TRGpro™

Development Kit for the TRGpro Handheld Computer

P/N 1009-00020

Revision 2
Revision Date: 11/12/99

CONTACT INFORMATION

TRG Products, Inc.	2851 104th St., Suite H Des Moines, IA 50322 U.S.A. 515-252-7525 www.trgpro.com
Palm Computing World Wide Web	www.palm.com
Metrowerks World Wide Web	www.metrowerks.com

TABLE OF CONTENTS

ABOUT THIS DEVELOPMENT KIT …

This document is intended to introduce the hardware and software developer to the TRGpro, and to act as an overview of the resources that TRG offers to developers. This document is intended to give developers and enterprise clients an understanding of the features of TRGpro.

This development kit includes the information necessary to take full advantage of the features and API (Application Programs Interface) calls found in the TRGpro, including:

Chapter	Description
FAT File System Library	An API reference that contains descriptions of the library function calls used to access a file system compatible with Microsoft MS-DOS, Windows, Macintosh, Linux and others.
CompactFlash Library	An API reference that describes library function calls used to access all types of CompactFlash cards.
Audio Library	An API reference that describes library function calls used to control the enhanced audio functions in the TRGpro.
Hardware	Reference material that contains detailed description of the TRGpro hardware. This section is targeted towards hardware developers who are interested in developing accessory products for the TRGpro.

ADDITIONAL DOCUMENTATION

PALM COMPUTING DOCUMENTATION

For developers new to the Palm Computing® platform, the best introductory information can be found in the Development Zone section of www.palm.com. It includes guides to application programming for the Palm OS. Resource works include:

- Palm OS SDK Reference
- Palm OS Programmer's Companion
- Palm OS 3.0 Tutorial
- Debugging Palm OS Applications

COMPACTFLASH DOCUMENTATION

In addition to the information contained in this document, TRGpro developers are encouraged to access the CompactFlash Association (CFA) web site for additional information about developing CompactFlash devices. The CFA web site can be found at http://www.compactflash.org. From this site, developers can:

- Read general information the CompactFlash Association (CFA)
- Download the latest CompactFlash and CF+ specifications
- Read FAQs about CompactFlash
- Identify companies that are developing CompactFlash products
- Search for CompactFlash products by category
- Read CFA press releases and view the CFA event calendar
- Find information about contacting or joining the CFA

CHAPTER ONE - TRGPRO OVERVIEW

This is the development kit for the TRGpro, the first handheld computer designed to meet the industry standards required by mobile professionals. Its design features include …

- **Palm OS.** The TRGpro handheld computer runs the industry-standard operating system for PDA's … the Palm OS. TRGpro is a complete Palm Computing® platform solution, able to run thousands of third party applications. Support for new hardware features of the TRGpro are built in to the OS, and complement the operation of existing software.

- **CompactFlash™ Expansion Slot**. While maintaining a form factor consistent with standard Palm OS devices, the TRGpro is designed with an expansion slot for CompactFlash (CF) Type I and Type II cards. CF cards have emerged as the standard for handheld computers and include a wide range of peripherals, including memory cards, modems, serial cards, bar code readers, and wireless devices.

- **MS Windows™ Compatible FAT File System**. The TRGpro handheld computer implements a file system for CF memory cards that is compatible with Microsoft Windows operating systems. This allows users to easily transfer files between their desktop and their TRGpro handheld computer.

- **Improved Sound.** The TRGpro handheld computer includes advanced audio amplification circuitry, a built-in speaker, and OS support to control these new features.

- **Flash Memory Support for OS Storage.** The TRGpro utilizes Flash memory to store the Operating System and all built-in applications. Updates to the OS are easily accomplished through the use of tools developed by TRG. Developers can customize the contents of this memory with TRG tools such as IMAGEpro.

TRGpro Design Goals

The TRGpro handheld computer was designed with the goals of bringing enhanced functionality and industry-standard expandability to the 3Com Palm community. Retaining 100% software compatibility with the standard Palm Operating System (OS) and as much hardware compatibility as possible with the PalmIII line of handheld computers was also a requirement.

The TRGpro was created by redesigning the PalmIIIx hardware to include:

- CF+ Type II expansion slot
- Extended internal memory addressing capability
- Enhanced audio output
- Improved power supply

This was accomplished while maintaining the basic size and shape of the PalmIII product line.

CompactFlash Overview

CompactFlash (CF) defines a class of small, removable devices for mobile computers and electronic equipment. CF cards first gained popularity as photographic storage devices for digital cameras. CF cards quickly emerged as the standard for handheld computers as well. The reasons for this rapid acceptance include:

- CF cards are **Compact**. They are approximately the size of a matchbook and weigh about half an ounce.
- CF storage cards are designed with **Flash**, and other non-volatile technology, which allows information to be stored without battery back up.
- CF storage cards provide the capability to easily transfer all types of information between a large variety of digital systems. For example, when fitted with an adapter, CF cards are compatible with the PC Card (PCMCIA) standard. This allows a CF memory card to be accessed as a standard FAT drive.
- CF cards include a wide range of peripherals, including memory cards, modems, serial cards, bar code readers, and wireless devices.
- The CompactFlash standard is an **open standard** controlled by a consortium of companies working together to promote the use of CompactFlash across a broad spectrum of devices. The CompactFlash Association (www.compactflash.org) currently has 143 member companies.

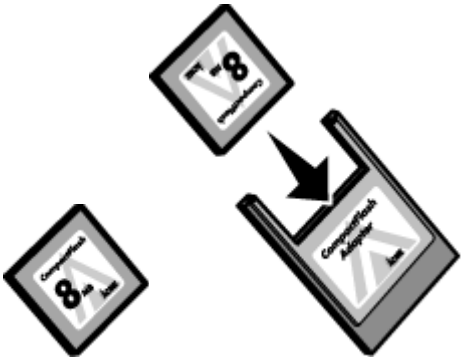
The TRGpro handheld computer features a CompactFlash Type II expansion slot, which can accept the original Type I (3.3 mm thick) and the Type II (5 mm thick) cards. CompactFlash Type I and II cards give your handheld computer additional capabilities, such as removable storage or a second communications port.

CF Storage Cards

CF storage cards provide a means of secondary storage for the TRGpro handheld computer. Storage cards provide the capability to easily transfer digital information between your TRGpro handheld computer and your personal computer. TRG has enhanced the Palm OS to directly support reading and writing to standard FAT File System (FFS) based cards.

The TRGpro uses industry-standard CF+ cards and brings industry-standard FAT file support to a Palm Computing® platform device. This allows TRGpro owners to use these cards in, and share information with, a wide variety of other products. Some of these products include:

- Digital cameras
- Laptop and desktop PCs running Windows or MacOS
- MP3 players
- WinCE devices

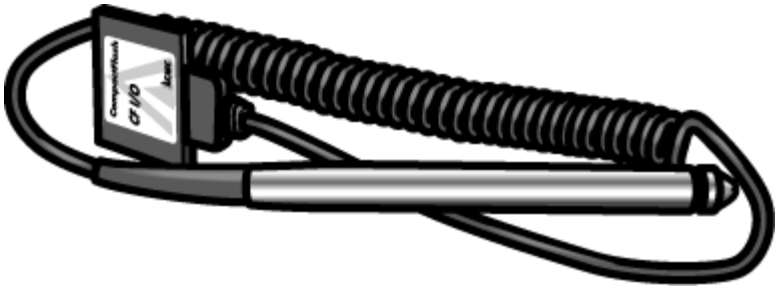


The TRGpro utilizes an application program called CFpro to enable user access to any FFS formatted CF storage card. The application provides users with the ability browse through the directory structure of the card, and to perform the following functions:

- Move files
- Copy files
- Delete files
- Rename files
- Make directories
- Format cards
- Play wave files

Most computer systems directly support CF storage cards through their OS. Laptop computers have PC Card slots that accept CF cards via a simple adapter (widely available from companies such as SanDisk, Pretec, and others). Simply plug the storage card into the computer and it appears as another drive in the system.

