module *Setup application*. If it is present when the module is inserted, the Setup application is copied to internal memory, then executed. When the module is removed, the Setup application is executed again to handle removal of the module's software. The system then deletes the Setup application from internal memory.

If desired, the module manufacturer can also designate one of the applications in the ROM as a module *Welcome application,* which is automatically launched whenever the module is inserted (after the Setup application is run).

Generic applications are those that typically do not patch any system trap calls, install shared libraries, or install interrupt handlers. The vast majority of existing Palm OS applications fall into this category. Hot plug-and-play support works with most generic applications without modification. If a generic application is running on or using a module when it is removed, the system transparently and cleanly switches the user back to the Application Launcher.

Special-purpose applications are those that change system functionality by patching system trap calls, installing shared libraries, or installing interrupt handlers. These applications cannot support true hot plug-and-play unless the module contains a Setup application that is responsible for installing and removing the appropriate libraries, patches, and handlers. Without a Setup application, the system is forced to soft-reset the device when the module is removed to maintain stability. The *Special-Purpose Applications* section below describes how to design these types of applications to be fully compatible with hot insertion and removal.

1.5.1. Generic Applications

Generic Palm OS applications that are placed onto a memory module -- such as the 8MB Flash Module -- are supported by hot plug-and-play without modification. Depending on the application, there are rare instances in which the application might get "confused" when it is re-launched after a module is pulled out and re-inserted. Designing an application to avoid this possible problem is straightforward. The following precautions are good design practices, and ensure that an application can be successfully re-launched after any soft reset (which can occur any time a user presses the soft reset button).

There are two cases to consider when designing your application:

- The current application is executing out of module memory when the module is removed.
- The current application is using the module memory when the module is removed.

In these instances, the operating system is forced to abort the application and clean up any resources in use by the application. It does this by closing all databases that are still open. Normally a problem is not encountered unless the application was in the middle of writing out changes to a database it owns. If the module is pulled out during this time, the application's database could be left in a partially updated state. This could cause the application to be confused or even crash the next time it executes and re-uses that database. This window of vulnerability is very small in most applications; it typically occurs only after dismissing a dialog or choosing a menu item (e.g., to create or delete a new record). It is unlikely, although possible, that a user would pull a module out during this period of time.

For an application to protect itself from this potential problem, it must perform some simple checks whenever it re-opens a database it owns. For example, the application could set a valid bit in a record as the last step in updating the record. If the valid bit is not set when re-reading the record, the record can be automatically fixed or simply deleted. This process also ensures that the application can survive any soft reset.

1.5.2. Special-Purpose Applications

As mentioned above, products that extend system functionality by installing system patches, shared libraries, or interrupt handlers should not be placed on a removable module unless it includes a module *Setup application*. If there is no Setup application, the system is forced into a soft reset when the module is removed.

Typically, software of this nature is placed on a module in order to provide access to special hardware on it. The system looks for the Setup application when the module is installed and automatically copies it into the built-in RAM of the device. The system calls the Setup application with an "`install`" message. The Setup application can then install interrupt handlers, system patches, shared libraries, or whatever else is required in order to support the module and its hardware. All of the code to support these system extensions must either be present in the Setup application or initially copied into built-in RAM by the Setup application before being installed into the system.. This architecture is more fully described in *Module Insertion Notification.*

When a module is removed, the system again calls the Setup application with a "`remove`" message, giving the Setup application the opportunity to remove all hooks it had previously placed into the system. Note that this function call happens *after* the module is removed, because it is the actual removal that interrupts the OS, which in turn calls the Setup application. The module has typically been removed when the Setup application is called, so it must not be used to set the module's hardware in a certain mode. If this operation is needed, the module must do this operation itself. After the Setup application completes the remove operation, the system deletes the Setup application from the handheld's built-in memory.

The system patches, interrupt handlers, and libraries that are installed by a Setup application will often need to access special hardware devices on the module itself. Handspring provides calls that ensure that system extension code can gracefully detect and recover from a module removal at any time. These calls are more fully described in *Catching Module Removals.*

The system extension code installed by the Setup application may also need to install or process interrupts and handle power management functions. Handspring provides API calls (described in *API Calls*) to manage various aspects of interrupt and power management control.

2. Module Design Details

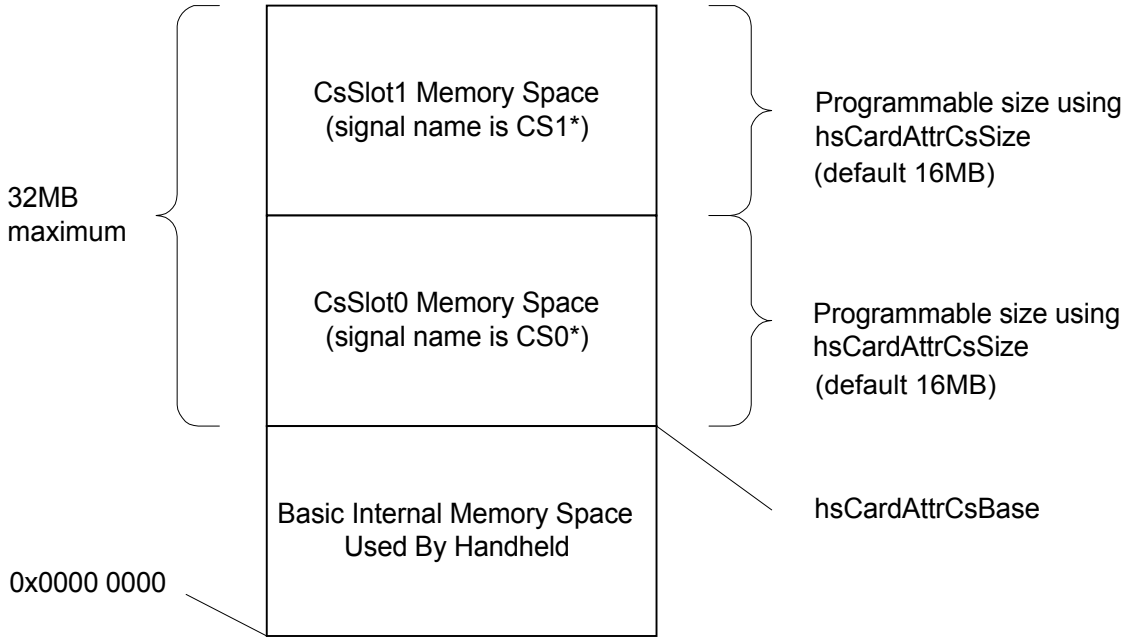
This chapter describes hardware aspects regarding memory space, interrupts, reset, power management, and insertion and removal of the modules. It also references the various Module Support API calls provided for interfacing to the module hardware; these calls are described in *API Calls.*

2.1. Module Memory Space

The figure below shows how memory within a Springboard module is mapped into the memory space of the handheld. Note that the base address and size may change in future products. You should use the hsCardAttrCsBase attribute of the HsCardAttrGet() call to obtain this information dynamically at run-time in order to ensure that your module software remains compatible with future revisions of the base unit and/or system software. The beginning of the module ROM is expected to be at this base address (0x2800 0000).

Two chip select lines, called CS0* and CS1* (* indicates an active low signal), are output to the Springboard module. By default, when a new module is inserted, the system assigns 16MB of address space to each chip select. The address space for CS0* is referred to as csSlot0; the address space for CS1* is referred to as csSlot1 (therefore, csSlot1 starts at csSlot0 + 16MB).

Figure 2-1. Module Memory Map



Each chip select is configured to address 16-bit wide memory devices. The ROM in the module must reside at the beginning of csSlot0 and must be 16 bits wide in order for the system to recognize the module. The system makes no assumptions about what resides at csSlot1.

You can use the HsCardAttrGet() and HsCardAttrSet() calls to query or change the csSlot0 and csSlot1 address ranges via the hsCardAttrCsSize attribute. The ranges can be set to any power of two between 128KB and 16MB, but both ranges must be set to the same size. csSlot1 always starts immediately after the csSlot0 range. If, for

example, csSlot0 start at address 0x28000000 and you change the size of the chip select address ranges to 1MB (100000h), then csSlot1 starts at address 0x28100000 and continues up to address 0x281FFFFF.

2.2. Module Access Time and Wait State

When a module is first inserted, the system accesses the ROM at csSlot0 with the maximum number of wait states allowed by the base unit hardware. Actual wait state definitions may change from system to system. Please refer to the appropriate Product Guide for wait state parameters for a specific handheld.

Once the system validates the ROM header, it reads a value out of the ROM header that indicates the actual required access time of the module in nanoseconds, and reprograms the number of wait states accordingly. The Palm-MakeROM tool places this value into the ROM header through the –tokStr parameter (see the *Handspring Development Tools Guide* for details). Both csSlot0 and csSlot1 must have the same number of wait states (a limitation imposed by the processor itself), so you must set this value to the worst case access time of your ROM and whatever other hardware is present on your module. The hsCardAttrAccessTime attribute of the HsCardAttrSet() call can also be used to change the number of wait states dynamically while the module is inserted. This method might be useful, for example, when temporarily accessing a slow device on your module.

Once the ROM has been validated, the system updates itself so that all applications and databases on the ROM are available to the system and other applications. All applications in the ROM appear in the Applications Launcher. If there is a Setup application on the module, it is copied in the handheld's internal memory and is sent an "`install`" message. Then, if the module has a *Welcome application*, this application is automatically launched as well.

2.3. Interrupts

IRQ* is a dedicated interrupt line that module hardware can assert to interrupt the CPU. This interrupt is level-sensitive and active low. The software for the module can install an interrupt handler for this interrupt using the hsCardAttrIntHandler attribute of HsCardAttrSet(). The module interrupt handler can be written in C or assembly language. A 32-bit reference parameter is passed to it that you specify in the hsCardAttrCardParam attribute of HsCardAttrSet() along with a Boolean flag named `*sysAwakeP` that is passed by reference. Typically the 32-bit reference parameter is a pointer to a variable which is part of an application's global data. Note that while global data is not accessible from an interrupt routine, referencing a variable (or structure) through this mechanism is allowed. The `*sysAwakeP` parameter usage is described below in *Interrupt Handler Interaction*.

In general, the interrupt handler must observe a number of restrictions that all Palm OS interrupt handlers observe. It cannot allocate, free, or move memory chunks, or allocate any system resources (e.g., semaphores, timers, and tasks). If asserted when the handheld is in sleep mode, this interrupt wakes it up unless `*sysAwakeP` is false.

The hsCardAttrIntEnable attribute can be used at any time to enable or disable module interrupts to the processor. It must be called after installing the interrupt handler for the first time since interrupts are disabled by default.

When the module is removed, the system immediately disables further module interrupts (using the hsCardAttrIntEnable attribute), and sends the "`remove`" message to the module's Setup application. The Setup application should then clear the hsCardAttrIntHandler attribute to `null` to remove the interrupt handler.

Note: The operating system will clear the hsCardAttrIntHandler automatically when the module is removed, however, it is better practice for the Setup application to do this itself for instances in which the module is inserted but not used.

2.4. Interrupt Latency

Interrupt latency is likely to vary from product to product. To support multiple generations of handhelds, we recommend that modules be designed to tolerate the interrupt latency times described below. These parameters

are three times greater than the measured interrupt latency of the Visor, Handspring's first product. For latency measurements of a specific product, please refer to the appropriate Handspring Product Guide.

The three conditions of interrupt latency include: 1) when the device is already awake, 2) the first time the interrupt handler is called after coming out of sleep mode but before the rest of the system is awake (the `*sysAwakeP` parameter is false), and 3) the second time the interrupt handler is called after coming out of sleep mode, which is after the rest of the system has awakened (the `*sysAwakeP` parameter is true).

In the third condition, the designer must allow for an *additional* 20 milliseconds if the OS needs to open the USB library. Note that this condition only occurs if the USB library was open when the device went to sleep. This situation is rare because most application will close the USB library before the device goes to sleep.

Table 2-1. Interrupt Latency Examples

Condition	Maximum Springboard Latency Specification
Device already awake	0.15 ms
Device asleep (*sysAwakeP == false)	4 ms
Device asleep (*sysAwakeP == true)	10 ms

2.5. Power Management

A module can provide software and hardware support for power management. A routine for taking the module into and out of low power mode can be installed through the hsCardAttrPwrHandler attribute. The operating system calls this routine whenever the handheld is turned on or off (that is, taken into or out of sleep mode). Parameters of the routine tell it whether to power up or down. If powering down, a second parameter indicates the reason. The reason code is either `hsCardPwrDownNormal` (a normal power down) or `hsCardPwrDownLowVoltage` (indicates that this is an emergency shutdown due to low or no battery voltage). Refer to the Handspring extension header file (HsExt.h) for more information on `hsCardPwrDownNormal` and `hsCardPwrDownLowVoltage`. The actual voltage levels associated with the reason codes may change from product to product. Please refer to the appropriate Product Guide for more details.

The power handler routine must observe the same restrictions as an interrupt handler because it might be called from the context of an interrupt routine, particularly when the system performs an emergency shutdown due to low battery voltage. In addition, a power handler must execute ***very*** quickly. When the batteries are removed, the power handler is executed using only the energy stored in the battery backup capacitor; thus it should do the minimal amount of work necessary to put the hardware into low power mode before returning. Ideally, this process involves simply setting or clearing a bit in one or two hardware registers.

The base unit asserts the LOWBAT* signal on the module when the batteries fall below a certain critical threshold voltage level. In the Visor handheld, power to the module is removed a few milliseconds after the batteries fall below this critical threshold. The LOWBAT* signal is only valid until power to the module is removed. When the batteries are replaced, the module is re-powered and a new initialization sequence occurs.

The handler routine is called first and puts the module into its low power state through software. However, LOWBAT* must be used to prevent the module from asserting its interrupt, IRQ*, so that the module does not attempt to wake up the device when the batteries are too low for operation.

2.6. Power Management Options For Interrupt Handlers

Various power-saving options are available to module interrupt handlers. Through return parameters and system calls that it makes, the interrupt handler can tell the system how much of the hardware to power up as a result of the interrupt and how long to stay awake before going back into sleep mode.

The `*sysAwakeP` parameter to the module interrupt handler is a Boolean flag passed by reference that tells the interrupt handler how much of the system is currently awake. If the device is asleep ("off" from a user's perspective) when the module interrupts the handheld, the system calls the module interrupt handler first before it wakes up any of the remaining hardware (e.g., sound, timers, keypad) and passes false to `*sysAwakeP`. Because the rest of the system is not yet awake, the interrupt handler cannot make any system calls at this stage, except for HsCardErrTry/ HsCardErrCatch. If the handler can process the interrupt at this stage, then it needs to clear the interrupt source and return. The system then immediately puts the system back into sleep mode. This procedure is the most power-efficient means of processing the interrupt because no other extra hardware is powered up, while limiting the interrupt handler to simple memory manipulations.

If the interrupt handler is called with `*sysAwakeP` set to false but needs to make one or more system calls (e.g., HsCardEventPost() or SysSemaphoreSet()), then it should set `*sysAwakeP` and return without clearing the source of the interrupt. When the system sees `*sysAwakeP` set upon return of the module interrupt handler, it continues the wake-up sequence and wakes the rest of the system (except for the LCD). Once the system wake-up is completed, it calls the module interrupt handler again with `*sysAwakeP` set true. At this stage, the interrupt handler is free to make any system calls that are normally valid from interrupt handlers.