ADDITIONAL CF CARD TYPES



CF cards also bring new versatility to the TRGpro handheld computer by allowing for functions to be added to the host system. Check out www.trgpro.com for a list of recommended CF devices for the TRGpro unit. Some CF devices may require an additional software driver. Information on these drivers will be available from the web site as well.

Developers interested in creating new types of CF cards for use in the TRGpro handheld computer should first get a copy of the document "CF+ And CompactFlash Specification Revision 1.4" from the CompactFlash Assocation (see www.compactflash.org). Developers should also consult this document for details on the TRGpro implementation.

SOFTWARE OVERVIEW

ACCESSING TRGPRO FEATURES

The TRGpro utilizes a modified Palm OS ver 3.3 operating system. New features are accessible through the use of three libraries that encapsulate all of the unique aspects of TRGpro:

- CF Library
- FFS Library
- Audio Library

Developers can use these software libraries to create exciting new application programs to take advantage of the non-Palm features of TRGpro. Software support and drivers for new CF peripherals can be written via these libraries as well.

The recommended software development system for using these software libraries is CodeWarrior for PalmOS from Metrowerks (http://www.metrowerks.com). CodeWarrior for PalmOS is an integrated PalmOS development system that includes a compiler, linker, assembly-level debugger, and Palm emulator.

CF LIBRARY

The CF library provides an interface to the TRGpro CF slot hardware. It includes low-level services such as turning the slot on and off, setting the bus width, and initializing the IRQ line. The primary purpose of the CF library is to hide the TRGpro CF-interface hardware details from applications/drivers wishing to use CF cards. The CF library does not provide CF card-specific functions, such as parsing CIS tuples or setting configuration registers.

FFS LIBRARY

The FAT File System (Ffs) shared library provides an interface to Compact Flash (CF) storage cards containing a FAT File System. The interface is based upon the unbuffered file/disk system and library calls typically used with the C language. Support is provided for manipulating both files and directories, simplifying the exchange of data between the TRGpro and a personal computer. In addition, the high-capacities of existing CF cards allow Ffs-aware applications to create, read, and modify files much larger than the total storage space available on existing Palm devices. A document reader, for example, could access documents directly on a CF card, without first having to move the documents in the Palm device RAM or Flash.

AUDIO LIBRARY

The Audio library provides an interface to the enhanced audio circuitry in the TRGpro. It includes routines to control the master volume as well as routines to play DTMF tones.

DEVELOPING BASIC PALMOS APPLICATIONS

This development kit does not contain information regarding the design and implementation of standard PalmOS applications. For this information, please refer to the PalmOS SDK documentation provided by 3Com at http://www.palm.com/devzone/. Revelant documents include:

- Palm OS SDK Reference
- Palm OS Programmer's Companion
- Palm OS 3.0 Tutorial
- Debugging Palm OS Applications

CHAPTER TWO – TRGPRO HARDWARE

SECTION 1 – ELECTRICAL INFORMATION

OVERVIEW

The TRGpro is a small battery-powered handheld computer which runs the Palm Operating System (Palm OS ver 3.3). The heart of the system is a Motorola microprocessor that is directly connected to on-board Flash memory and DRAM. The TRGpro handheld computer displays information to a user on a small LCD screen. Its main user input mechanism is a touch screen interface but it also incorporates a small 7-key keypad that a user can press to launch specific applications. The TRGpro features two serial data I/O interfaces, an RS-232 port and an IrDA transceiver.

In addition, the TRGpro handheld computer extends the platform by adding …

- **CompactFlash expansion slot.** The TRGpro handheld computer incorporates a user-accessible CF slot that can be used with a wide variety of off-the-shelf CF cards. The TRGpro has built in support for FAT File System formatted CF storage cards.

- **Enhanced Power Supply.** The TRGpro includes a modified power supply designed to meet the increased current required to operate CF cards, and to enable louder sound production.

- **Improved Sound.** The design of the TRGpro includes advanced audio amplification circuitry, and an enhanced built-in speaker.

CF+ EXPANSION SLOT

An industry-standard CF+ type II card slot and support circuitry in the TRGpro replaces the proprietary 3Com memory board connector used in the PalmIIIx. The standard 3.3V switching power supply found in the PalmIIIx is also replaced with one capable of delivering additional current to the wide variety of CF+ cards available for the TRGpro. Hardware buffers were added to ensure that any CF+ type II device could safely be inserted or removed from the TRGpro at any time without risk of damaging the unit.

The expansion slot in the TRGpro handheld computer is CF+ V1.4 power level 0 compliant and is capable of operation with Type I and Type II CF+ memory and I/O cards. See

www.compactflash.org for information about obtaining complete details on the CompactFlash standards.

The CF+ slot is isolated from the CPU address and data buses by buffers. The address/control buffer is unidirectional and can be either enabled or tri-stated. The 16-bit bi-directional data buffer can be placed in tri-state, straight-through or byte-swapped modes. The buffers for the CF+ slot are controlled by TRG OS extensions (See the CF Library chapter). Hardware isolates the CF+ slot from the system bus to remove load in the event of the loss of main battery power.

The CF+ slot is powered from the 3.3V supply of the TRGpro. Power to the slot can be switched on or off through a low-on-resistance PFET. Continuous currents up to 250mA (500mA peak) can be delivered to a CF+ device. The PFET switch is controlled by software extensions to the OS. Hardware will remove power from the CF+ slot if battery power is lost.

The DragonBall EZ cannot directly produce CF+ bus cycles. For this reason, a state machine, implemented in the CPLD, is used to generate the necessary CF+ control signal timing. The state machine is used to generate the timing of control signals between the processor and the CF+ device, ensuring that the data setup and hold timing requirements of both devices are met. The state machine also performs the cycle stretching needed to accommodate slower CF device accesses.

The CF+ card interface registers are mapped into the memory space of the DragonBall EZ CPU using the CSB0- chip select. The CF+ extensions to the PalmOS configure this chip select to access a predetermined region of memory at boot up. Various CF+ registers are accessed by performing CPU memory cycles to this predetermined memory region. Use the function CfBaseAddress() to obtain the base address of CSB0-.

CF+ cards require only eleven address bits (A10..A0), or a total of 0x800 bytes of address space. However, to fully implement the CF+ specification using the DragonBall EZ, three additional address lines were used: A11, A12 and A13. This brings the total address space used in the TRGpro for CF+ access to 0x3000 bytes.

CF+ cards can be operated in three distinct modes:

- memory mode
- I/O mode
- true IDE mode

True IDE mode is used to allow CF+ storage cards to easily interface to hardware platforms containing IDE controllers such as notebook computers and PCs. Since the TRGpro handheld computer contains no such IDE controller, true IDE mode is **not** supported by the TRGpro. Memory mode is used for CF+ storage devices and I/O mode is used for CF+ I/O devices such as modems and Ethernet controllers. TRGpro fully supports CF+ memory and I/O modes.

The CF+ standard specifies that separate read and write control signals be used to distinguish between CF+ memory and I/O cycles. However, the DragonBall EZ CPU can only generate memory cycles and does not contain separate I/O control signals as do Intel-based CPUs. To solve this problem in the TRGpro, two separate address spaces were created. CPU address line A11 is used to break the CF+ address space into memory and I/O space. CPU memory accesses activating CSB0- with A11 low result in CF+ memory cycles being issued to the CF+ card. Similar accesses with A11 high result in CF+ I/O cycles being issued to the CF+ card.

CF+ cards contain a hardware control signal, REG-, used to differentiate between two distinct memory spaces within the card. When REG- is asserted (low) the register space (aka attribute space) is accessible. When REG- is de-asserted (high) common memory space is accessible. The TRGpro connects CPU address line A12 to the CF+ REG- signal. CPU memory cycles

activating CSB0- with A12 low, access CF+ register space. Similar cycles in which A12 is high, access CF+ common memory space.

CF+ cards contain a hardware reset signal which is used to place the card into a known initial state. The TRGpro allows software to directly manipulate the CF+ reset signal. This is accomplished by using CPU address line A13. CPU memory cycles activating CSB0- with A13 high cause the CF+ card to be reset. The function CfReset() should be used to reset CF+ cards.