Before the interrupt handler returns from being called with `*sysAwakeP` set true, it has the additional option of telling the system whether or not to turn on the LCD, and how long to stay awake before returning to sleep mode. To do this, the handler calls HsEvtResetAutoOffTimer(). If the handler does not call HsEvtResetAutoOffTimer() before exiting, the default behavior of the system is to return to sleep mode on the next call to EvtGetEvent() by the current application. Typically, this occurrence is on the order of tens or hundreds of milliseconds, depending on the current application's event loop.

The HsEvtResetAutoOffTimer() call takes two parameters: a `stayAwakeTicks` value and a `userOn` Boolean for controlling the LCD. The `stayAwakeTicks` tells the system the minimum amount of time to stay awake before going back to sleep mode. This value is specified in system ticks; however, keep in mind that the system checks the timer approximately every five seconds to verify if it needs to put itself in sleep mode. Passing (-1) tells the system to stay awake for the current auto-off setting indicated in the General Preferences panel. If the interrupt handler wants the LCD to turn on, it sets the `userOn` Boolean. The interrupt handler sets the `userOn` Boolean if it has just posted an event through HsCardEventPost() that results in an alert being displayed or other user interface activity.

2.7. Module Insertion Notification

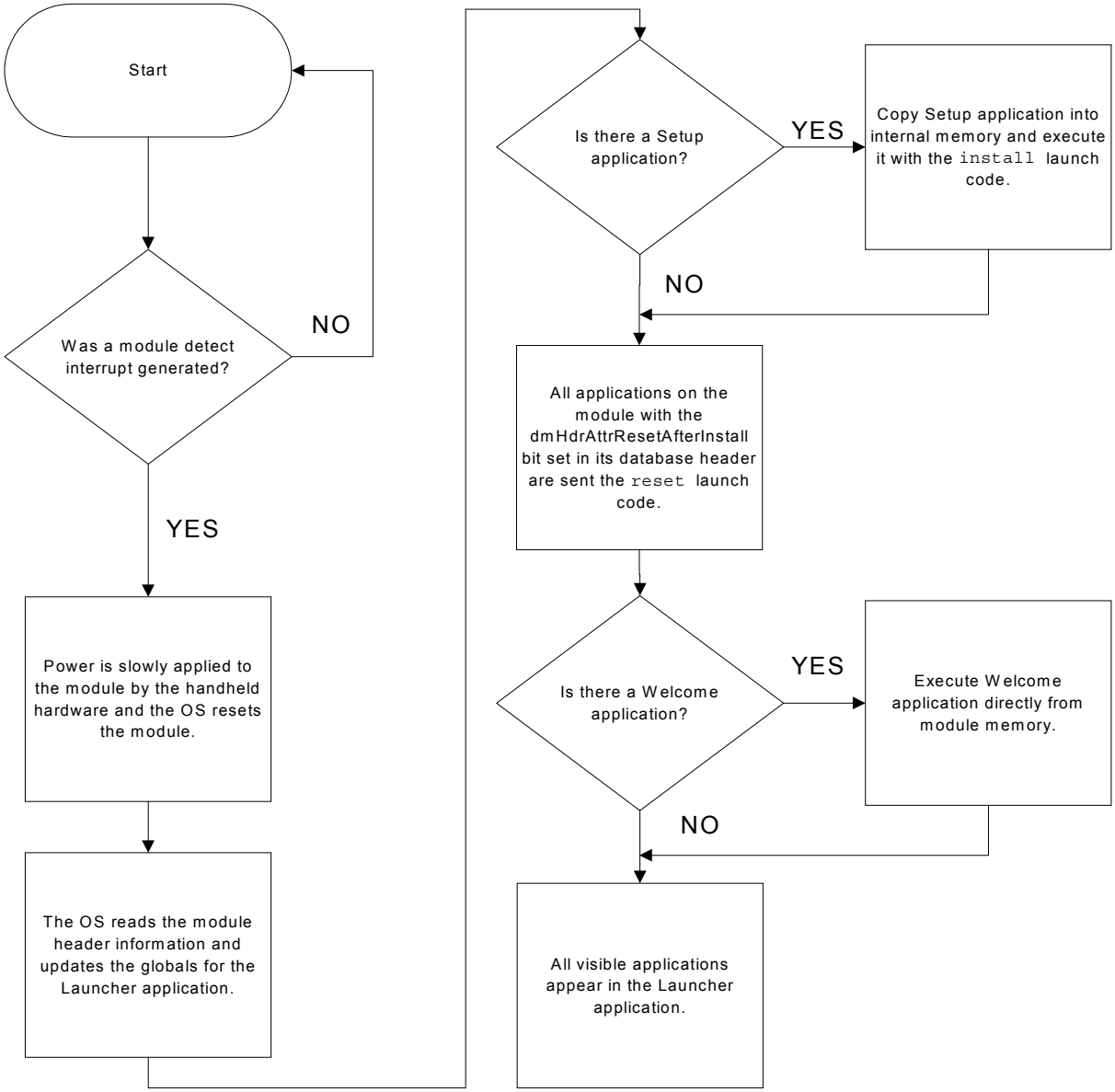
When a module is inserted, the handheld will attempt to read a valid header for up to ~3 seconds. If a module is already plugged in during a device reset, *all* applications and panels on the module are sent the standard Palm OS reset action code (`sysAppLaunchCmdSystemReset`), as are all normal built-in applications and panels.

When a module is hot-inserted at run time, the system also broadcasts the reset action code, but only to applications (not panels) on the module that have the `dmHdrAttrResetAfterInstall` bit set in their database header. This behavior ensures that applications requiring the reset notification are notified, without unnecessarily slowing down the module insertion process by calling every application and panel on a module when it is inserted.

This is a slight modification to the original purpose of the `dmHdrAttrResetAfterInstall` bit in the database header. Its original purpose was to tell HotSync® to soft reset the device after one or more of these "reset-after-install" applications are installed onto the device. During the reset sequence, all applications (and thus the ones just installed) are sent the reset action code. However, when present on a module application, this bit now indicates that the application wants to receive the reset action code after the module is inserted. Thus, the

application receives the reset action code, even though the system has not really gone through a soft reset. This particular behavior was chosen for maximum compatibility with most existing applications. Unfortunately a small subset of applications might be confused by it.

Figure 2-2. Operational Flowchart when Inserting a Module



2.8. Catching Module Removals

In all cases, when a module is removed, an interrupt is generated and the base address for the module chip selects are disabled. What happens next depends on the design of the currently executing application.

There are three cases to consider when a module is removed:

Case 1: An application is executing from handheld memory, and is not accessing module memory or any shared libraries resident on the module. If the Setup application was installed (copied into the handheld memory when the module was inserted), it is executed with a "`remove`" launch code. The system then deletes the Setup application and control passes back to the previously executing application.

Case 2: An application is running on the module memory and does not use shared libraries. If a program is executing from module memory during a removal, a bus error will occur when the CPU attempts to fetch and execute the next instruction code from the module. The bus error trap causes control to be passed to the system bus error handler. The system does not force a soft-reset because no libraries were opened. If the Setup application was installed (copied into the handheld memory when the module was inserted), it is executed with a "`remove`" launch code. The system then deletes the Setup application and control passes back to the Application Launcher.

Case 3: If an application is using shared libraries, you must take precautions to catch bus errors when the library attempts to access the module after it has been removed. If the system detects that a library is still open when the module is removed, it will force a soft-reset to ensure stability.

To handle these situations, Handspring provides the HsCardErrTry/HsCardErrCatch macros. These macro calls should be wrapped around any code in your interrupt routines, shared libraries, or system extensions that access memory or other hardware devices on the module. Alternative methods of using the Try and Catch macros are discussed in detail in the Application Note referenced below. Developers should ensure that all of the code in the Try and Catch routine is executed. You cannot return out of the Try and Catch section, as it will corrupt the stack if a module is removed.

If a module is removed when code in the `HsCardErrTry` section is executing, a bus error occurs and the system automatically passes control to the `HsCardErrCatch` section. You can then look at various local variables that you have set up to determine the best course of action to take in dealing with the module removal.

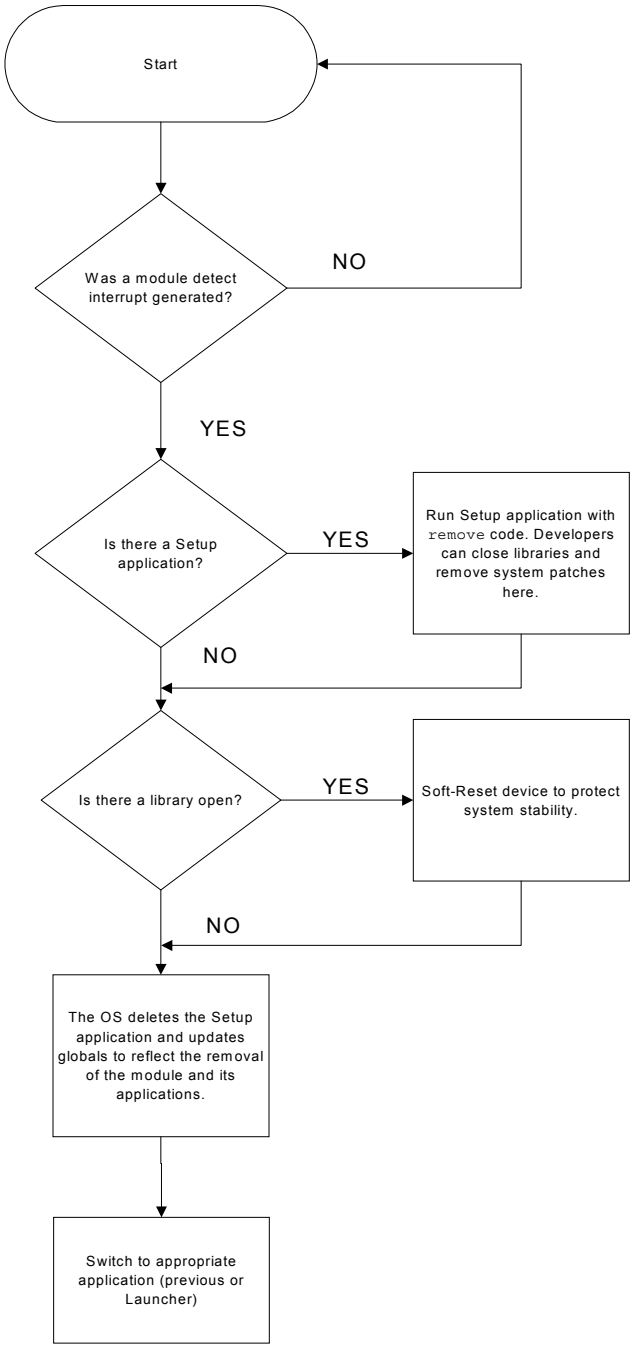
In Handspring handheld devices using PalmOS 3.5 or greater, you can simply set a flag that indicates the module has been removed and wait for your Setup application to be called with the "`remove`" message. Your Setup application should then close any open libraries and uninstall all module-dependent hooks that you have placed into the system. For systems using PalmOS 3.1, you must close the libraries manually in the `HsCardErrCatch` block.

Further details about these macros can be found in application notes (AN-02: Springboard Soft Reset Protection) on Handspring's website:

http://www.handspring.com/developers/tech_notes.jhtml

Note: Database record "busy" bits will be automatically cleared if the database is open when the system bus error handler responds to an unexpected card removal.

Figure 2-3. Operational Flowchart when Removing a Module



3. Springboard Software Integration

Modules that simply provide additional user applications in ROM are straightforward to design and construct; modules that provide additional hardware functionality often involve interrupt handlers, shared libraries, and/or system extensions to enable other applications or system software to use the new hardware. The most critical portion of software design for a module is probably the operation and interaction of these various pieces with each other and the rest of the system.

For example, a pager module might have an interrupt routine that fires off whenever a page comes in. That interrupt routine might have to store the contents of the incoming page and then decide whether or not to inform the user that a message has arrived. To inform the user of the message, it might simply display an alert on the screen (similar to a Datebook alarm going off) and/or cause a switch to a "Pager" application at which the user can view and manage the incoming pages.

This chapter provides an overview of how interrupt routines and other system extensions interact with module hardware, application code, and the rest of the system. It also references some of the API calls provided for these purposes. The most important thing to remember about interrupt handlers, shared libraries, and system extensions is that their code must be copied into main memory *before* they are installed into the system. If they are not, the system cannot recover from a module removal situation without forcing a soft reset. The handlers can, however, access memory or other devices on the module as long as they follow the guidelines described in *Catching Module Removals*. The module *Setup application* mechanism provides a convenient method for copying these types of code into the handheld internal memory before installation.

3.1. Module Setup Application

Any module that provides custom hardware and software to access the hardware usually requires a module Setup application. The Setup application installs any system extensions, interrupt handlers, shared libraries, system trap patches, etc. that are necessary for using the new hardware. It is also responsible for uninstalling them when the module is removed.

Because the Setup application is copied into the main memory before it is executed, complete installation and removal of the module software is guaranteed, even if the module is removed prior to or during installation. Note that the module has already been removed when the Setup application is called to uninstall. When a module is inserted or when the device is reset, the operating system automatically queries the module ROM to determine if there is a Setup application for the module. If a Setup application exists, the operating system copies it into main memory and then executes the Setup application by sending it an `install` message. Likewise, when the module is removed, the system calls the same Setup application, sends it a `remove` message, then deletes the Setup application itself.

A Setup application is built like any other Palm OS application, except that it must be given a special database creator 'HsCd' and type '`HsSu`'. The Handspring website contains several Software Development Kits (SDKs) that include samples of Setup applications:

http://www.handspring.com/developers/sw_dev.jhtml

A module Setup application is called with one of two action codes sent to its `PilotMain()`: `hsSysAppLaunchCmdInstall` or `hsSysAppLaunchCmdRemove`. In both cases, the `cmdPBP` parameter block passed to the application is a structure pointer containing the `card number` that has been inserted or removed. In this context, `card number` is the standard Palm OS `cardNo` parameter used by function like `DmCreateDatabase(UInt cardNo, ...)`. Usually the Setup application saves this `card number` in its globals that it allocates during the install. Refer to the Handspring extension header file (HsExt.h) for more information on `hsSysAppLaunchCmdInstall` and `hsSysAppLaunchCmdRemove`.

The parameter block passed during the `hsSysAppLaunchCmdInstall` action code also contains an `isReset` field. This value is true (non-zero) only if the install action code is being sent as a result of the device going

through a soft or hard reset. Most Setup applications can safely ignore this field because their actions are independent of whether or not the module was inserted before the reset. If the module was inserted before the reset, any shared library or other databases that the Setup application normally copies to built-in RAM are already present in built-in RAM. There is no harm in copying them from the module again.

During installation and removal processing, a Setup application is not allowed to use global or static application variables; all variables must be normal stack- or register-based local. This is the same restriction that is placed on other Palm OS applications when processing other system action codes, such as `find` or `goto`. However, most module software requires globals of some kind. These globals are most likely shared by the module's interrupt handler, applications, and other extensions. The hsCardAttrCardParam attribute of the module is provided for this purpose. Use this attribute to store a 32-bit pointer to the module's globals; it can be set using HsCardAttrSet() and retrieved using HsCardAttrGet(). In addition, this attribute is automatically passed to the module's interrupt handler as a parameter on the stack.

If the module Setup application needs to install a shared library that is in a separate database on the module, it must first copy the shared library database from the removable module into main memory before installing the library. Likewise, it should delete the library from main memory during removal processing. The library database can be easily copied into memory using the HsDatabaseCopy() routine.

If a module Setup application needs to patch any system traps, it must use the HsCardPatchInstall() routine to install them. It should *not* use SysSetTrapAddress(). Using HsCardPatchInstall() ensures that the patch can be safely removed using HsCardPatchRemove() during removal processing without interfering with other third-party extensions that may have been activated or de-activated (using an application like HackMaster) in the meantime.

If a module Setup application installs an interrupt handler, event handler, or system patch from its own code segment (a subroutine linked in with the Setup application itself), it must be sure to lock down its code segment before returning from the install action code. This precaution ensures that its code segment is not inadvertently moved by the Palm OS memory manager while the module is installed. The following portion of code performs this operation:

```
// Because we installed a patch from this code resource, make sure
// this code resource remains locked down after we exit.
VoidHand            codeResH;
codeResH = DmGet1Resource ('code', 1);
if (codeResH) MemHandleLock (codeResH);
```

Similarly, during the remove action code the module Setup application should restore the lock count of the code resource as follows:

```
// Restore lock count of code resource
VoidHand            codeResH;
codeResH = DmGet1Resource ('code', 1);
if (codeResH) MemHandleUnock (codeResH);
```

A typical module Setup application allocates a memory chunk for the module's globals using `MemPtrNew()`, resets the owner of this chunk to 0 using `MemPtrSetOwner()`, then stores the returned pointer in the hsCardAttrCardParam attribute of the module. A code example follows:

```
DWord
PilotMain (Word cmd, Ptr cmdPBP, Word launchFlags)
{
  void*                    globalsP;
  Err   err;
  VoidHand          codeResH;

  if (cmd == hsSysAppLaunchCmdInstall)
    {
      HsSysAppLaunchCmdInstallType* installP;
```

```
      installP = (HsSysAppLaunchCmdInstallType*)cmdPBP
      globalsP = MemPtrNew (sizeof (MyGlobalsType));
      if (globalsP)
        {
          MemPtrSetOwner (globalsP, 0);
          globalsP->cardNo = installP->cardNo;
          HsCardAttrSet (globalsP->cardNo, hsCardAttrCardParam,
                         &globalsP);
        }

      // Install shared libraries using SysLibInstall()
      // ...
      // Patch system traps using HsCardPatchInstall()
      // ...
      // Install interrupt handler using
      //  HsCardAttrSet(globalsP->cardNo, hsCardAttrIntHandler, ...)
      // ...
      // Enable module interrupts using
      //  HsCardAttrSet (globalsP->cardNo, hsCardAttrIntEnable, ...)
      // ...

      // Because we installed a patch from this code resource, make sure
      // this code resource remains locked down after we exit.
      codeResH = DmGet1Resource ('code', 1);
      if (codeResH) MemHandleLock (codeResH);
  }

  else if (cmd == hsSysAppLaunchCmdRemove)
    {
      HsSysAppLaunchCmdRemoveType* removeP;

      removeP = (HsSysAppLaunchCmdRemoveType*)cmdPBP
      err = HsCardAttrGet (removeP->cardNo, hsCardAttrCardParam,
                           &globalsP);
      if (!err && globalsP)
        MemPtrFree (globalsP);
      // Remove shared libraries using SysLibRemove()
      // ...
      // Restore system traps using HsCardPatchRemove()
      // ...
      // Restore lock count of our code resource
      codeResH = DmGet1Resource ('code', 1);
      if (codeResH) MemHandleUnlock (codeResH);

    }

  return 0;
}
```

Another restriction placed on a module Setup application occurs when it is called with the "`remove`" message. When this happens, it cannot display any alerts (using `FrmAlert()` or similar mechanism) or make system calls that might display user interface (UI) elements or process user interface events. The "`remove`" action code can be sent to the Setup application after the current application has aborted but before a new application is launched; therefore the system might not be in a state to process user interface events. The install action code, however, does not have these restrictions. It can display alerts or progress dialogs if it wishes to because it is always called from the context of the current application's main event loop.

By the time the Setup application is called with the "`remove`" message, it can assume that the system has already done the following:

1. Disabled module interrupts by resetting the hsCardAttrIntEnable attribute to 0.
2. Removed the module interrupt handler by resetting the hsCardAttrIntHandler attribute to 0.

3. Removed the module event handler by resetting the hsCardAttrEvtHandler attribute to 0.

4. Removed the module's power handler by resetting the hsCardAttrPwrHandler attribute to 0.

Note: The operating system will clear the hsCardAttrIntHandler automatically when the module is removed; however, it is better practice for the Setup application to do this itself for instances in which the module is inserted, but not used.

One condition to consider is when the user replaces the main batteries. In this case, the module Setup application may not be executed until the device is powered on. This may have design implications for developers who want to put the module into a low power state through the Setup application.

Note: The Setup application can be bypassed by holding down the Up button during module insertion.

3.2. Overriding Module Software

The basic principle behind Handspring's removable modules is that all of the module software and hardware resides on the module itself. In this way, a module inserted in any handheld is immediately functional without requiring any manual software installation or configuration. Also, most modules will be built with masked ROM in order to keep costs to a minimum. In the unfortunate event that a bug is discovered after a masked ROM module has been released, however, you may need to provide a software patch for users of your module.

By taking special precautions in the design of your Setup application, you can minimize the need to design and implement a software patch for your module. For example, suppose your module copies a shared library from the module ROM to internal memory. Instead of blindly copying the shared library from the module to internal memory, your Setup application should first check internal memory for a newer version of that library. If a newer version already exists in internal memory, you can skip copying the version from the module. Use the `DmGetNextDatabaseByTypeCreator()` call in these situations; it automatically searches for the latest version of a database by type and/or creator. If you find the newer version in internal memory (`card number 0`), then you can skip copying the database from the module.

Note: Always register your database creator IDs with 3Com Developer Support to ensure that they are unique.

3.3. Module Welcome Application

Whenever a module is inserted, and after copying and executing the *Setup application* (if present), the system looks for a Welcome application on the module. This application is a normal Palm OS application with "`CardWelcome`" as the database name. If this application is found, the OS automatically switches to it. For example, this application could be a module-specific application launcher or an application that lets you set preferences for the module. Because it is a normal Palm OS application, it appears in the Palm OS applications launcher and can be accessed through the launcher anytime after the module has been inserted. The name that appears in the Palm OS application launcher for this Welcome application can be set up in the "`tAIN`" resource of the welcome application --the same as for other Palm OS applications. This launcher's visible name in the "`tAIN`" resource is independent of the actual database name of "`CardWelcome`".

Note: The Welcome application can be bypassed by holding down the Up button during module insertion.

If a module is inserted during a soft or hard reset, the system does not automatically switch to the Welcome application. You can, however, override this behavior and have the Welcome application launch automatically during a reset. To do this, include the "`HsWR`" token in the module's ROM. To include this token, specify it on the command line to the Palm-MakeROM tool (for example, `-tokStr "HsWR" "1",` launches the Welcome application after a reset) as described in the *Handspring Development Tools Guide*.

Note: In Handspring handhelds running Palm OS 3.5 or later, the Welcome application can also be specified by type and creator if the "HsWt" token is present (for example, `-tokStr "HsWt" "applHsIm"` will launch an application with type `appl` and creator `HsIm`). Developers should use the method described above which will

work in both Palm OS 3.1 and 3.5 devices to provide compatibility for the widest range of systems. Finally, the HsWt option cannot be disabled through compiler options.

3.4. Interrupt Handler Interaction

All interrupt handlers in the Palm OS must observe a number of restrictions:

- They cannot allocate, free, or move memory in any memory heap
- They cannot create, modify, or delete databases
- They cannot create, delete, or block system resources such timers, tasks, or semaphores
- They cannot display any user interface.

About the only thing an interrupt handler can safely do is change the contents of pre-allocated, locked memory blocks, queue keyboard events, or trigger system semaphores. In addition, interrupt handlers must execute as quickly as possible in order to maximize user-responsiveness of the device and minimize the chances of adversely interfering with other interrupt response times.

One of the first tasks of any module interrupt handler is to remove the source of the interrupt. This step usually involves reading a register on the module that effectively deasserts the interrupt line. Module interrupts are level-sensitive and unless the module interrupt handler removes the source of the interrupt by the time it exits, the interrupt handler is immediately re-executed.

Next, the module interrupt handler usually processes and/or stores some data regarding the interrupt. For this purpose, it usually needs a pointer to some global data containing one or more buffers and/or counters. This pointer must have been allocated in advance by application or setup code. Usually, this type of buffer is allocated during initialization time using `MemPtrNew()` and is set to have an owner ID of 0 (using `MemPtrSetOwner`) to prevent the system from freeing it when the current application quits. Module interrupt handlers are passed a 32-bit parameter on the stack, which is set up through the hsCardAttrCardParam attribute of the HsCardAttrSet() call. Usually this parameter is the globals pointer for the interrupt handler.

Finally, in may be necessary for the interrupt handler to notify the system or an application that an event of particular interest has occurred. In particular, if an alert or similar message needs to be displayed, the interrupt handler must rely on application or system code to display it *after* the interrupt handler returns, because interrupt handlers themselves are not allowed to have any user interface. Typically the interrupt handler uses the HsAppEventPost() system routine to trigger the user interface.

Note: Refer to *Power Management Options for Interrupt Handlers* for important information about when it is safe to make system calls from a module interrupt handler. System calls cannot be made from a module interrupt handler if the `*sysAwakeP` parameter passed to it is false.

The interrupt handler can use the HsCardEventPost() call to post a module event. This call accepts an event number between 0 and `hsMaxCardEvent` and a 16-bit event parameter. After the interrupt handler exits and control is returned to the main event loop, the system calls the module's `CardEvtHandler()` with the given event number and 16-bit event parameter. `CardEvtHandler()` is installed through the hsCardAttrEvtHandler attribute of HsCardAttrSet(). For convenience, the `CardEvtHandler()` is also passed a copy of the same 32-bit parameter that the interrupt handler gets (the interrupt handler globals).

Because `CardEvtHandler()` executes from the context of the main event loop, it has no restrictions as far as allocating memory or making other system calls. It can put up an alert, call `SysUIAppSwitch()` to cause a switch to another application, or do anything else it desires. It is important to note that the `CardEvtHandler()` function executes in the context of -- or is effectively called from -- the current UI application. For this reason, the module event handler function should keep its stack usage (e.g., local variables or nested function calls) to a minimum.

4. Software API Extension

This chapter describes in detail the calling conventions and parameters for each of the Handspring API calls.

For a description of how and where to use most of these calls, refer to the section entitled *Module design details*, and the chapter entitled *Springboard Software Integration.* The use of the remaining generic utility calls provided by Handspring is described in *Utility Calls*.

The header file "`HsExt.h`" provided with the development tools contains all the public equates referenced in this chapter, including all constants, structure definitions, function prototypes, etc.

4.1. Checking Presence and Version of Handspring Extensions

For application code or other types of code that might be installed onto both Handspring devices and non-Handspring Palm OS devices, you must first ensure that you are on a Handspring device before making Handspring-specific API calls. Use the `FtrGet()` call of Palm OS and check for the presence of the Handspring extensions feature. The `hsFtrCreator` and `hsFtrIDVersion` constants that are passed to `FtrGet()` are defined in the Handspring header file `HsExt.h`. For example:

```
DWord               value;
err = FtrGet (hsFtrCreator, hsFtrIDVersion, &value);
if (!err)
  {
    // Since FtrGet() did not return an error, we can
    // safely make Handspring specific API calls like
    // HsCardAttrGet(), HsCardAttrSet(), etc.
  }
```

If `FtrGet()` returns no error, then the Handspring extensions are present and it is safe to call any Handspring API call that is described in this chapter. The current version level of the Handspring extensions is returned in the `value` parameter. It is encoded as 0xMMmfsbbb, where MM is the major version number, m is the minor version number, f is the bug fix level, s is the stage (3-release, 2-beta 1-alpha, 0-development) and bbb is the build number for non-releases. For example, Version 2.00a2 would be encoded as 0x02001002, where the major version number is 02, the minor version number is 0, the bug fix level is 0, the stage is alpha, and the build number is 0x2.

Because the format used for the version number of the Handspring extensions is the same as that used for the Palm OS ROM version number, your code can use version macros such as `sysGetROMVerMajor` and `sysGetROMVerMinor` (defined in `SystemMgr.h` of the Palm Computing Platform SDK) for decoding the version number of the Handspring extensions.

Another Handspring feature provides the modification date of the Handspring extensions. This feature has an ID of `hsFtrIDModDate` (replaces `hsFtrIDVersion` in the above example). The value of this feature changes with every release of the Handspring extensions, regardless of whether or not new features were added. Thus you should use it for informative purposes only. If your software makes decisions based on a certain version or feature of the Handspring extensions in order to run, it should use the `hsFtrIDVersion` feature instead.

4.2. Utility Calls

Besides the API calls that are provided specifically for dealing with removable modules, the Handspring utility functions described in this section are not necessarily module-related.

The HsDatabaseCopy() call can be used to copy a database from a module to built-in memory, from built-in memory to a module, or to duplicate an existing database within the same module. This function can be useful in module Setup applications if they need to copy any databases from the module to built-in memory as part of the setup process.

The HsAppEventHandlerSet() and HsAppEventPost() calls can be used to provide more flexible event-handling mechanisms for an application. Normally, when applications have their own custom event types, they must specifically look for these events in their main event loop after calling `EvtGetEvent()`. This process is not a problem in the main event loop of the application, but it can be a problem if a system event loop (such as the system "Find" dialog or "Category Edit" dialog event loops) is executing at the time the event is posted. Because the system event loop is not aware of any application-specific event types, it simply ignores them.

If the application registers its own custom event handler procedure using HsAppEventHandlerSet(), then this procedure is called automatically by the system during `SysHandleEvent()` in response to any event posted by HsAppEventPost(). This way, the application's event handler is called even if the event is posted, while the system is in one of its own custom dialog event loops.

Keep in mind that your application's event handler might be called from the context of another application's action code processing. Even though your application is the "current" application visible to the user at the time, the global CPU registers are not your application's globals due to the action code processing. Thus your application event handler routine must never rely on global variables. Instead, it should use the `evtRefCon` parameter that is passed to it as a pointer to any variables it needs to access or update. This `evtRefCon` parameter value is set up by the HsAppEventHandlerSet() call.

4.3. Generic Module Support in Palm OS

Besides the calls mentioned in this chapter, the standard Palm OS memory and data manager calls for dealing with modules and memory on modules are always available. For example, `MemNumCards()` returns the number of modules that are present. It returns "2" if there is a removable module inserted and "1" otherwise. The built-in memory of the handheld base unit is accessed by passing a module number of "0" to the appropriate memory and data manager calls, whereas the removable module memory is accessed using a module number of "1." Calls such as `MemCardInfo()` can be used to get the module name, manufacturer name, etc., for any module by passing the appropriate module number.