Memory Map:

Offset	Description
0x0000	Attribute memory space
0x0800	I/O Space
0x1000	Common memory space
0x1800	Invalid
0x2XXX	Accesses to this memory range will generate a hardware reset the CF+ device

The table below lists all of the signals associated with the CF+ interface and includes notes that may be of interest to hardware developers.

Signal Name	Dir	Pin	CompactFlash V 1.3 Description	TRG Comments
A10-A0 (PC Card Memory Mode)	I	8, 10, 11, 12, 14, 15, 16, 17, 18, 19, 20	These address lines along with the –REG signal are used to select the following: The I/O port address registers within the CompactFlash Storage Card, a byte in the card's information structure and its configuration control and status registers.	Fully supported
A10-A0 (PC Card I/O Mode)	I		This signal is the same as the PC Card Memory Mode signal.	Fully supported
A2-A0 (True IDEMode)			In True IDE Mode only A[2..0] are used to select the one of eight registers in the Task File, the remaining address lines should be grounded by the host.	True IDE Mode not supported
BVD1 (PC Card Memory Mode)	I/O	46	This signal is asserted high as the BVD1 signal since a battery is not used with this product.	This is a no connect on TRGpro. CF Cards containing batteries cannot have their battery voltage monitored by TRGpro.
-STSCHG Status Changed (PC Card I/O Mode)			This signal is asserted low to alert the host to changes in the RDY/-BSY and Write Protect states, while the I/O interface is configured. Its use is controlled by the Card Config and Status Register	CompactFlash Cards cannot automatically inform TRGpro of changes to internal status.
-PDIAG (True IDE Mode)			In the True IDE Mode, this input / output is the Pass Diagnostic signal in the Master / Slave handshake protocol.	True IDE Mode not supported
BVD2 (PC Card Memory Mode)	I/O	45	This output line is always driven to a high state in Memory Mode since a battery is not required for this product.	This is a no connect on TRGpro. CF Cards containing batteries cannot have their battery voltage monitored by TRGpro.
-SPKR (PC Card I/O Mode)			This output line is always driven to a high state in I/O Mode since this product does not support the audio function.	TRGpro does not support driving its internal speaker from a CF card.
-DASP (True IDE Mode)			In the True IDE Mode, this input / output is the Disk Active/Slave Present signal in the Master/Slave handshake protocol.	True IDE Mode not supported

Signal Name	Dir	Pin	CompactFlash V 1.3 Description	TRG Comments
-CD1, -CD2 (PC Card Memory Mode)	O	26, 25	These Card Detect pins are connected to ground on the CompactFlash Storage Card. They are used by the host to determine that the CompactFlash Storage Card is fully inserted into its socket.	CD1- is connected to the CPU, but CD2- is not connected. The CF V1.3 specification *requires* that both pins function as described.
-CD1, -CD2 (PC Card I/O Mode)			This signal is the same for all modes.	CD1- is connected to the CPU, but CD2- is not connected. The CF V1.3 specification *requires* that both pins function as described.
-CD1, -CD2 (True IDE Mode)			This signal is the same for all modes.	True IDE Mode not supported
-CE1, -CE2 (PC Card Memory Mode)	I	7, 32	These input signals are used both to select the card and to indicate to the card whether a byte or a word operation is being performed. –CE2 always accesses the odd byte of the word. –CE1 accesses the even byte or the Odd byte of the word depending onA0 and –CE2. A multi-plexing scheme based on A0,-CE1, -CE2 allows 8 bit hosts to access all data on DO-D7. See Tables 4-11, 4-12, 4-15, 4-16 and 4-17.	Fully supported.
-CE1, -CE2 (PC Card I/O Mode)			This signal is the same as the PC Card Memory Mode signal.	Fully supported.
-CS0, -CS1 (True IDE Mode)			In the True IDE Mode CS0 is the chip select for the task file registers while CS2 is used to select the Alternate Status Register and the Device Control Register.	True IDE Mode not supported
-CSEL (PC Card Memory Mode)	I	39	This signal is not used for this mode.	No connect on TRGpro.
-CSEL (PC Card I/O Mode)			This signal is not used for this mode.	No connect on TRGpro
-CSEL (True IDE Mode)			This internally pulled up signal is used to configure this device as a Master or a Slave when configured in the True IDE Mode. When this pin is grounded, this device is configured as a Master. When the pin is open, this device is configured as a Slave	True IDE Mode not supported
D15 – D00 (PC Card Memory Mode)	I/O	31, 30, 29, 28, 27, 49, 48, 47, 6, 5, 4, 3, 2, 23, 22, 21	These lines carry the Data, Commands and Status information between the host and the controller. D00 is the LSB of the Even Byte of the Word. D08 is the LSB of the Odd Byte of the Word.	Fully supported
D15 – D00 (PC Card I/O Mode)			This signal is the same as the PC Card Memory Mode signal.	Fully supported
D15 – D00 (True IDE Mode)			In True IDE Mode, all Task File operations occur in byte mode on the low order bus D00-D07 while all data transfers are 16 bit using D00-D15.	True IDE Mode not supported
GND (PC Card Memory Mode)	--	1, 50	Ground.	Fully supported
GND (PC Card I/O Mode)			This signal is the same for all modes.	Fully supported
GND (True IDE Mode)			This signal is the same for all modes.	True IDE Mode not supported

-INPACK (PC Card Memory Mode)	O	43	This signal is not used in this mode.	No connect on TRGpro. No impact on performance.
-INPACK (PC Card I/O Mode)			The Input Acknowledge signal is asserted by the Compact Flash Storage Card when the card is selected and responding to an I/O read cycle at the address that is on the address bus. This signal is used by the host to control the enable of any input data buffers between the CompactFlash Storage Card and the CPU.	No connect on TRGpro. No impact on performance.
-INPACK (True IDE Mode)			In True IDE Mode this output signal is not used and should not be connected at the host.	True IDE Mode not supported
-IORD (PC Card Memory Mode)	I	34	This signal is not used in this mode.	Fully supported.
-IORD (PC Card I/O Mode)			This is an I/O Read strobe generated by the host. This signal gates I/O data onto the bus from the CompactFlash Storage Card when the card is configured to use the I/O interface.	Fully supported.
-IORD (True IDE Mode)			In True IDE Mode, this signal has the same function as in PC Card I/O Mode.	True IDE Mode not supported
-IOWR (PC Card Memory Mode)	I	35	This signal is not used in this mode.	Fully supported.
-IOWR (PC Card I/O Mode)			The I/O Write strobe pulse is used to clock I/O data on the Card Data bus into the CompactFlash Storage Card controller registers when the CompactFlash Storage Card is configured to use the I/O interface. The clocking will occur on the negative to positive edge of the signal (trailing edge).	Fully supported.
-IOWR (True IDE Mode)			In True IDE Mode, this signal has the same function as in PC Card I/O Mode.	True IDE Mode not supported
-OE (PC Card Memory Mode)	I	9	This is an Output Enable strove generated by the host interface. It is used to read data from the CompactFlash Storage Card in Memory Mode and to read the CIS and configuration registers.	Fully supported.
-OE (PC Card I/O Mode)			In PC Card I/O Mode, this signal is used to read the CIS and configuration registers.	Fully supported.
-OE (True IDE Mode)			To enable True IDE Mode this input should be grounded by the host.	True IDE Mode not supported