Remember that even if `MemNumCards()` returns a "2", (indicating that the removable module is currently installed), the user may pull the module out at any time—even immediately after this call returns. When a module is removed, the system automatically aborts any application code that is currently in the process of accessing memory on the removable module and returns control to the application launcher. In general, application code can simply call `MemNumCards()` during its start-up and assume that the information will remain current until the application exits. If the module is pulled out while the application is using it, the system automatically and immediately aborts the application. If the application is *not* referencing the module when the module is pulled out, then the system sends it an exit event and waits for the application to exit normally. When a module is inserted, the current application is sent a normal exit event as well. It may or may not be re-launched after it exits, depending upon the contents of the module (for example, it might have a Welcome application on it that is launched instead).

4.4. Copy Protecting Module Applications

If desired, module applications can easily be designed so that they will not run if copied off the module to another device's internal RAM. A simple mechanism is for the application to check for the presence of the removable module and compare the module's name when it starts up. If the module is not present or the module's name is not correct, it can display an appropriate error message and refuse to run.

All removable modules must have unique module and manufacturer names that must be registered with Handspring. These names are null-terminated ASCII strings of up to 31 characters. To get the name of a module, use the `MemCardInfo()` call and pass in the module number of the removable module.

For example:

```
char          cardName[32];
Err    err;

if MemNumCards() > 1
  {
  err = MemCardInfo (1 /*cardNo*/, cardName, 0 /*manufName*/,
          0 /*versionP*/, 0 /*crDateP*/, 0 /*romSizeP*/,
          0 /*ramSizeP*/, 0 /*freeBytesP*/);
  if (err || StrCompare (cardName, "MyCardName"))
    {
      DisplayCopyProtectError();
    }
  }
```

4.5. API Calls

This section lists the Handspring API calls in alphabetical order.

Table 4-1. API Call Summary

Call	Description
HsAppEventHandlerSet	Register event handler procedure
HsAppEventPost	Enable event posting by application event handler
HsCardAttrGet	Retrieve attribute
HsCardAttrSet	Set attribute
HsCardErrTry/HsCardErrCatch	Enable safe recovery for module removals
HsCardEventPost	Queue an event
HsCardPatchInstall	Enable patch installs
HsCardPatchPrevProc	Get address of previous system trap implementation
HsCardPatchRemove	Remove Setup application patch
HsDatabaseCopy	Copy Palm OS database
HsEvtResetAutoOffTimer	Reset auto-off timer
HsPrefGet	Get Handspring preferences
HsPrefSet	Set Handspring preferences

4.5.1. HsAppEventHandlerSet

Summary

Provided for applications so they can register their own event handler procedure that can be triggered using HsAppEventPost().

Prototype

```
Err
HsAppEventHandlerSet (HsAppEventHandlerPtr procP, DWord evtRefCon)
```

Description

Sets up an application's event handler procedure. The system calls this handler procedure from the main event loop in response to an event posted by the HsAppEventPost() routine.

The event handler has this prototype:

```
Boolean AppEvtHandler (DWord evtRefCon, Word evtNum, Word evtParam)
```

It returns "true" if the event was successfully handled and "false" if not. The `evtRefCon` parameter is a copy of the `evtRefCon` value passed to HsAppEventHandlerSet(). The `evtNum` and `evtParam` parameters are copies of values passed in to HsAppEventPost().

Note that this event handler can be called while the system is in the middle of sending an action code, such as `find`, to another application. Even though your application is the "current" application visible to the user at the time, the global CPU registers may not be your application's globals due to the action code processing. Thus the AppEvtHandler routine must never use global variables. Instead it should use the `evtRefCon` parameter as a pointer to a structure containing any variables it needs to reference or update.

The system automatically removes your `AppEvtHandler`() for you when your application quits.

Parameters

procP	IN	Pointer to event handler procedure, or NIL to remove the current one.
evtRefCon	IN	This 32-bit reference constant gets passed to the event handler when it is called by the system.

Returns

0	If no error

4.5.2. HsAppEventPost

Summary

Provided for applications so that they can post an event to be processed by their own application event handler procedure installed using the HsAppEventHandlerSet() call.

Prototype

```
Err
HsAppEventPost (Word evtNum, Word evtParam)
```

Description

Queues an event for the application's own event handler procedure. The system calls the event handler procedure from the main event loop. The event handler can be installed using the HsAppEventHandlerSet() call.

The event handler has this prototype:

```
Boolean AppEvtHandler (DWord evtRefCon, Word evtNum, Word evtParam)
```

It returns "true" if the event was successfully handled and "false" if not. The `evtRefCon` parameter is a copy of the `evtRefCon` value passed to HsAppEventHandlerSet().

Parameters

evtNum	IN	The event number to post. Can be any value between 0 and hsMaxAppEvent.
evtParam	IN	A 16-bit parameter that is passed to CardEvtHandler.

Returns

0	If no error

4.5.3. HsCardAttrGet

Summary

Retrieves any one of the attributes of a module or its software/hardware interface.

Prototype

```
Err
HsCardAttrGet (Word cardNo, HsCardAttrEnum attr, void* valueP)
```

Description

Returns the current value of a particular module attribute designated by the `attr` parameter. The return value is placed in the buffer pointed to by `valueP`.

Parameters

attr	IN	Which attribute to retrieve
cardNo	IN	Which module number to query about
*valueP	OUT	Value of attribute is returned in this buffer

The possible values of `attr` and the corresponding return types are shown below. The R/W column indicates if settings are read-only (R) or read-write (RW). The read-write attributes can be configured through the HsCardAttrSet() call.

Setting	Value	R/W	Description
hsCardAttrRemovable	Byte	R	True if this module is a removable module. False if the module is not removable (built-in module: cardNo = 0). Returns hsErrInvalidCard if slot "cardNo" does not exist.
hsCardAttrHwInstalled	Byte	R	True if a module is physically installed at "cardNo" and has finished its power-on reset cycle. False if not. Returns hsErrInvalidCard if slot "cardNo" does not exist. Note that this attribute is true before hsCardAttrSwInstalled is true.
hsCardAttrSwInstalled	Byte	R	True if the Palm OS memory, data, and other managers have been updated to access the given module. False otherwise. Note that this attribute is not true until some period of time after the module is physically installed and hsCardAttrHwInstalled is true.
hsCardAttrCsBase	DWord	R	Base address of first slot chip select. The second chip select always starts at hsCardAttrCsBase + hsCardAttrCsSize.
hsCardAttrCsSize	DWord	RW	Address range of each of the chip selects.

Setting	Value	R/W	Description
hsCardAttrAccessTime	DWord	RW	Minimum access time of slot chip selects in nanoseconds. Note that when set, the value passed in is rounded up to the next access time setting available in hardware.
hsCardAttrReset	Byte	RW	If value is non-zero, the module reset signal going to the module is asserted. If zero, the module reset signal is deasserted.
hsCardAttrIntEnable	Byte	RW	If value is non-zero, module interrupts are enabled. If value is zero, the module interrupts are disabled.
hsCardAttrCardParam	DWord	RW	Contains the 32-bit parameter that is passed to the module's interrupt handler, power handler, and event handler.
hsCardAttrIntHandler	void*	RW	This attribute is a pointer to the module interrupt handler which must have this calling convention: void CardIntHandler (DWord cardParam, Boolean* sysAwakeP); The cardParam that is passed to this interrupt handler can be set up through the hsCardAttrCardParam attribute.
hsCardAttrPwrHandler	void*	RW	This attribute is a pointer to the module's power handler routine, which must have this calling convention: void CardPwrHandler (DWord cardParam, Boolean sleep, HsCardPwrDownEnum reason) The cardParam is a convenience copy of the hsCardAttrCardParam module attribute.
hsCardAttrEvtHandler	void*	RW	This attribute is a pointer to a Module Event Handler procedure. A module interrupt handler can trigger the system to call this event handler using HsCardEventPost(). This mechanism is described in *Interrupt Handler Interaction*
hsCardAttrLogicalBase	void*	R	Contains the logical base address reserved for the given module slot. This value, added to the module's header offset, gives the address of the module's ROM header. Most applications do not need to use this value. It is provided mainly for advanced applications that need to format their own module headers (e.g. flash programming applications).

Setting	Value	R/W	Description
hsCardAttrLogicalSize	DWord	R	Contains the logical address space reserved for the given module. This value is greater than or equal to the actual addressable memory on the module.
hsCardAttrHdrOffset	DWord	R	Contains the offset from the module base address to the module's ROM header. Most applications do not need to use this value. It is provided mainly for advanced applications that need to format their own module headers (e.g. flash programming applications).

Returns

0	If no error
hsErrNotSupported	This setting is not supported
hsErrInvalidCard	Invalid module number

4.5.4. HsCardAttrSet

Summary

Sets an attribute of a module's interface.

Prototype

```
Err
HsCardAttrSet (Word cardNo, HsCardAttrEnum attr, void* valueP)
```

Description

Sets the new value of a particular module attribute designated by the `attr` parameter.

Parameters

attr	IN	Which attribute to set
cardNo	IN	Which module number to set attribute on
*valueP	IN	New value of attribute is passed in this buffer

The possible values of `setting` and the corresponding data type of `*valueP` are documented in the HsCardAttrGet() call. Only those indicated as being read/write (RW) attributes can be changed through this call.

Returns

0	If no error
hsErrNotSupported	This setting is not supported
hsErrInvalidCard	Invalid module number
hsErrReadOnly	This attribute is read-only and cannot be changed.

4.5.5. HsCardErrTry/HsCardErrCatch

Summary

These macros are provided for interrupt handlers, system extensions, and shared libraries that need to access memory or devices on a removable module. These calls enable safe recovery if the module is removed while in critical sections of code.

Prototype

```
HsCardErrTry
{
       // Do something that accesses the removable module
}

HsCardErrCatch
{
       // Recover or cleanup after a failure in the above Try block.
       // The code in this Catch block does not execute if
       // the above Try block completes without a module removal
} HsCardErrEnd

// You must structure your code exactly as above. You cannot have a
// HsCardErrTry { } without a HsCardErrCatch { } HsCardErrEnd,
// or vice versa.
```

Description

The HsCardErrTry/HsCardErrCatch macros should be wrapped around any section of code within an interrupt handler, system extension, shared library, or other system code that needs to access memory or hardware on a removable module. If the module is removed while the critical section of code is executing, control is passed to the HsCardErrCatch() section.

These macros can be nested. For example, you can call a subroutine from within your HsCardErrTry block that has its own try/catch block. Every routine that has an HsCardErrTry clause, however, must have an HsCardErrCatch.

Note that these macros require some amount of time to execute that should be considered when using them within an interrupt service routine. Using these macros will essentially extend the interrupt latency time.

Limitations

HsCardErrTry and HsCardErrCatch are based on setjmp/longjmp. At the beginning of a Try block, setjmp saves the machine registers. A module removal triggers longjmp, which restores the registers and jumps to the beginning of the Catch block. Therefore, any changes in the Try block to variables stored in registers are not retained when entering the Catch block.

The solution is to declare variables that you want to use in both the Try and Catch blocks as “volatile.”

For example:

```
        volatile long    x = 1;     // Declare volatile local variable
        HsErrErrTry
        {
        ...
        x = 100;                // Set local variable in Try
        ...
        }

HsCardErrCatch
{
        if (x > 1)              // Use local variable in Catch
        SysBeep(1);
} HsCardErrEnd
```

Parameters

none

Returns

N/A

4.5.6. HsCardEventPost

Summary

Provided for interrupt handlers so that they can queue an event for processing later by a CardEvtHandler() procedure.

Prototype

```
Err
HsCardEventPost (Word cardNo, Word evtNum, Word evtParam)
```

Description

Queues an event for a CardEvtHandler() procedure. The system calls the CardEvtHandler procedure from the main event loop of the current application after the interrupt handler returns. The CardEvtHandler can be installed using the hsCardAttrEvtHandler attribute of HsCardAttrSet().

The CardEvtHandler has this prototype:

```
Boolean CardEvtHandler (DWord cardParam, Word evtNum, Word evtParam)
```

It returns "true" if it successfully handled the event and "false" if it did not. The `cardParam` value passed to CardEvtHandler is a convenience copy of the hsCardAttrCardParam module attribute.

Parameters

evtNum	IN	The event number to post. Can be any value between 0 and hsMaxCardEvent.
cardNo	IN	Module number for which to post event.
evtParam	IN	A 16-bit parameter that is passed to CardEvtHandler.

Returns

0	If no error
hsErrInvalidCard	Invalid module number

4.5.7. HsCardPatchInstall

Summary

Enables Module Setup utilities to install patches to Palm OS system calls.

Prototype

```
Err
HsCardPatchInstall (Word trapNum, void* procP)
```

Description

Patches a Palm OS system trap call. Setup utilities should always use this call to patch traps rather than the Palm OS SysSetTrapAddress() call because this call ensures compatibility with other third-party extensions that may have been installed by the user (through HackMaster or equivalent).

The implementation of the patch must use the HsCardPatchPrevProc() routine to obtain the address of the "old" trap call in order to pass control over to it.

When the module Setup application is called to remove the module software, it must use HsCardPatchRemove() to remove every patch installed by HsCardPatchInstall().

Important: Module Setup applications are only allowed to install *one* patch per system trap number. If HsCardPatchInstall() is called twice for the same trap without an intervening HsCardPatchRemove(), it returns the error code `hsErrCardPatchAlreadyInstalled`.

Here is an example of a patch implementation that does some work then passes control over to the previous implementation:

```
static Boolean
PrvCardSysHandleEvent (EventPtr eventP)
{
	Boolean	handled;
	Boolean	(*oldProcP) (EventPtr eventP) = 0;

	// Do some stuff
	//...

	// Call old routine
	HsCardPatchPrevProc (sysTrapSysHandleEvent,
		(DWord*)&oldProcP);
	handled = (*oldProcP) (eventP);

	return handled;
}
```

Parameters

trapNum	IN	The trap number of the call to patch. This value is a SysTrapNumber Palm OS enum value as found in the Palm OS header file <SysTraps.h>.
procP	IN	Pointer to procedure to plug into the system trap.

Returns

0	If no error
hsErrInvalidCard	Invalid module number

4.5.8. HsCardPatchPrevProc

Summary

Used by system patches installed using HsCardPatchInstall() to get the address of the previous implementation of the system trap.

Prototype

```
Err
HsCardPatchPrevProc (Word trapNum, void** prevProcPP)
```

Description

Use this call inside the implementation of a system patch for a module in order to get the address of the previous implementation of the call. In most cases, patches do their own work before calling the previous implementation. See HsCardPatchInstall() for an example of a patch implementation.

Parameters

trapNum	IN	The trap number of the call that was patched. This value is a SysTrapNumber Palm OS enum value, as found in the Palm OS header file <SysTraps.h>
*prevProcPP	OUT	The address of the previous implementation is returned in this pointer.

Returns

0	If no error
ftrErrNoSuchFeature	Trap was not patched by HsCardPatchInstall

4.5.9. HsCardPatchRemove

Summary

Removes a module Setup application patch installed by HsCardPatchInstall().

Prototype

```
Err
HsCardPatchInstall (Word trapNum, void* procP)
```

Description

When the module Setup application gets called to remove the module software, it must use this call to remove every patch installed by HsCardPatchInstall().

Parameters

trapNum	IN	The trap number of the call that was patched. This value is a SysTrapNumber Palm OS enum value as found in the Palm OS header file <SysTraps.h>

Returns

0	If no error
hsErrCardPatchNotInstalled	Trap was not patched by HsCardPatchInstall

4.5.10. HsDatabaseCopy

Summary

Copies an entire Palm OS database.

Prototype

```
Err
HsDatabaseCopy (Word srcCardNo, LocalID srcDbID, Word dstCardNo,
char* dstNameP, DWord hsDbCopyFlags, char* tmpNameP,
LocalID* dstDbIDP)
```

Description

Copies an entire Palm OS database. The source and destination can be the same module or different modules, and the source database can be copied from ROM or RAM.

The `hsDbCopyFlags` parameter can be used to control the copy operation. The caller has the option of preserving the creation date, modification date, backup date, and/or modification number of the source database, as well as whether or not an existing database with the same name should be automatically overwritten or not.

Note that this function does not inherently support writing to Flash memory. The developer should check whether the destination is read-only memory.

Parameters

srcCardNo	IN	Module number of source database.
srcDbID	IN	Database ID of source database.
dstCardNo	IN	Module number of destination database.
dstNameP	IN	Name of new database. If a nil pointer is passed, then the name of the source database is used.
hsDbCopyFlags	IN	One or more of hsDbCopyFlagXXX flags as described below in Table II.2.
tmpNameP	IN	Temporary name to use while copying, or nil pointer to use default temporary name.
*dstDbIDP	OUT	The database ID of the created destination database is returned here unless dstDbIDP is a nil pointer.

Table 4-2 lists the possible flags for the hsDbCopyFlags parameter.

Table 4-2. hsDbCopyFlags Flags

Flag	Description
hsDbCopyFlagPreserveCrDate	Preserve the creation date of the source database. If not set, then the destination database gets the current date and time as its creation date.
hsDbCopyFlagPreserveModDate	Preserve the modification date of the source database. If not set, then the destination database gets the current date and time as its modification date.
hsDbCopyFlagPreserveModNum	Preserve the modification number of the source database. If not set, then the destination database gets a new modification number unrelated to the source database.
hsDbCopyFlagPreserveBckUpDate	Preserve the backup date of the source database. If not set, then the destination database gets a backup date of 0.
hsDbCopyFlagOKToOverwrite	OK to overwrite an existing database with the same name. Preserve the creation date of the source database. Any pre-existing destination database with the same name is left intact until the source has been copied over as a temporary database. This guarantees that any pre-existing database is not lost if the copy operation fails.
hsDbCopyFlagDeleteFirst	Delete existing destination database first, if it exists. Normally any pre-existing destination database with the same name is left intact until the source has been copied over as a temporary database. This guarantees that any pre-existing database is not lost if the copy operation fails. If space is limited on the destination module, however, there may not be room for two temporary copies of the destination database, so this flag can be set to override the default behavior.

Returns

0	If no error
non-zero	If a data, memory, or other type of error occurs during the copy operation

4.5.11. HsEvtResetAutoOffTimer

Summary

Provided for interrupt handlers so that they can reset the auto-off timer of the system and turn on the LCD, if it is not already on.

Prototype

```
Err
HsEvtResetAutoOffTimer (SDWord stayAwakeTicks, Boolean userOn)
```

Description

By default, if a module interrupt wakes the device and the handler returns without calling HsEvtResetAutoOffTimer(), the system puts the device back to sleep on the next round through the event loop (see *Power Management Options for Interrupt Handlers* for a complete description).

However, by calling HsEvtResetAutoOffTimer(), the interrupt handler can tell the system to turn on the LCD if it is not already turned on and tell the system to stay awake for at least a certain number of system ticks. If this call is made when the LCD is already on, it has no effect other than to possibly extend the auto-off timer.

The `stayAwakeTicks` parameter is specified in system ticks, but the current granularity of the system in this respect is only approximately five seconds. Consequently, if you pass the value like `sysTicksPerSecond*1`, the system might not shut off for five seconds. Passing (-1) for `stayAwakeTicks` makes the system stay awake for at least the current auto-off time as specified in the General Preferences panel. If `stayAwakeTicks` is zero, the system will go back to sleep during the next immediate event loop.

Parameters

stayAwakeTicks	IN	Passing –1 instructs the system to stay awake for the current Auto-off time as specified in the General Preference panel. Passing 0 instructs the system to go to sleep in next loop. Passing any other value will set the minimum amount of time in system ticks to stay awake
userOn	IN	If true, turn on the LCD if it is not already on.

Returns

0	If no error

4.5.12. HsPrefGet

Summary

This routine is used to retrieve the value of various Handspring preferences.

Prototype

```
Err
HsPrefGet (Word pref, void* bufP, DWord* prefSizeP);
```

Description

Currently, the only preferences that can be retrieved by this routine are those that specify which serial library to use for certain applications, such as HotSync or Debugger Console. By default, most serial applications use the built-in serial port on the cradle, but through these Handspring preferences, these applications can be re-directed to use a different serial library (e.g., one on a Springboard module).

The various applications that can be re-directed to a different serial library include:

- Local HotSync
- Modem HotSync
- Debugger Console
- IrDA
- All Others (i.e., any and all other applications that would normally use the built-in serial port)

When called to retrieve one of the serial library preferences, this call returns the name of the actual serial library to be used. If the particular application class has not been re-directed to a different library, then the name of the built-in library is returned ("BuiltIn SerLib").

The *pref* parameter is an enumerated constant (of type *HsPrefEnum*) that specifies the preference to retrieve. The *bufP* parameter is a pointer to a buffer to hold the preference value and **prefSizeP* must be initialized to the size of the bufP buffer. When called to retrieve the name of a serial library preference, bufP should be at least 32 bytes long. On exit, **prefSizeP* will contain the actual length of the name (including null byte).

Parameters

pref	IN	Enumerated constant of type HsPrefEnum that specifies which preference to retrieve
bufP	IN	Pointer to buffer to hold preference value. When retrieving one of the hsPrefSerialLib type preferences, this buffer should be at least 32 bytes long.
prefSizeP	INOUT	On entry, size of the bufP buffer. On exit, actual size of the retrieved preference.

The possible values of `pref` are:

Setting	Description
hsPrefSerialLibHotSyncLocal	Returns the name of the serial library to use for local HotSync
hsPrefSerialLibHotSyncModem	Returns the name of the serial library to use for modem HotSync
hsPrefSerialLibConsole	Returns the name of the serial library to use for the debugger console
hsPrefSerialLibIrda*	Returns the name of the serial library to use for IRDA.
hsPrefSerialLibDef	Return the name of the serial library to use for all other applications (i.e. any application that opens up the built-in serial port library named "Serial Library").

* This feature is not supported on Visor or Visor Deluxe.

Returns

0	If no error
hsErrInvalidParam	Invalid pref parameter
hsErrBufferTooSmall	*prefSizeP is too small to return the designated preference. On exit, *prefSizeP will be updated to the required size.

4.5.13. HsPrefSet

Summary

This routine is used to set the values of various Handspring preferences.

Prototype

```
Err
HsPrefSet (Word pref, void* bufP, DWord* prefSizeP);
```

Description

Currently, the only preferences that can be set by this routine are those that specify which serial library to use for certain applications, such as HotSync or Debugger Console. By default, most serial applications use the built-in serial port on the cradle, but through these Handspring preferences, these applications can be re-directed to use a different serial library (e.g., one on a Springboard module).

The various applications that can be re-directed to a different serial library include:

- Local HotSync
- Modem HotSync
- Debugger Console
- IrDA
- All Others (i.e., any and all other applications that would normally use the built-in serial port)

When called to set one of the serial library preferences, this call sets the name of the actual serial library to be used. To change a serial library preference back to the built-in port, pass in a pointer to an empty string (i.e., bufP = “”).

The *pref* parameter is an enumerated constant (of type *HsPrefEnum*) that specifies the preference to retrieve. The *bufP* parameter is a pointer to the new value and **prefSizeP* is the size of the new value (including null terminator).

Parameters

pref	IN	Enumerated constant of type HsPrefEnum that specifies which preference to retrieve.
bufP	IN	Pointer to new value. When setting one of the hsPrefSerialLib type preferences, this buffer should point to a null terminated library name string. When setting a serial library back to it’s default, pass in a pointer to an empty string.
prefSizeP	IN	Size of the new value, including null terminator.

The possible values of `pref` are:

Setting	Description
hsPrefSerialLibHotSyncLocal	Sets the name of the serial library to use for local HotSync.
hsPrefSerialLibHotSyncModem	Sets the name of the serial library to use for

	modem HotSync.
hsPrefSerialLibConsole	Sets the name of the serial library to use for the debugger console.
hsPrefSerialLibIrda*	Sets the name of the serial library to use for IRDA.
hsPrefSerialLibDef	Sets the name of the serial library to use for all other applications (i.e., any application that opens up the built-in serial port library named "Serial Library").

* This feature is not supported on Visor or Visor Deluxe.

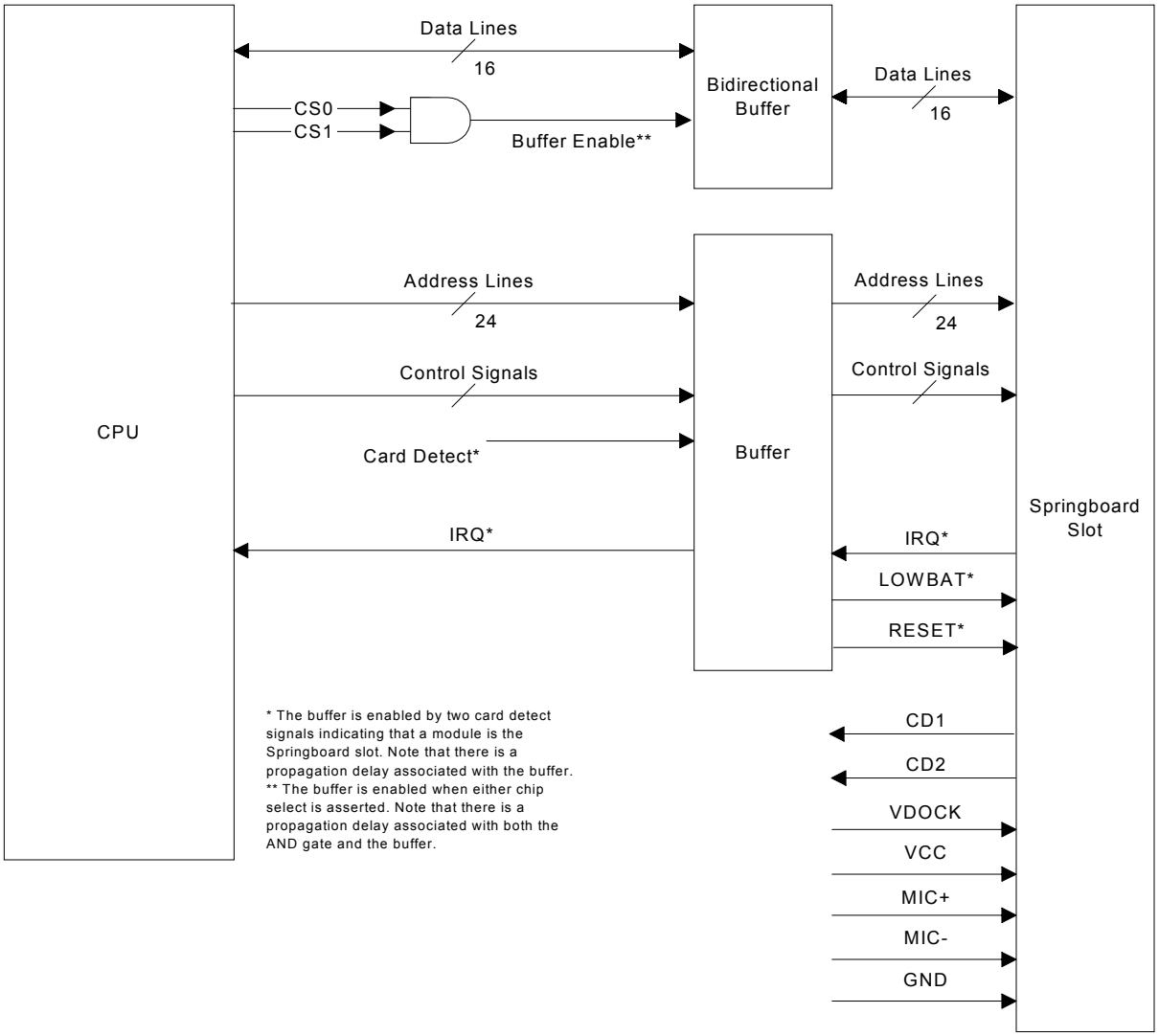
Returns

0	If no error
hsErrInvalidParam	Invalid pref parameter

5. Springboard Interface Pinout and Signal Description

The Springboard Expansion Slot allows for hardware and software expansion of Handspring's family of Palm OS-compatible handheld computers, and for seamless hot plug-and-play capability.

Figure 5-1. Block diagram of the slot interface



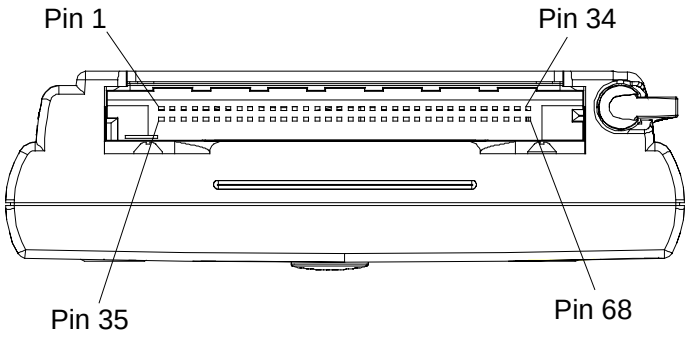
The Springboard Expansion Slot provides a slave-only interface for expanding the capabilities of the main handheld unit. This slot supports hot swapping via buffers and transceivers; otherwise, it functions as if directly connected to the host CPU bus. The core of the Springboard Expansion Slot connector is physically identical to a PCMCIA connector; however, the mechanical keying features and electrical specifications differ.

Modules can be inserted into or removed from the base unit at any time, even when the device is on. When a module is inserted, the software and hardware automatically configure the module, making its features instantly available. A module can be a simple ROM-only module with a set of additional applications, or it can provide additional hardware functionality, such as network connectivity, sound support, and communications options. A

variety of functions, such as pagers, radios, or backup flash memory can be interfaced with Handspring's handheld computers in this way.

Figure 5-2 illustrates the Springboard 68-pin expansion slot.

Figure 5-2. Springboard 68-pin Expansion Slot



5.1. Pinout

Table 5-1 summarizes the signal names with their respective pin numbers on the 68-pin expansion slot connector. Note that the signal direction (I/O/P/PU[1]) is viewed with reference to the module. For example, CS1* is driven by the CPU and is an input (I) to the module.