RDY/-BSY (PC Card Memory Mode)	O	37	In Memory Mode this signal is set high when the CompactFlash Storage Card is ready to accept a new data transfer operation and held low when the card is busy. The Host memory card socket must provide a pull-up resistor. At power up and Reset, the RDY/-BSY signal is held low (busy) until the CompactFlash Storage Card has completed its power up or reset function. No access of any type should be made to the CompactFlash Storage Card during this time. The RDY/-BSY signal is held high (disabled from being busy) whenever the following condition is true: The CompactFlash Storage Card has been powered up the +RESET continuously disconnected or asserted.	Not supported on TRGpro. This information can be read from a status register in the CF card.
-IREQ (PC Card I/O Mode)			I/O Operation – After the CompactFlash Storage Card has been configured for I/O operation, this signal is used as – Interrupt Request. This line is strobed low to generate a pulse mode interrupt or held low for a level mode interrupt.	Fully supported.
INTRQ (True IDE Mode)			I n True IDE Mode signal is the active high Interrupt Request to the host.	True IDE Mode not supported
-REG (PC Card Memory Mode)	I	44	This signal is used during Memory Cycles to distinguish between Common Memory and Register (Attribute) Memory accesses. High for Common Memory, Low for Attribute Memory.	Fully supported.
-REG (PC Card I/O Mode)			The signal must also be active (low) during I/O Cycles when the I/O address is on the Bus.	Fully supported.
-REG (True IDE Mode)			In True IDE M ode this input signal is not used and should be connected to VCC by the host.	True IDE Mode not supported
RESET (PC Card Memory Mode)	I	41	When the pin is high, this signal Resets the CompactFlash Storage Card. The CompactFlash Storage Card is Reset only at power up if this pin is left high or open from power-up. The CompactFlash Storage Card is also Reset when the Soft Reset bit in the Card Configuration Option Register is set.	Fully supported.
RESET (PC Card I/O Mode)			This signal is the same as the PC Card Memory Mode signal.	Fully supported.
-RESET (True IDE Mode)			In the True IDE Mode this input pin is the active low hardware reset from the host.	True IDE Mode not supported

VCC (PC Card Memory Mode)	--	13, 38	+5 V, +3.3 V power.	3.3 V on TRGpro, 5 V not supported
VCC (PC Card I/O Mode)			This signal is the same for all modes.	3.3 V on TRGpro, 5 V not supported
VCC (True IDE Mode)			This signal is the same for all modes.	True IDE Mode not supported
-VS1 -VS2 (PC Card Memory Mode)	O	33 40	Voltage Sense Signals. –VS1 is grounded so that the CompactFlash Storage Card CIS can be read at 3.3 volts and –VS2 is reserved by PCMCIA for a secondary voltage.	Not supported on TRGpro. These two lines are used to inform the host what the CF card voltage should be. TRGpro assumes 3.3 V.
-VS1 -VS2 (PC Card I/O Mode)			This signal is the same for all modes.	Not supported on TRGpro. These two lines are used to inform the host what the CF card voltage should be. TRGpro assumes 3.3 V.
-VS1 -VS2 (True IDE Mode)			This signal is the same for all modes.	True IDE Mode not supported
-WAIT (PC Card Memory Mode)	O	42	The –WAIT signal is driven low by the CompactFlash Storage Card to signal the host to delay completion of a memory or I/O cycle that is in progress.	Fully supported.
-WAIT (PC Card I/O Mode)			This signal is the same as the PC Card Memory Mode signal.	Fully supported.
-WAIT (True IDE Mode)			In True IDE Mode this output signal may be used as IORDY.	True IDE Mode not supported
-WE (PC Card Memory Mode)	I	36	This is a signal driven by the host and used for strobing memory write data to the registers of the CompactFlash Storage Card when the card is configured in the memory interface mode. It is also used for writing the configuration registers.	Fully supported.
-WE (PC Card I/O Mode)			In PC Card I/O Mode, this signal is used for writing the configuration registers.	Fully supported.
-WE (True IDE Mode)			In True IDE Mode this input signal is not used and should be connected to VCC by the host.	True IDE Mode not supported
WP (PC Card Memory Mode)	O	24	Memory Mode – The CompactFlash Storage Card does not have a write protect switch. This signal is held low after the completion of the reset initialization sequence.	This is not supported on TRGpro. The same information can be read in a CF card status register.
-IOIS16 (PC Card I/O Mode)			I/O Operation – When the CompactFlash Storage Card is configured for I/O Operation Pin 24 is used for the –I/O Selected is 16 Bit Port (-IOIS16) function. A Low signal indicates that a 16 bit or odd byte only operation can be performed at the addressed port.	Dynamic bus sizing is not supported on TRGpro, this pin is a no connect on TRGpro. If this ever turns out to be a problem, a unique software driver can cause the card to be read in 8 bit only mode, or 16 bit only mode.
-IOIS16 (True IDE Mode)			In True IDE Mode this output signal is asserted low when this device is expecting a word data transfer cycle.	True IDE Mode not supported

ENHANCED POWER SUPPLY

The TRGpro handheld computer operates on two standard AAA alkaline batteries. Power from the batteries is converted and used to power the computer as well as any CF cards that are placed in the expansion slot. The Compact Flash Association CF specification 1.4 defines two different power levels for cards. Power Level 0 limits current drawn to 75 mA. Power Level 1 allows for up to 500 mA to be drawn.

TRGpro fully supports Power Level 0 cards. The power supply can actually supply current to most cards that require more than the 75mA specified by Power Level 0. TRGPro can operate with Power Level 1 cards, however battery life when using these cards may be considered unacceptable by the user. IBM's Microdrive, for example, requires up to 400 mA during peak conditions, and it can be operated continuously in the TRGpro.

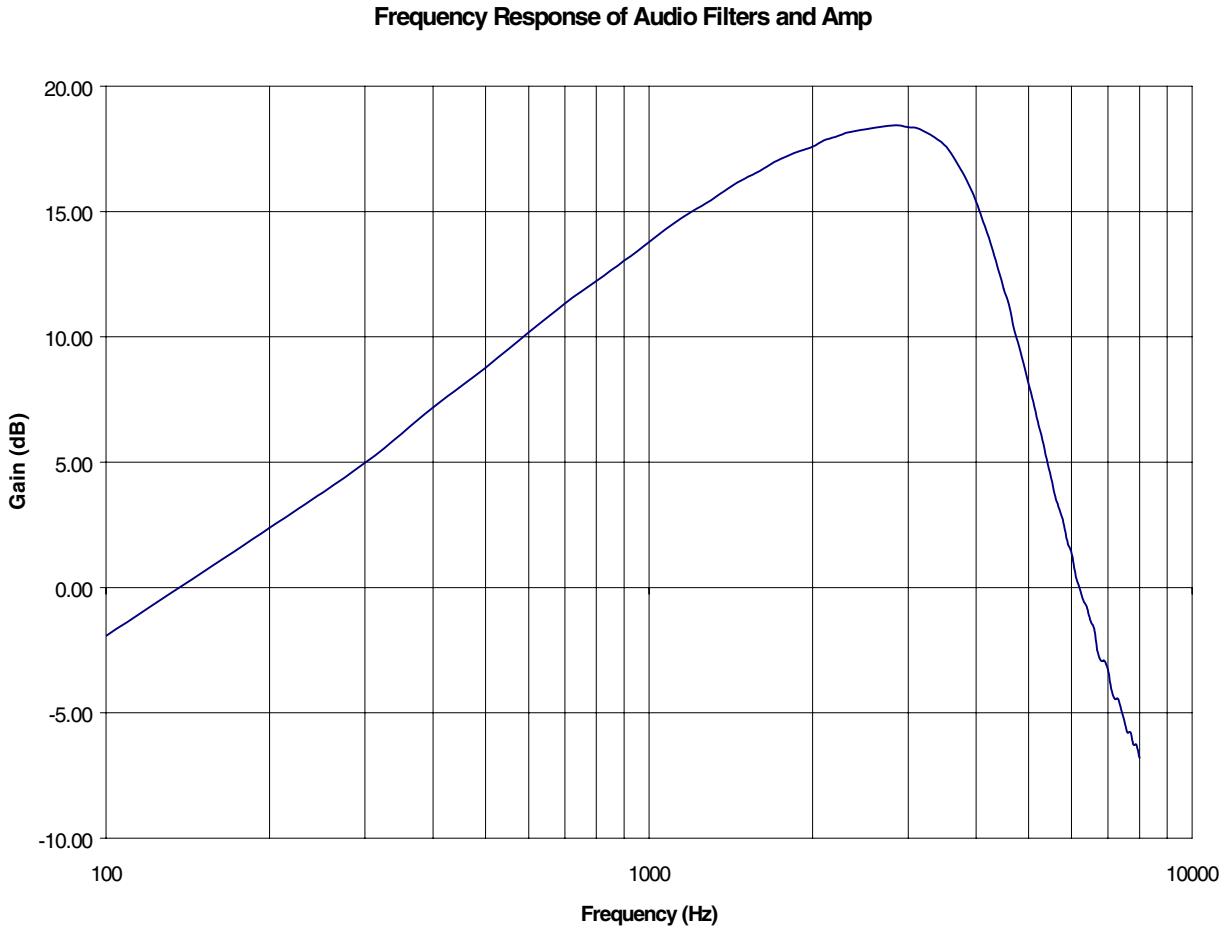
Typical current consumption for Flash storage cards is very low. As of this writing, all CF Flash storage cards draw less than 75 mA peak. Read operations typically require 32-50 mA depending upon capacity, and write operations require 32-70 mA. During sleep mode, cards typically draw 200 μA. If an application program is reading and writing 20% of the time, average currents will be in the range of 6.6 - 14.2 mA.