Table 5-1. Springboard Expansion Slot Connector Pin Summary

Pin	Name	I/O/P/PU[1]	Function	Pin	Name	I/O/P/PU	Function
1	GND	P	Module Ground	35	GND	P	Module Ground
2	D3	I/O	Data Bus	36	CD1*	O/PU	Card Detect
3	D4	I/O	Data Bus	37	D11	I/O	Data Bus
4	D5	I/O	Data Bus	38	D12	I/O	Data Bus
5	D6	I/O	Data Bus	39	D13	I/O	Data Bus
6	D7	I/O	Data Bus	40	D14	I/O	Data Bus
7	CS0*[2]	I	Chip Select	41	D15	I/O	Data Bus
8	A10	I	Address Bus	42	CS1*	I	Chip Select
9	OE*	I	Output Enable	43	Reserved		Reserved
10	A11	I	Address Bus	44	Reserved		Reserved
11	A9	I	Address Bus	45	Reserved		Reserved
12	A8	I	Address Bus	46	A17	I	Address Bus
13	A13	I	Address Bus	47	A18	I	Address Bus
14	A14	I	Address Bus	48	A19	I	Address Bus
15	WE*	I	Write Enable	49	A20	I	Address Bus
16	IRQ*	O/PU	Interrupt Request	50	A21	I	Address Bus
17	VCC	P	Module VCC	51	VCC	P	Module VCC
18	VDOCK	P	Docking Voltage	52	VDOCK	P	Docking Voltage
19	A16	I	Address Bus	53	A22	I	Address Bus
20	A15	I	Address Bus	54	A23	I	Address Bus
21	A12	I	Address Bus	55	Reserved		Reserved
22	A7	I	Address Bus	56	Reserved		Reserved
23	A6	I	Address Bus	57	Reserved		Reserved
24	A5	I	Address Bus	58	RESET*	I	Module Reset
25	A4	I	Address Bus	59	Reserved		Reserved
26	A3	I	Address Bus	60	MIC-	I	Microphone+/-
27	A2	I	Address Bus	61	MIC+	I	Microphone+/-
28	A1	I	Address Bus	62	Reserved		Reserved
29	A0	I	Address Bus	63	LOWBAT*	I	Low Battery
30	D0	I/O	Data Bus	64	D8	I/O	Data Bus
31	D1	I/O	Data Bus	65	D9	I/O	Data Bus
32	D2	I/O	Data Bus	66	D10	I/O	Data Bus
33	Reserved		Reserved	67	CD2*	O/PU	Card Detect
34	GND	P	Module Ground	68	GND	P	Module Ground

1. I = input, O = output, and P = power, with respect to the module. For example, the IRQ signal is driven by the module and is an output to the handheld. PU indicates the signal is internally pulled up within the handheld.

2. * indicates an active low signal.

5.2. Signal Descriptions

The signals for the 68-pin expansion connector are described below in alphabetical order. Active-low signals have an asterisk "*" at the end of their names.

Address Bus **A[23:0]**

Each of the two chip selects (CS0 and CS1) has direct access of up to 16MB. A total range of 32MB is addressable on Springboard. Address line A23 is the most significant address bit, and A0 is the least significant address bit. The default size of each region is 16MB and is software programmable. This bus is an input to the expansion slot; it is always driven during normal and sleep mode. The address bus is valid throughout the entire bus cycle.

Note: The Springboard Expansion Slot data bus is 16-bit only and memory accesses are conducted on even-byte boundaries.

Card Detect **CD1*, CD2***

CD1* and CD2* are active-low module detect signals that indicate to the handheld when the expansion module has been firmly seated into the Springboard Expansion Slot. The two Module Detect pins are physically shorter than all other pins on the expansion connector. On the host side, the signals perform two functions: 1) they interrupt the handheld to alert the CPU that a module has been inserted or removed, and, 2) they begin turning on the Vcc power supply. Depending on the electrical load on the module, VCC is valid within 5 ms. Both signals should be tied directly to GND on the expansion module.

Chip Select **CS0*, CS1***

These two active-low chip select signals control access to the two addressable regions on the module. The address space for CS0* is referred to as csSlot0; the address space for CS1* is referred to as csSlot1. In order for the Palm OS to recognize the module and its contents, use CS0* to access ROM or Flash. CS1* is optional and can be used to interface with additional ROM, Flash, UARTs, or other peripheral devices. Both chip select signals are asserted for the duration of the memory cycle. Only one of the two chip selects is valid for each module access. The address bus is guaranteed to be valid before and during the assertion of the chip select signal. Refer to Section 2.1, "Memory Space," for more information on the chip selects and their corresponding address spaces.

Data Bus **D[15:0]**

The data bus consists of 16 data lines, D[15:0]. D15 is the most significant data bit, and D0 is the least significant data bit. Only 16-bit operations are performed on the data bus.

Module Ground **GND**

GND is the ground connection to the module. All GND signals must be connected to the module's ground reference or plane.

Interrupt Request **IRQ***

The active low interrupt request is level-sensitive. This signal is output from the Springboard module whenever interrupt service is required from the handheld computer. There is no default interrupt service routine for the module, so the application resident on the expansion module must install the interrupt service routine (ISR) during initialization (see *Interrupts* for more information on interrupt handling). Interrupt acknowledgment is user-defined and must be accommodated by the expansion module application as well. The internal Visor interface has a pull-up resistor, so the module does not require one.

Low Battery Warning **LOWBAT***

The low battery warning signal indicates that the handheld's batteries are below a critical threshold or are being swapped out. When this signal is asserted, the expansion module is electrically "removed" to prevent data loss in the handheld. When the batteries are replaced or recharged, the module is "re-inserted."

Microphone +/- **MIC+,MIC-**

These two signals interface to the microphone on the handheld unit. These signals are a differential pair and are directly connected to an electret condenser microphone. Appropriate bias must be supplied by the module on the MIC+ signal. Please refer to the appropriate Product Guide for details on the microphone.

Output Enable **OE***

OE* is the active-low output enable, or read signal, for the module. When qualified with a low on either CS0* or CS1*, a low on OE* indicates a read cycle from the module. The address bus is valid before the assertion of OE*. The module can drive the data bus as soon as a chip select and OE* are asserted. Data must be driven for the entire cycle, as determined by the levels on the chip select and OE*. WE* is deasserted during read cycles.

Note: The chip selects are not guaranteed to be asserted prior to the assertion of OE*. Also the cycle is ended when either a chip select or OE* is deasserted.

Module Reset **RESET***

RESET* is an active low reset signal for the expansion module. During module insertion, RESET* is asserted while VCC is rising; it is held asserted for 30 ms minimum after the module is inserted to allow circuitry on the module sufficient time to stabilize. Because module power is guaranteed to be applied within 5 ms, the module will have a minimum valid reset signal of 25 ms. Application software can also assert this signal at any time to reset the module.

RESET does *not* assert LOWBAT* or remove power to VCC.

Module VCC **VCC**

VCC is the 3.0-3.6V power supply, which can be used to power the expansion module (some expansion modules can supply their own power). Power is not provided on these pins until the module is firmly seated in the slot and both module detects (CD1* and CD2*) are asserted. The power supply ramps up to VCC within 5 ms of insertion. The maximum current that can be supplied by the slot is 100 mA. Expansion module designs that use this power source must take into account what the users will see when LOWBAT* is asserted and the Springboard Expansion Slot power is removed.

Docking Voltage **VDOCK**

This pin could provide a charging supply to the module when the handheld is placed into a special charging dock. The handheld passes this charging supply from a pin on its cradle connector through to pins (VDOCK) on the Springboard expansion module connector. VDOCK provides a 4.75 – 6.2v on two pins with a maximum current of 500mA. Developers should ensure that VDOCK is not connected with Vcc.

Write Enable **WE***

WE* is the active-low write enable signal for the module. When qualified with a low on either CS0* or CS1*, a low on WE* indicates a write cycle to the module. The address bus is valid before the assertion of WE*. The data bus, driven by the host interface, is valid before the assertion of WE*. In addition, WE* is deasserted prior to the deassertion of the chip select. OE* is deasserted during write cycles.

Note: Because there is no separate write enable for each byte, all 16 bits of the data bus are written at the same time; thus, there is no support for byte writes to the expansion module.

Reserved

Saved for future use. These pins should remain unconnected.

6. Electrical Specifications

This chapter provides electrical and timing characteristics for the Springboard platform. Detailed electrical, mechanical, and environmental specifications for each product are detailed in the applicable Product Guide.

6.1. DC electrical characteristics

Table 6-1 below lists the DC electrical characteristics for Springboard modules.

Table 6-1. DC electrical characteristics

Symbol	Parameter	Min	Max	Unit
Vcc	Supply Voltage[1]	3.0	3.6	V
I_{cc}	Operating Current	–	100	mA
I_{s1}	Standby Current, LOWBAT* deasserted	–	100	µA
I_{s2}	Standby Current, LOWBAT* asserted[2]	–	10	µA
V_{IH}	Input High Voltage	2.0	Vcc + 0.5	V
V_{IL}	Input Low Voltage	0.0	0.8	V
V_{OH}	Output High Voltage (I_{OH} = 2.0 mA)	2.4	Vcc + 0.5	V
V_{OL}	Output Low Voltage (I_{OL} = -2.5 mA)	0.0	0.4	V
I_{IL}	Input Leakage Current (0V <= V_{IN} <= Vcc)	–	±5	µA
I_{OZ}	3-state Leakage Current (0V <= V_{OUT} <= Vcc)	–	±5	µA
V_{DOCK}	Docking Voltage (500mA max)	–	4.75 to 6.2v	V

1. In Visor, Handspring's first generation of handheld computer, the minimum supply voltage could actually be zero if LOWBAT* is asserted.

2. Since the Visor handheld (see Note 1) actually removes power from the Springboard Expansion Slot, this specification is for future product compatibility.

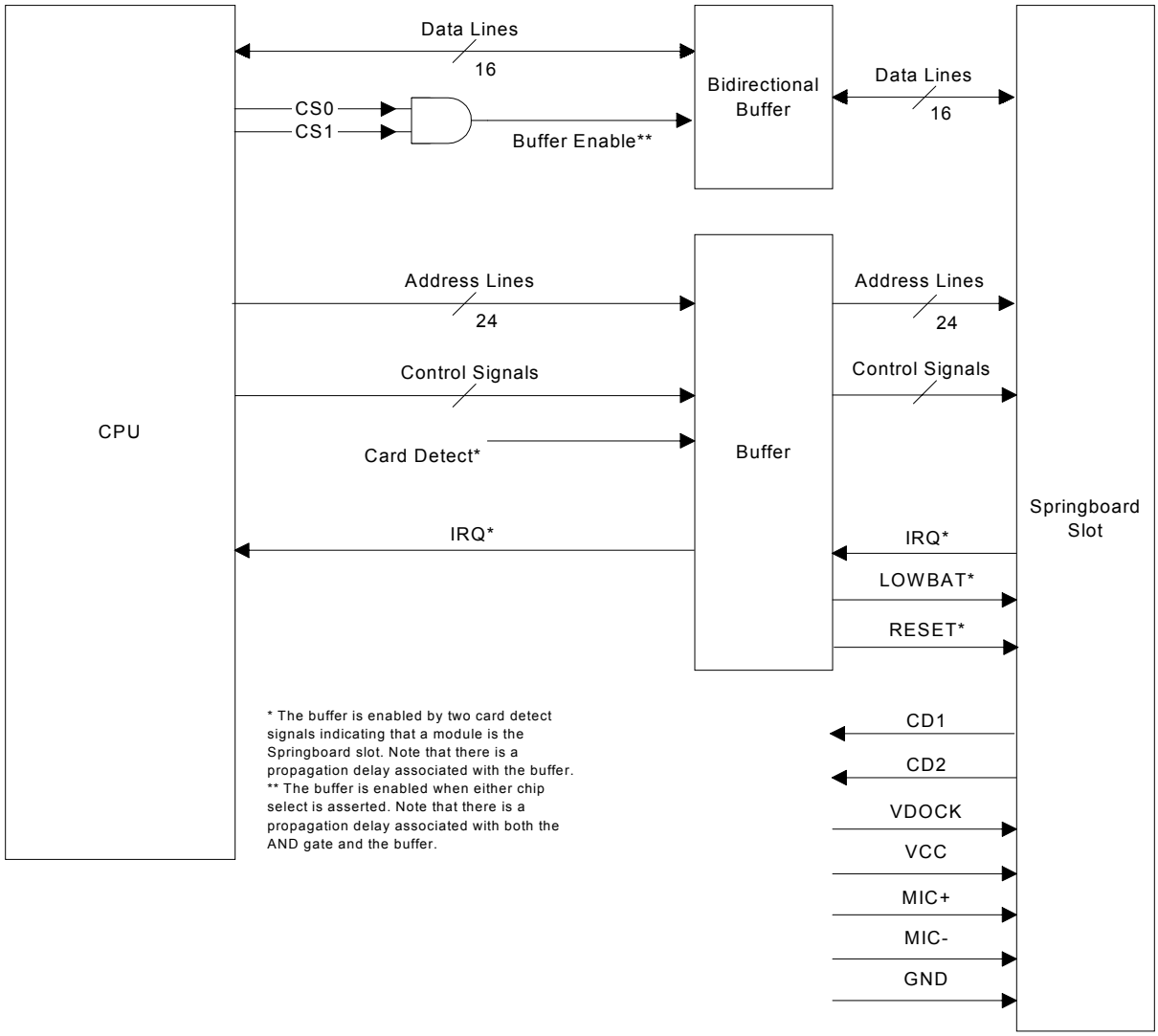
6.2. AC Characteristics

This section lists the AC timing parameters for Springboard modules.

6.2.1. General Information on Springboard Timing

Many signals on the Springboard port are CPU signals powered through buffers. To illustrate this point, the following diagram depicts the Springboard implementation on the Handspring Visor.

Figure 6-1. Springboard Implementation on Visor

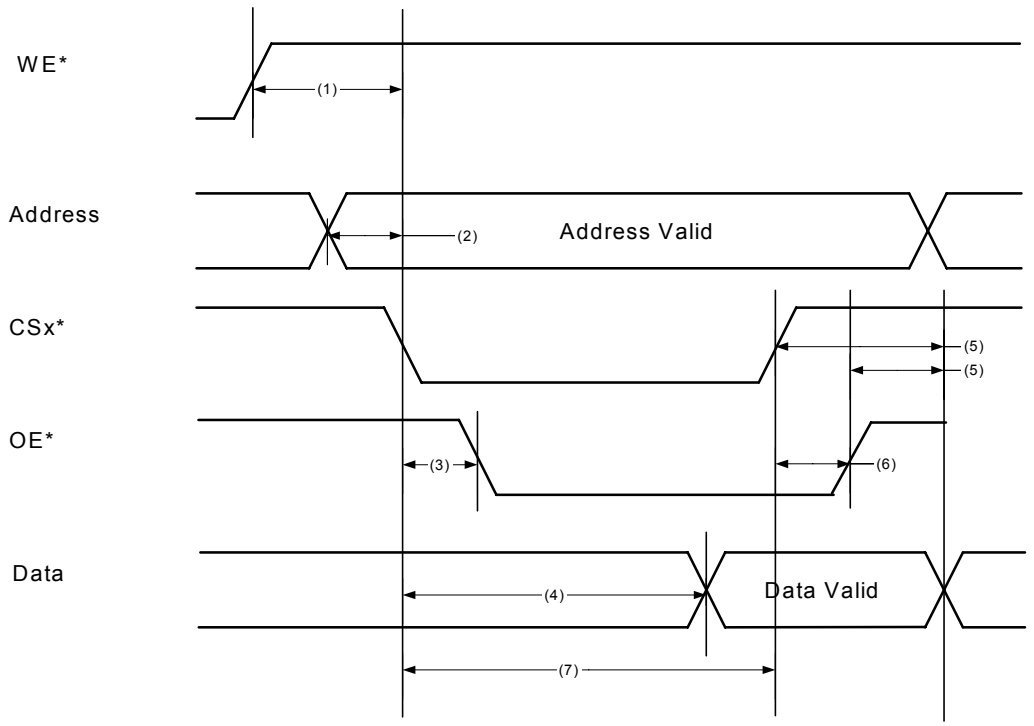


6.2.2. Read Cycle

Table 6-2. Read Cycle Details

Num	Parameter	Min[3]	Max[3]	Unit
1	WE* negated before cycle begins and CSx* is asserted	0	-	ns
2	CSx* asserted after address valid	5	-	ns
3	OE* asserted after CSx* asserted	-	15	ns
4	CPU expecting data after CSx* asserted	20 + T[1]	300	ns
5	Data hold required after CSx* or OE* negated	0	-	ns
6	OE* negated after CSx* negated	-10		ns
7	Access Time defined in ROM header file[2]	-	-	ns

1. T is the delay caused by inserting wait states. The operating system uses the Access Time (7) to setup the appropriate number of wait states.
2. Minimum and maximum Access Times are specific to each Handspring product and may vary with CPU model and clock speed. For example, some Handspring handheld computers can be configured with a longer access time. Access times beyond 300ns are specific to a handheld rather than part of the Springboard specification.
3. All timing is specified with a maximum 50pF capacitance.

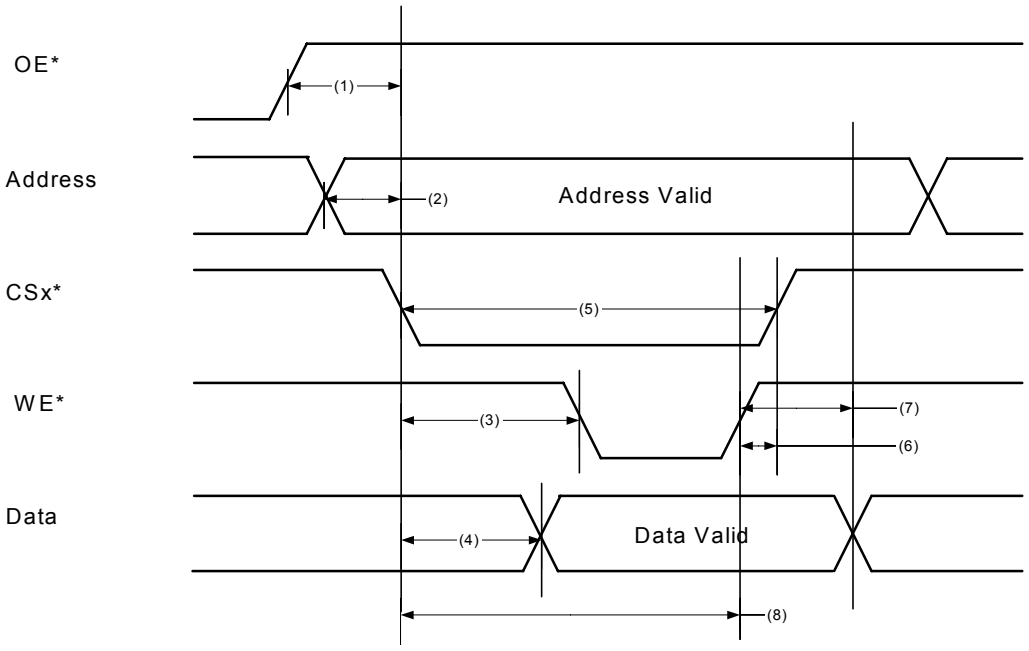


6.2.3. Write Cycle

Table 6-3. Write Cycle Details

Num	Parameter	Min[3]	Max[3]	Unit
1	OE* negated before cycle begins and CSx* is asserted	0	-	ns
2	CSx* asserted after address valid	5	-	ns
3	WE* asserted after CSx* asserted	-	70	ns
4	CPU data valid after CSx* asserted	-	40	ns
5	CSx* pulse width	25+T[1]	300	ns
6	CSx* negate after WE* negate	0	-	ns
7	Data hold after WE* negated	7.5	-	ns
8	Access Time defined in ROM header file[2]	-	-	ns

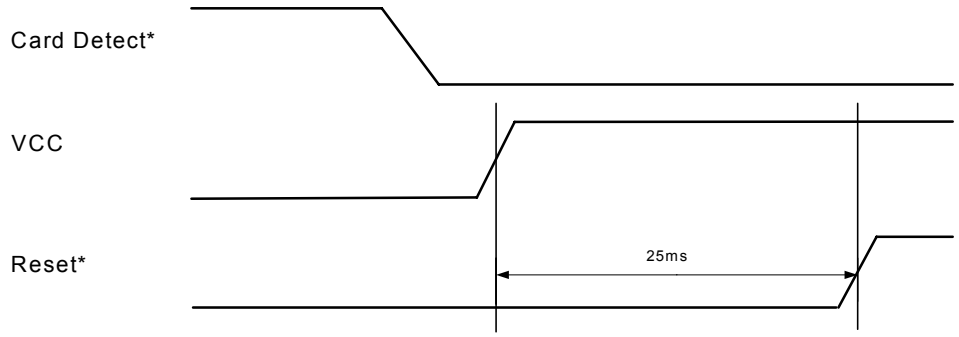
1. T is the delay caused by inserting wait states. The operating system uses the Access Time (7) to setup the appropriate number of wait states.
2. Minimum and maximum Access Times are specific to each Handspring product and may vary with CPU model and clock speed.
3. All timing is specified with a maximum 50pF capacitance.



6.2.4. Reset Timing

When a module is first inserted, an interrupt is generated to the handheld and power is slowly applied to the module. In the Handspring handheld computer, the OS ensures that the reset signal (RESET*) remains asserted for at least 30 ms before releasing it. Because the Springboard module power ramps up in approximately 5 ms, there is a guaranteed 25 ms of power-on reset time for the module. Reset timing is specified with a maximum 33µF capacitance.

If required, the module software can manually assert and release RESET* after the module has been inserted by setting the hsCardAttrReset attribute of HsCardAttrSet().



7. Mechanical Information

This section covers an overview of the mechanical aspects of the Springboard Expansion Slot. Mechanical information specific to each product will be covered in the Product Guides. Complete mechanical information referenced in these documents is available on Handspring's website in a variety of formats. Design files for Springboard modules, cradle assemblies, and the exterior dimensions of the Handspring™ handheld are included. Full mechanical files for modules, Handspring handheld computers, and cradles are located on the mechanical section of Handspring's developer website:

http://www.handspring.com/developers/dev_mechanical.jhtml

There are several general areas of interest to examine when designing a Springboard module:

- Springboard Connector
- Geometry of the Springboard slot
- Mechanical Interaction with the Handspring handheld
- Springboard Module Base Color

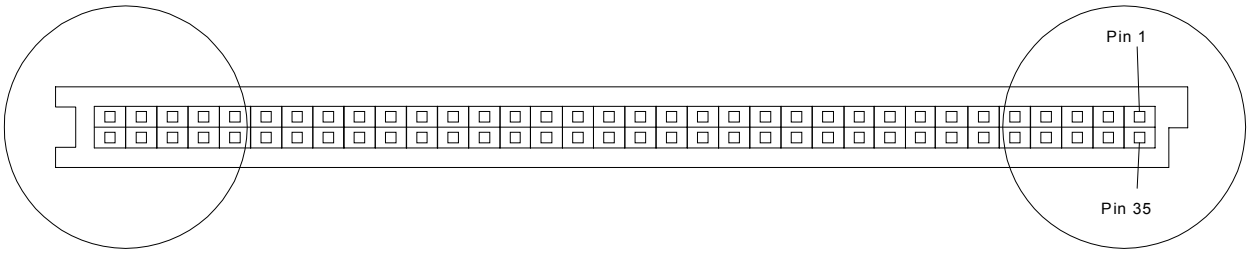
We'll cover each of these in more detail.

7.1. Springboard Connector

Familiarity with the Springboard connector will ensure a proper fit. The Handspring website contains CAD files that fully document various implementations of Springboard modules.

The connector core is a PCMCIA style, 68-pin connector with different keying features. These unique keying features are designed to prevent accidental insertion of an incompatible card. A close examination of the plastic housing that surrounds a Springboard module connector core will show the keying features as illustrated below.

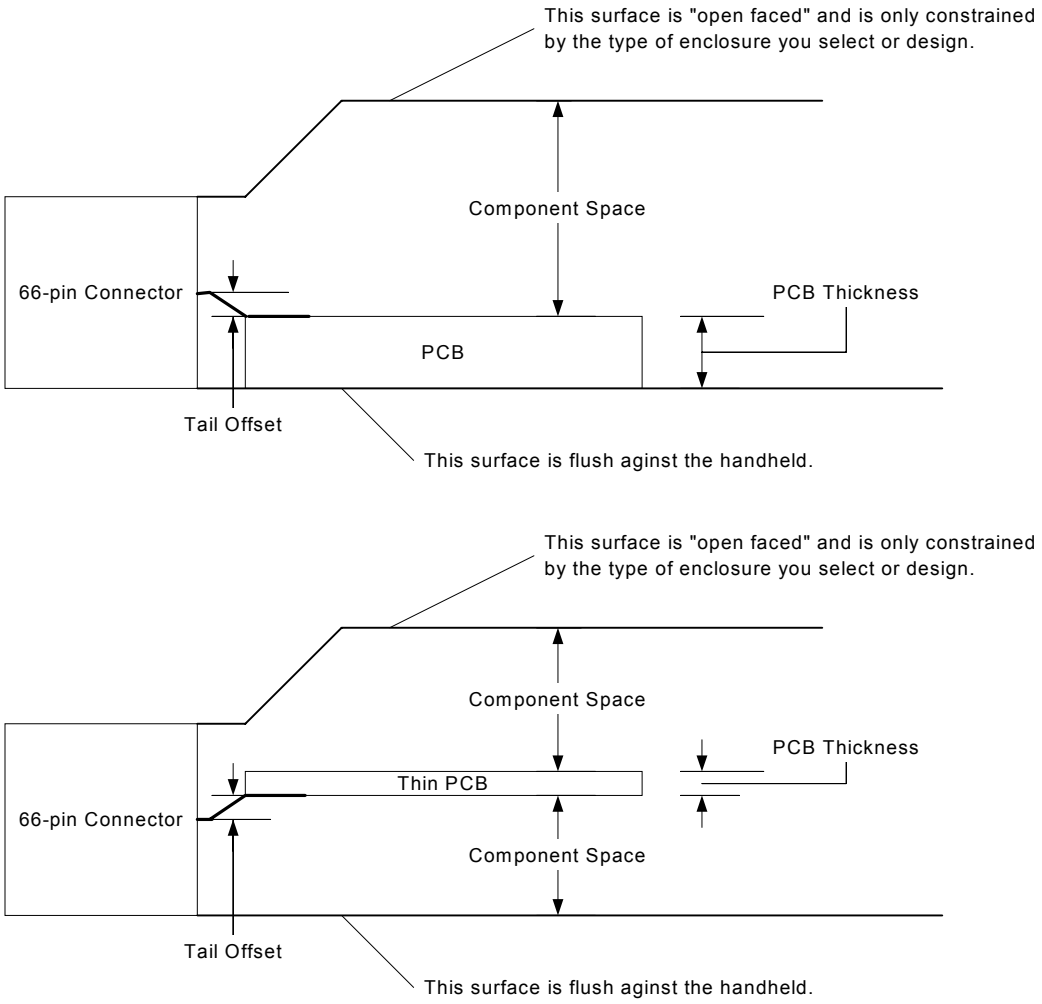
Figure 7-1. Springboard Connector Keying



Another area of interest associated with the Springboard connector relates to the various tail offsets available. Developers should select the tail offset driven by their design. Typical design considerations that drive the tail offset selection include:

- PCB thickness
- Top/bottom mount of connector onto PCB.

Figure 7-2. Connector Considerations



The following table lists the mechanical insertion/extraction force ratings for the Springboard Expansion Slot connector.

Table 7-1. Insertion/extraction force for Springboard Expansion Slot connector

Rating	Value	Unit
Insertion force[1]	1.5 - 6	pounds
Extraction force	1.5 - 5	pounds

1. These ratings were verified up to 3000 insertions/extractions of the module

7.2. Geometry of the Springboard Slot

The slight draft required by tooling for plastics introduces small angles into the Springboard slot in each Handspring handheld. The result is that the Springboard module reference plastics are not perfectly rectangular. The CAD files located on the Handspring website provide a reference design for developers who want to create completely custom plastic enclosures for their modules.

7.2.1. Mechanical Interaction with the Handspring Handheld

The final mechanical consideration involves how the module will interact with various Handspring handhelds, as these plastics may change from product to product. The CAD files located on the Handspring website contain detailed information on each handheld device and cradle. These CAD files include non-encroachment zones that highlight areas of concern.

In particular, module plastics should consider interaction with:

- Cradle
- Stylus holder
- Belt clip (if applicable)
- Reset hole
- Battery Door

7.3. Springboard Module Base Color

Color information for Handspring products may change from product to product. Please refer to the Handspring website for the latest information on plastic materials, textures, and colors. Details are updated within Application Notes in this section:

http://www.handspring.com/developers/tech_notes.jhtml

8. Compatibility Testing

8.1. Compatibility Testing Overview

Developers who wish to use the Springboard-compatible logo must ensure that their products conform to the Springboard software, electrical, and mechanical specifications. Developers are responsible for self-certifying their products. Complete details are located on Handspring's website here:

http://www.handspring.com/developers/compatibility_testing.jhtml

9. Springboard Trademarks and Logos

9.1. Springboard Trademarks and Logos Overview

This section of the document defines the specifications for the Springboard™ symbol and colors. It also provides a view of available Springboard logos. Full details on trademarks and logos for the Springboard™ platform along with the latest guidelines can be found on our website at.