IMPROVED SOUND

The TRGpro computer contains enhanced sound capabilities. A louder, higher fidelity speaker has been added, and more sophisticated drive electronics allow for a new level of audio performance. For the first time, the Palm OS can play speech quality sound, and deliver significant volume for alarms.

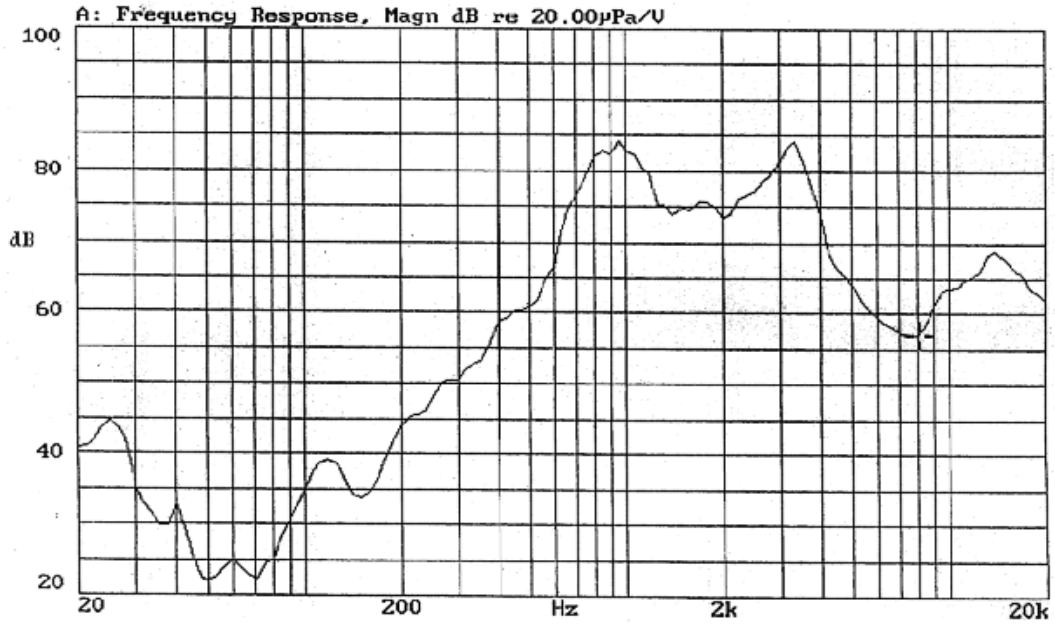
The TRGpro produces sound in a way that is compatible with the Palm OS. The CPU is used to create a PWM signal that can be modulated. The output from the CPU is then filtered by a low pass filter that removes the PWM carrier frequency and leaves only frequencies between 200 Hz and 4 kHz un-attenuated. After filtering, the audio output signal is acted upon by a variable gain audio amplifier. The gain setting of the amplifier can be controlled by using functions described in the Audio chapter found later in this document. The setting affects all sound that is produced by the unit in the same way. Thus, a user can set the maximum output level for all applications and OS functions with one operation. The gain of the amplifier can be smoothly varied by software functions.

Audio waveforms are created by the microprocessor through use of its internal PWM controller which can operate at rates up to 32KHz. A 5th order elliptical switched-capacitor low-pass filter removes the PWM carrier frequency. The resultant waveform is high-pass filtered and delivered to the input of the audio amplifier. The table below shows the overall frequency response of the filters and amplifier.



.

The 16 Ohm speaker is housed in the back case of the TRGpro and is connected to the audio amp through two leaf springs. The figure below shows the frequency response of the speaker itself.



ONBOARD FLASH MEMORY

TRGpro has two nearly-identical sites (U17 and U18) for population of on-board Flash memory. Both sites use the footprint for the JEDEC standard TSOP48 – type II package. The only difference between the two sites is that they are connected to two different chip selects from the CPU. U17 is connected to CS0- and is the Flash from which the CPU will fetch code immediately following a hard reset. U18 is connected to CS1-.

Flash chips ranging in size from 1MB to 8MB on each site are supported. Due to a slight change in pin-out between 8MB parts and all of the smaller parts, three address-line jumpers are required for each site. For parts less than 8MB, jumpers R24 and R55 should be installed while R30, R41, R51 and R54 should be removed. For 8MB parts the converse is true. 120nS Flash chips are sufficient to allow zero-wait state operation.

Valid Flash Configurations.

Total Onboard Flash	U17	U18
1MB	1MB	Empty
2MB	1MB	1MB
2MB	2MB	Empty
3MB	2MB	1MB
4MB	2MB	2MB
8MB	8MB	Empty
10MB	8MB	2MB
16MB	8MB	8MB

The address lines needed to access this larger amount of memory were unavailable in the PalmIIIx. The TRGpro recovers these needed address lines by moving some digital I/O functions off of the CPU to peripheral logic. PalmOS applications will not be effected by these changes since the PalmOS provides the necessary level of abstraction from the hardware.

The following lists manufacturer's part numbers of Flash devices approved for use in the TRGpro handheld computer.

Approved Flash Devices

Device Size	Manufacturer	Part number
1MB	AMD	29LV800BB-120EC
1MB	Toshiba	TC58FVB800FT-10
2MB	AMD	29LV160BB-120EC
2MB	Toshiba	TC58FVB160FT-10
2MB	AMD	29LV160DB-120EC
8MB	AMD	29LV641D(HLU)-120EI

COMPATIBILITY

Because every effort has been made to retain electrical and mechanical compatibility with the PalmIII line of handheld computers, TRGpro owners can continue to use many of the external peripherals and accessories that they already own. Some of these include:

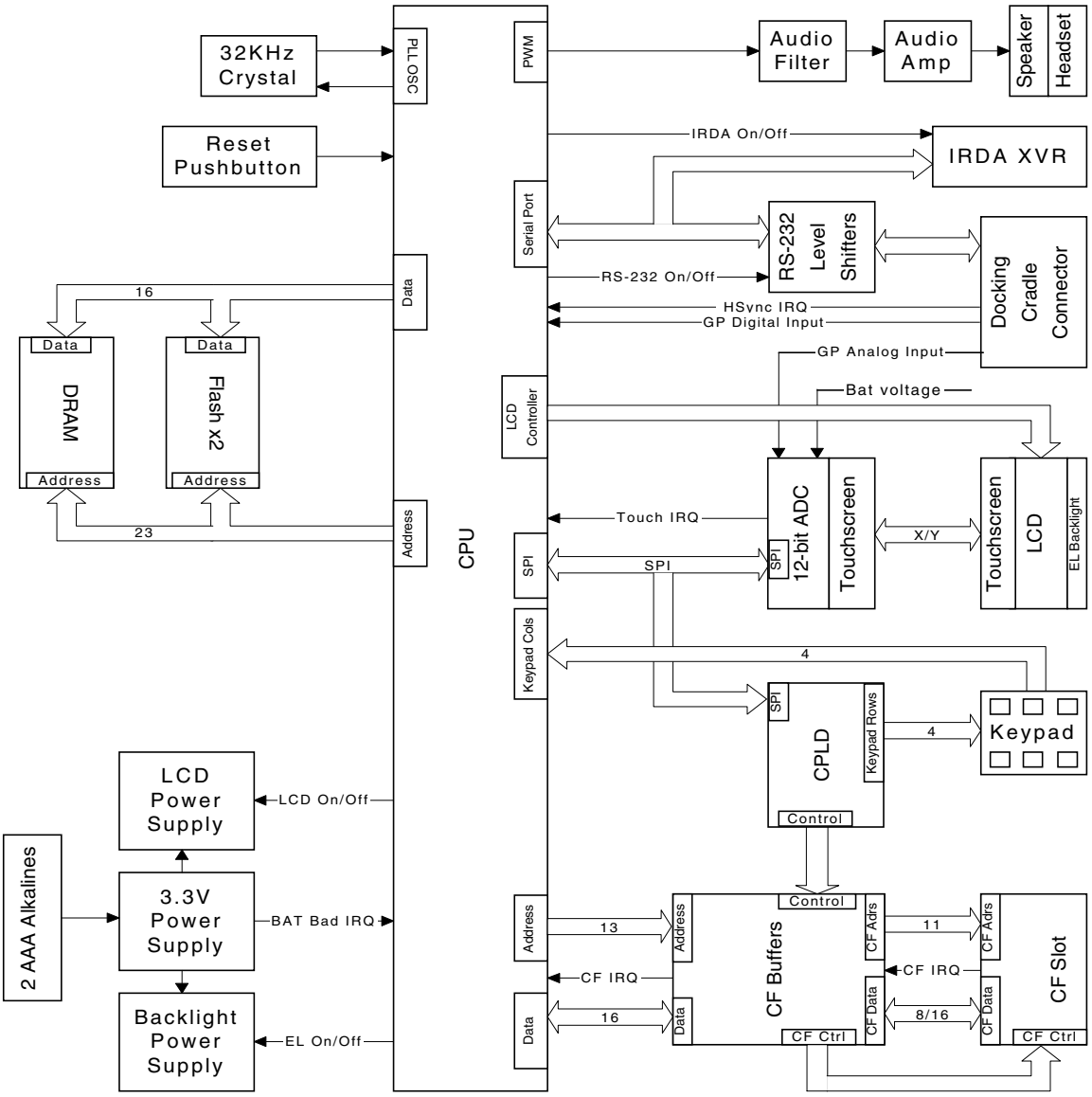
- Bottom port peripherals such as the GoType keyboard and PalmModem
- Leather belt clip and flip-top cover
- Styli designed for the Palm III product line

OTHER ELECTRICAL CHANGES

To make room for the CF+ card slot and to allow the IR window to remain along the top edge of the TRGpro handheld computer, it was necessary to move the IrDA module to the opposite side of the PCB. Due to height restrictions on that side of the board, it became necessary to change to an IrDA module with a much smaller footprint than the one used in the PalmIIIx. The new IrDA module is fully IrDA v1.3 (low-power option) compliant to 115Kbps.

To recover some address lines necessary to facilitate larger internal FLASH memory options and to provide the control logic needed to properly integrate the CF+ slot with the Dragonball CPU, a Complex Programmable Logic Device (CPLD) was added to the design.

TRGPRO BLOCK DIAGRAM



SECTION 2 – MECHANICAL INFORMATION

MECHANICAL CHANGES

The mechanical dimensions of the TRGpro have changed very little compared to those of the PalmIIIx. The overall height and width dimensions are identical. The thickness is also identical in all areas except for the upper part of the case that lies above the CF slot, which increased by 0.1 inches. The location and dimensions of the buttons, battery area, stylus area, the edges of the case and the bottom cradle port are all identical to that of the PalmIII line of devices. The mechanical changes have been kept to a minimum to ensure as much compatibility as possible with peripherals that were designed for the PalmIII line of devices.

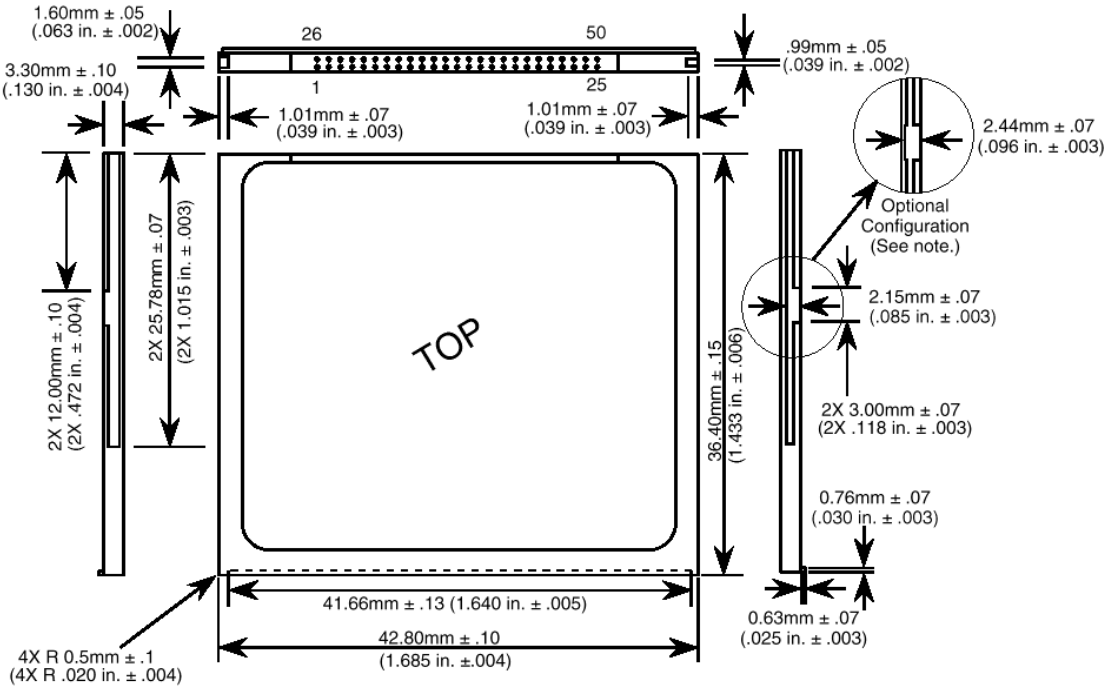
A detachable door has been added to the TRGpro to facilitate insertion and removal of CF+ devices. The door is made of IR transmissive material to allow the IrDA port, located at the top of the unit, to operate properly when the door is in place. Some CF+ I/O devices such as modems and Ethernet adapters require that a cable be connected to them during operation. When using these devices, the user can simply use the TRGpro with the CF door removed.

Description	**Physical Specifications**
Overall Dimensions	4.7 x 3.2 x 0.7 in (121x81x17mm)
Weight	5.9 oz. (167 g) including batteries
LCD Active Area	2.2 x 2.2 in (55 x 55 mm)
Touchscreen Active Area	3.1 x 2.3 in (79 x 59 mm)
Operating Temperature	32º–104º F (0º-40º C)
Storage Temperature	-22º to 158º F (-30º to 70º C)
Humidity	less than 90% (non-condensing)

Type I CF+ Card	**Physical Specifications**
Length	36.4 ± 0.15 mm (1.433 ±.006 in.)
Width	42.80 ± 0.10 mm (1.685 ±.004 in.)
Thickness Including Label Area	3.3 mm ± 0.10 mm (.130 ± .004 in.)

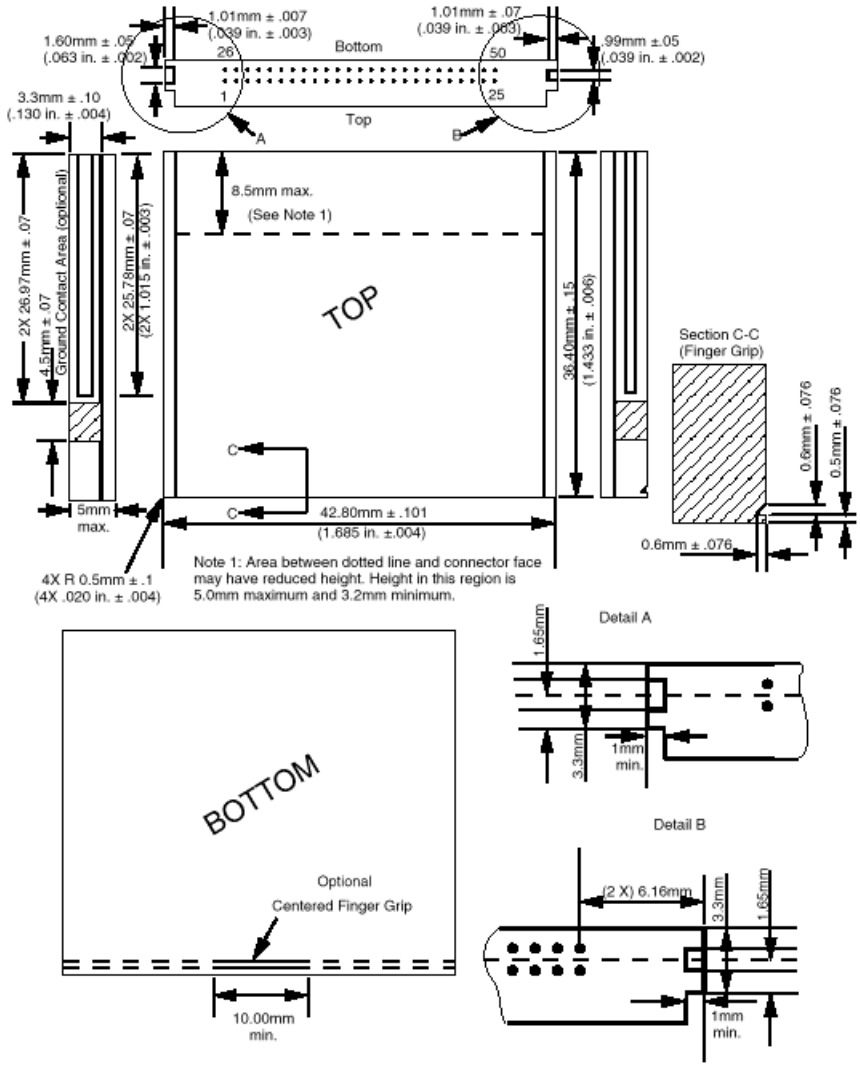
Type II CF+ Card	**Physical Specifications**
Length	36.4 ± 0.15 mm (1.433 ±.006 in.)
Width	42.80 ± 0.10 mm (1.685 ±.004 in.)
Thickness Including Label Area	5.0 mm maximum (.1968 in. maximum)

TYPE I COMPACTFLASH STORAGE CARD AND CF+ CARD DIMENSIONS



Note: The optional notched configuration was shown in the CF Specification Rev. 1.0. In specification Rev. 1.2, the notch was removed for ease of tooling. This optional configuration can be used but it is not recommended.

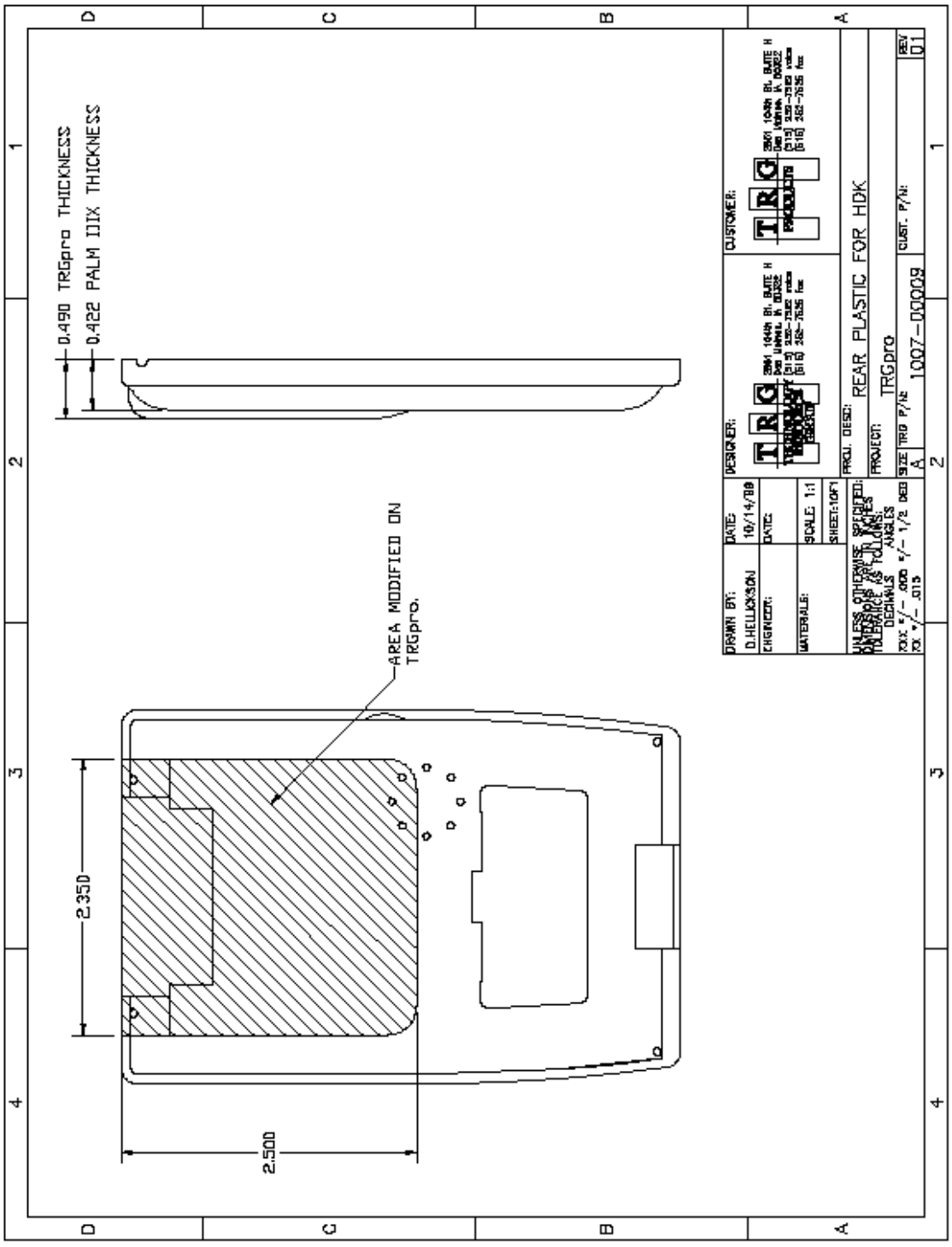
Type II CompactFlash Storage Card and CF+ Card Dimensions



Note: the recessed centered finger grip (Section C-C) is optional although it is recommended for CF+ Type II cards. Additionally, it is recommended that Type II host slots include an ejector mechanism.

Hardware developers that want to create custom hardware in CF+ type I or type II card formats can purchase CF card connector and frame kits from various manufacturers. These kits typically contain all of the mechanical components necessary to produce a CF+ card such as the 50 position socket, plastic frame that supports the PCB and any covers that are necessary to seal the card.

Rear Case Modification Area Dimensions



CHAPTER THREE - CF LIBRARY

This chapter is intended to introduce the use of, and provide a reference to, the CF Library procedures. It is directed toward Palm OS application developers who wish to access CF cards from within their applications. It is assumed that the reader is familiar with the C programming language, in particular within the context of the Palm OS. Section 1 of this document describes the use of shared libraries in Palm OS applications, summarizing the functionality provided by the Cf Library. Section 2 details each of the library calls, describing their function, parameters, and return value. Section 3 lists the possible error return codes, and Section 4 discusses a sample project.

SECTION 1 – CF LIBRARY OVERVIEW

THE PURPOSE OF THE COMPACT FLASH LIBRARY

The CF library provides an interface to the TRGpro CF slot hardware. It includes low-level services such as turning the slot on and off, setting the bus width, and initializing the IRQ line. The primary purpose of the CF library is to hide the TRGpro CF-interface hardware details from applications/drivers wishing to use CF cards. The CF library does not provide CF card-specific functions, such as parsing CIS tuples or setting configuration registers.