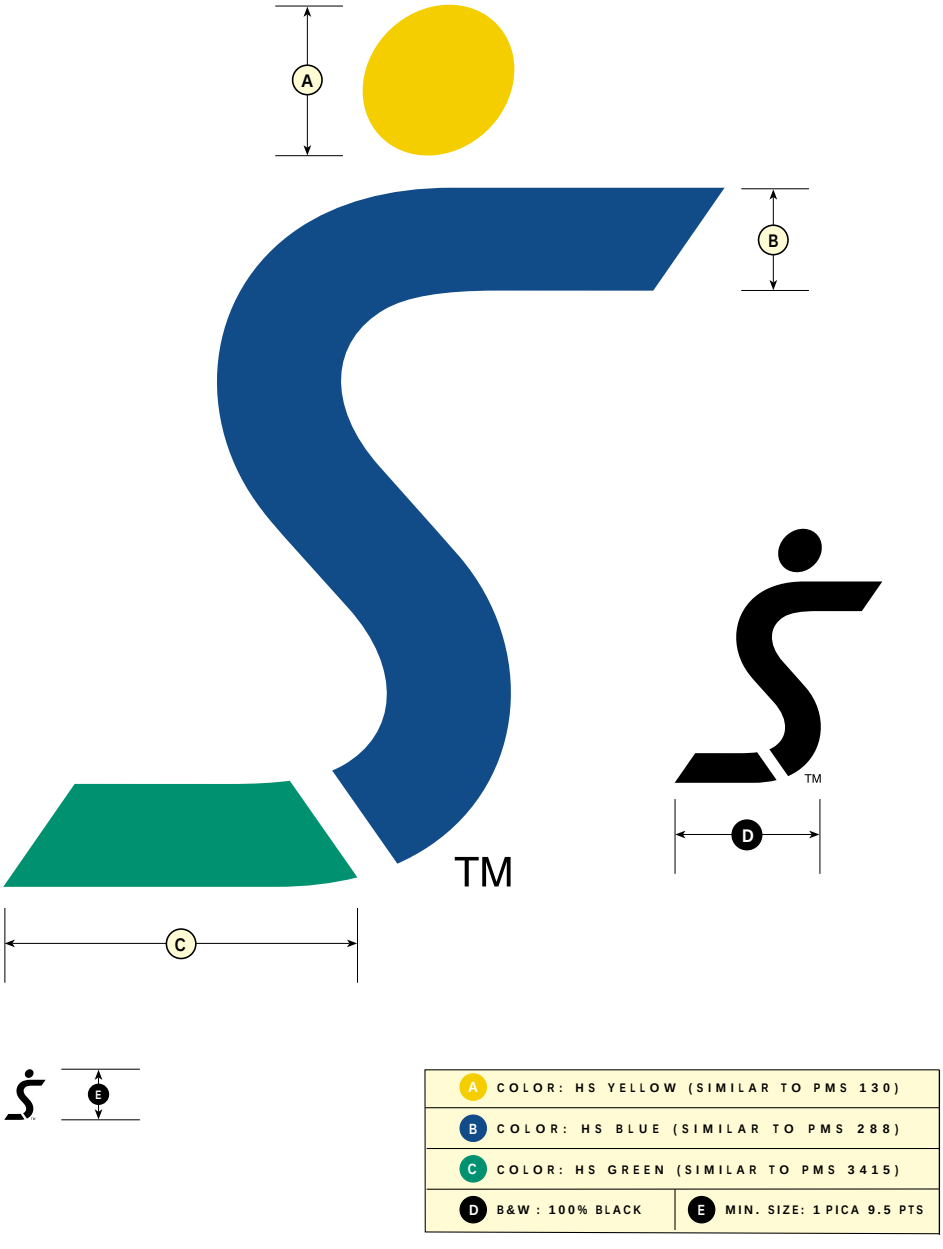
http://www.handspring.com/developers/dev_logos.jhtml

9.2. Trademarks

SPRINGBOARD SYMBOL & COLOR

The Springboard symbol was designed to capture the modularity and flexibility of our Springboard technology. Shown here are the color and black and white versions of the symbol. As with our corporate symbol, never alter or reproportion the Springboard symbol in any way.

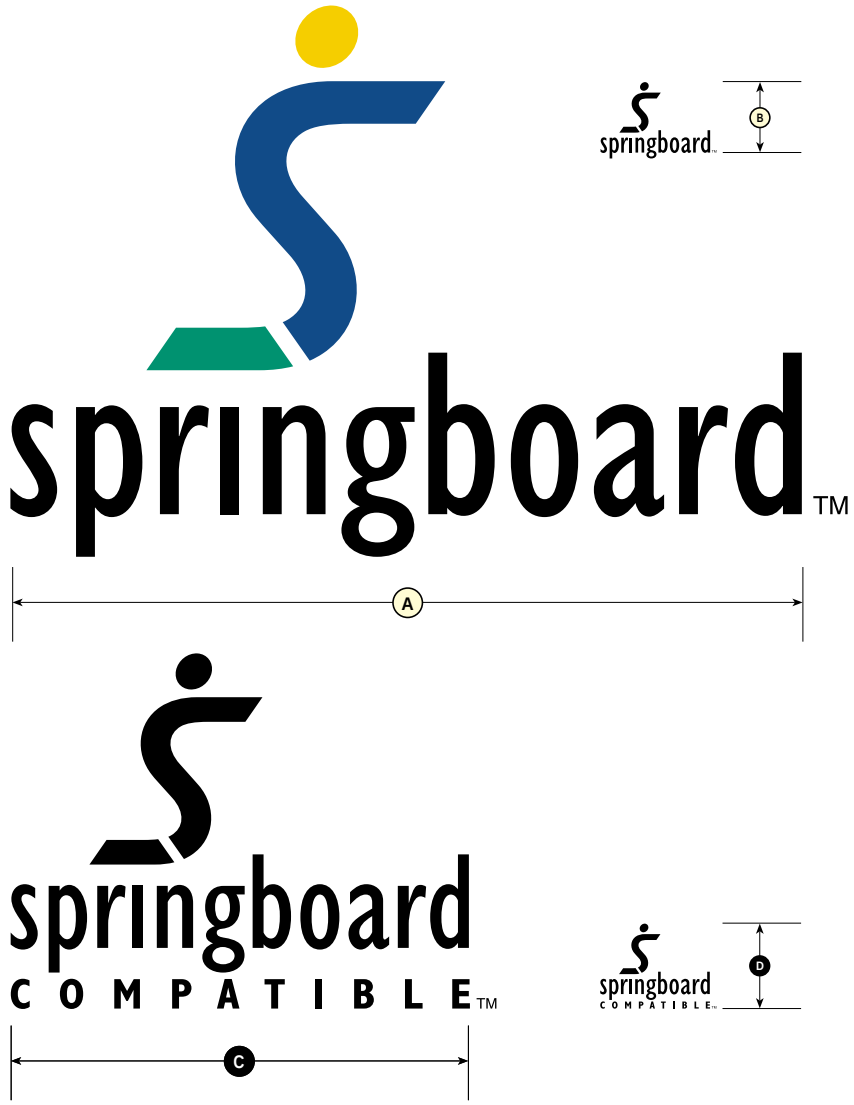
Figure 9-1. The Springboard Symbol



SPRINGBOARD SIGNATURES

The Springboard signature includes the symbol and the Springboard logotype used together in a specific orientation, as shown. A variation on this signature, the Springboard Compatible signature, also is featured. As with our corporate signature, consistent use of the elements that comprise the Springboard and Springboard Compatible signatures, helps strengthen and reinforce our brand.

Figure 9-2. Springboard Signatures



9.3. Logos

Figure 9-3. Color.springboard.gif

Figure 9-4. Color.tag.springboard.gif

Figure 9-5. SB.symbol.eps

Figure 9-6. SB.color.eps

Figure 9-7. Sbcom.color.eps

Figure 9-8. Symbol.rev.eps

Figure 9-9. SB.B&W.eps

Figure 9-10. Sbcom.B&W.eps

Figure 9-11. BW.springboard.gif

Figure 9-12. BW.tag.springboard.gif

10. Handspring Developer Agreement

Handspring Licensing

HANDSPRING, INC.

Developer Agreement

PLEASE READ THE TERMS OF THE FOLLOWING AGREEMENT CAREFULLY. BY USING THE MATERIALS DISTRIBUTED WITH THIS AGREEMENT (THE "DEVELOPMENT KIT"), YOU ARE AGREEING TO BE BOUND BY THE TERMS AND CONDITIONS OF THIS AGREEMENT. IF YOU DO NOT AGREE TO THE TERMS AND CONDITIONS OF THIS AGREEMENT, PLEASE DO NOT USE THE DEVELOPMENT KIT. INSTEAD, PLEASE DESTROY ALL COPIES OF THE DEVELOPMENT KIT WHICH YOU MAY HAVE.

Definition. "Springboard(TM) Enabled Products" are Handspring handheld computers that contain an external "slot" (the "Springboard(TM) slot") into which compatible third party hardware or software products can be inserted.

License Grant. Subject to the terms and conditions of this Agreement, Handspring hereby grants to Developer a non-exclusive, non-transferable license under Handspring's intellectual property rights in the Development Kit (a) to use, reproduce and create derivative works of the materials provided by Handspring under this Agreement, solely internally in connection with Developer's development and manufacture of i) products which plug into the Springboard(TM) slot and meet Handspring's Springboard(TM) compatibility requirements ("Licensed Plug-Ins") or ii) accessory products (such as keyboards or reference manuals) for use with Springboard(TM) Enabled Products (such plug-in products and accessory products, collectively, "Licensed Products"); (b) to make, have made, use, distribute and sell Licensed Products directly or indirectly to end users for use with Springboard(TM) Enabled Products; and (c) to distribute the unmodified Development Kit in its entirety (including this Agreement) to third parties who agree to be bound by the terms and conditions of this Agreement.

LICENSE RESTRICTIONS. Except as otherwise expressly provided under this Agreement, Handspring grants and Developer obtains no rights, express, implied, or by estoppel, in any Handspring intellectual property, and Developer shall have no right, and specifically agrees not to (a) transfer or sublicense its license rights to any other person; (b) decompile, decrypt, reverse engineer, disassemble or otherwise reduce the software contained in the Development Kit to human-readable form to gain access to trade secrets or confidential information in such software, except and only to the extent such activity is expressly permitted by applicable law notwithstanding such limitation; (c) use or allow others to use the Development Kit, in whole or part, to develop, manufacture or distribute any products other than Licensed Products; (d) use or allow others to use the Development Kit, in whole or part, to develop, manufacture or distribute products (including Licensed Products) for use as a plug-in or accessory to any product other than Springboard(TM) Enabled Products; (e) use or allow others to use the Development Kit, in whole or part, to develop, manufacture or distribute any products incorporating an external or internal slot design; or (f) modify or create derivative works of any portion of the Development Kit.

Ownership. Handspring is the sole and exclusive owner of all rights, title and interest in and to the Development Kit, including, without limitation, all intellectual property rights therein. Developer's rights in the Development Kit are limited to those expressly granted hereunder. Handspring reserves all other rights and licenses in and to the Development Kit not expressly granted to Developer under this Agreement. Subject to Handspring's rights in the Development Kit and the Springboard(TM) Enabled Products, Developer shall retain all rights in the Licensed Products developed by Developer in accordance with this Agreement.

Compatibility Testing AND Branding. Prior to Developer's use of Handspring's Springboard(TM) compatibility trademark (the "Mark") in connection with a Licensed Plug-In, Developer shall conduct reasonable testing in accordance with Handspring's compatibility testing guidelines to ensure that the Licensed Plug-In conforms in all respects to Handspring's Springboard(TM) compatibility requirements (the "Compatibility Criteria"). Developer agrees that it will not use the Mark or make any statements claiming or implying compatibility with the Springboard(TM) slot in connection with any Licensed Plug-Ins which have not passed such compatibility testing

and that, if Handspring determines that any Licensed Plug-In is not compliant with the Compatibility Criteria, Developer shall immediately cease use of the Mark in connection with that Licensed Plug-In. All goodwill generated by Developer's use of the Mark shall inure to Handspring's benefit.

Subject to the terms and conditions of this Agreement, Handspring hereby grants to Developer a non-exclusive, non-transferable license to use, subject to the guidelines set forth in Handspring's trademark policy and other applicable guidelines, (i) Handspring's Springboard(TM) compatibility trademark solely in connection with the marketing and sale of Licensed Plug-ins which comply with the Compatibility Criteria; and (ii) artwork, icons, logos, color schemes, and other industrial designs and designations of source provided by Handspring to Developer hereunder solely in connection with the marketing and sale of Licensed Products

Developer Indemnification. Developer will defend at its expense any action brought against Handspring to the extent that it arises from or relates to Developer's development, manufacturing, marketing or distribution of Licensed Products, and Developer will pay any settlements and any costs, damages and attorneys' fees finally awarded against Handspring in such action which are attributable to such claim; provided, the foregoing obligation shall be subject to notifying Developer promptly in writing of the claim, giving it the exclusive control of the defense and settlement thereof, and providing all reasonable assistance in connection therewith. Notwithstanding the foregoing, Developer shall have no liability for any claim of infringement to the extent required by compliance with the Compatibility Criteria.

Warranty Disclaimer. HANDSPRING MAKES NO WARRANTIES, EITHER EXPRESS OR IMPLIED, AS TO ANY MATTER WHATSOEVER, AND SPECIFICALLY DISCLAIMS ALL WARRANTIES OR CONDITIONS OF MERCHANTABILITY, NONINFRINGEMENT, OR FITNESS FOR ANY PARTICULAR PURPOSE.

Limitation of Liability. EXCEPT FOR BREACHES OF THE SECTIONS ENTITLED "LICENSE GRANT", OR "LICENSE RESTRICTIONS", IN NO EVENT WILL EITHER PARTY BE LIABLE TO THE OTHER FOR LOST PROFITS, LOST BUSINESS, OR ANY CONSEQUENTIAL, EXEMPLARY OR INCIDENTAL DAMAGES ARISING OUT OF OR RELATING TO THIS AGREEMENT, REGARDLESS OF WHETHER BASED IN CONTRACT OR TORT, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

TERM AND TERMINATION. This Agreement shall remain in effect for the partial calendar year ending on the first March 31 following the effective date, and shall automatically renew for additional one (1) year terms ending on each subsequent March 31, except that the Agreement shall automatically terminate if either party materially breaches or is in default of any obligation hereunder or if either party provides notice of non-renewal by January 1. The parties agree that Handspring may provide notice by making the notice available in a manner similar to the manner in which the Development Kit was made available.

General. This Agreement will be governed by and construed and interpreted in accordance with the internal laws of the State of California, excluding that body of law applicable to conflict of laws. No waiver, amendment or modification of any provision hereof or of any right or remedy hereunder will be effective unless made in writing and signed by the party against whom such waiver, amendment or modification is sought to be enforced. No failure by any party to exercise, and no delay by any party in exercising, any right, power or remedy with respect to the obligations secured hereby will operate as a waiver of any such right, power or remedy. Neither this Agreement nor any right or obligation hereunder may be assigned or delegated by Developer (including by operation of law) without Handspring's express prior written consent, which consent will not be unreasonably withheld, and any assignment or delegation without such consent will be void. This Agreement will be binding upon and inure to the benefit of the successors and the permitted assigns of the respective parties hereto. If any provision of this Agreement is declared by a court of competent jurisdiction to be invalid, void, or unenforceable, the parties will modify such provision to the extent possible to most nearly effect its intent. In the event the parties cannot agree, then either party may terminate this Agreement on sixty (60) days notice. This Agreement constitutes the entire understanding and agreement of the parties hereto with respect to the subject matter hereof and supersedes all prior agreements or understandings, written or oral, between the parties hereto with respect to the subject matter hereof.