LOADING, UNLOADING, AND ACCESSING THE CF SHARED LIBRARY

Currently, the Cf Library is implemented as a Palm OS shared library. To access the Cf calls, an application must search for the library, load the library if not found, then open the library. Opening the library returns a library reference number that is used to access the individual functions within the library. When the application is finished with the library, it should close the library. The current implementation of the CF library supports only single-user access; applications cannot share an open library. If one application leaves the CF library open upon exit and another application finds the library and begins using library calls, the results are unpredictable.

The calling application must include the header file cflib.h. This file contains the required constant definitions, structure typedefs, and function prototypes for the library. In addition, this file maps library functions calls to the corresponding system trap instructions, through which all library routines are accessed. If the caller requires notification of CF card insertion/removal events, it must also include notify.h and the PalmOS header NotifyMgr.h.

To find a loaded library, an application calls **SysLibFind**, specifying the library. If not found, an application loads the library using **SysLibLoad**, specifying the library type and creator IDs. For the Cf library, the name, type and creator IDs are defined in cflib.h as *CfLibName*, *CfLibTypeID* and *CfLibCreatorID*, respectively. After loading the library, it must be opened with a call to

CfLibOpen. Opening the library allocates and initializes its global variables, and sets up the CF socket hardware.

Once the library is open, the application may make calls to the library functions. The first parameter to a library call is always the library reference number returned when the library is loaded. Most library calls return an integer result of 0 on success and a non-zero error code on failure.

The application that opens the library is responsible for closing and unloading the library. The library is closed with the **CfLibClose** call, and unloaded with the **SysLibRemove** call. It is possible for an application to leave the library loaded when exiting. The library may then be accessed by other applications through the **SysLibFind** command, which returns a reference to an already-loaded library. Once the reference number is obtained, the library is opened as usual with **CfLibOpen** call. In either case, however, the caller must open the library on startup and close it on exit. The library cannot be left open between applications.

Currently, the name of the Cf library used for **SysLibFind** is “Cf.lib,” the creator ID is “CfL ”, and the type ID is “libr”. These constants are all defined in cflib.h.

SUMMARY OF CF LIBRARY FUNCTIONS

The CF library calls may be grouped into two categories: hardware management and library management. The calls, grouped by category, are listed below, with brief descriptions of each call’s function. An alphabetical listing with a detailed specification of each call is given in section 2.

Hardware management

• CfBaseAddress	Return the base address of the card’s attribute memory.
• CfCardInserted	Check whether a card is inserted in the slot.
• CfDisableIRQLine	Disable the IRQ line associated with the slot.
• CfEnableIRQLine	Enable the IRQ line associated with the slot.
• CfGetDataWidth	Return the current data bus width.
• CfGetIRQLevel	Return the IRQ line associated with the slot.
• CfPowerSlot	Apply/remove power from the slot.
• CfReset	Reset the card.
• CfSetSwapping	Enable/disable CF data bus byte swapping.
• CfSetDataWidth	Set the data bus width.
• CfSlotPowered	Check whether the slot is currently powered.

Library management

• CfLibClose	Close the CF library.
• CfGetLibAPIVersion	Get the library version number.
• CfLibOpen	Open the CF library.

CARD INSERTION/REMOVAL NOTIFY

To notify the caller of card insertion/removal events, a launch code can be sent by the system to the application. Applications “register” at startup for notification of CF insertion/removal events, and are then sent the launch code sysAppLaunchCmdNotify when these events occur. The *cmdPBP* points to a *SysNotifyParamType* object, containing a field *notifyParamP* which in turn points to a UInt32 containing one of the following event types (defined in notify.h):

- CFEventCardInserted — a CF card has been inserted.
- CFEventCardRemoved — a CF card has been removed.
- CFEventPowerIsBackOn — the device just powered up, the card may have changed while the device was off.

An application registers for notification by calling **SysNotifyRegister**, defined in system header file NotifyMgr.h. The parameters, briefly, are as follows:

- CardNo — card number of calling application.
- DbID — database ID of calling application.
- NotifyType — must be sysNotifyCFEvent, defined in notify.h.
- CallbackP — not used, should be NULL.
- Priority — not used, should be 0.
- UserDataP — not used, should be NULL.

When an application exits, it should unregister by calling **SysNotifyUnregister**, defined in system header file NotifyMgr.h. The parameters, briefly, are as follows:

- CardNo — card number of registered application.
- DbID — database ID of registered application.
- notifyType — must be sysNotifyCFEvent, defined in notify.h.
- priority — not used, should be 0.

Note that an application may, in theory, remain registered after exiting. However, any system event that causes the application’s database ID to change (such as hot-syncing a new copy of the application) may cause a system crash during the next CF event.

The notification manager requires PalmOS 3.3 header files or above.

SECTION 2 – CF LIBRARY FUNCTION REFERENCE

This section contains an alphabetical listing of the functions available in the Cf Library, along with a brief description of each.

All library functions require an open library reference of type *UInt16* as their first parameter. This reference is used by the Palm OS to locate the jump table for the library, and is returned by the **SysLibLoad** call.

DATA TYPES

This section details data types used by multiple functions in the Cf library. Note that the library uses the following data types, which may not be defined in the pre-OS 3.3 headers:

- Int8 formerly defined as *char*
- UInt8 formerly defined as *Byte*
- Int16 formerly defined as *Int*
- UInt16 formerly defined as *Word*
- Int32 formerly defined as *long*
- UInt32 formerly defined as *DWord*

FUNCTIONS

CFBASEADDRESS

Purpose Get the base address of the card's attribute, common, and io memory spaces.

Prototype

```
Err CfBaseAddress(UInt16  libRef,
                  UInt8   slot_num,
                  Uint32 *attribute,
                  Uint32 *common,
                  UInt32 *io);
```

Parameters

`-> libRef`	Library reference number
`-> slot_num`	Currently always 0
`-> attributes`	Returns the base of attribute memory. Set NULL if return value not used.
`-> common`	Returns the base of common memory. Set NULL if return value not used.
`-> io`	Returns the base of io space. Set NULL if return value not used.

Result 0 on success or error code on failure

Comments These addresses are based on the hardware interface to the slot, not on the specific CF card currently inserted; a given card may not support all three memory spaces.

CfCardInserted

Purpose Determine whether a card is currently inserted in the slot. The card need not be powered.

Prototype

```
Err CfCardIsInserted(UInt16   libRef,
                     UInt8    slot_num,
                     Boolean *inserted);
```

Parameters

`-> libRef`	Library reference number
`-> slot_num`	Currently always 0
`-> inserted`	Set to **true** is card is currently inserted in slot, **false** otherwise.

Result 0 on success or error code on failure

Comments

CFDISABLEIRQLINE

Purpose Disables the IRQ line associated with the CF slot (currently IRQ3).

Prototype

```
Err CfDisableIRQLine(UInt16 libRef,
                     UInt8  slot_num);
```

Parameters

`-> libRef` Library reference number

`-> slot_num` Currently always 0

Result 0 on success or error code on failure

Comments This routine merely disables the physical hardware line between the processor and the slot (it sets the IRQ3 line on the EZ processor to be a general purpose input). It does not mask interrupts at the EZ interrupt controller or on the card itself.

CfEnableIRQLine

Purpose Enables the IRQ line associated with the CF slot (currently IRQ3).

Prototype

```
Err CfEnableIRQLine(UInt16 libRef,
                    UInt8  slot_num);
```

Parameters

`-> libRef` Library reference number

`-> slot_num` Currently always 0

Result 0 on success or error code on failure

Comments This routine merely activates the physical hardware line between the processor and the slot (it sets the IRQ3 line on the EZ processor to its dedicated interrupt function). It does not setup interrupts on the card itself, nor setup the IRQ3 bit in the EZ interrupt controller. These operations must be performed by the application program.

CfGetDataWidth

Purpose Returns the current data bus width of the slot. An error code is returned if the slot is not powered.

Prototype

```
Err CfGetDataWidth(UInt16 libRef,
                   UInt8 slot_num,
                   data_width_type *width);
```

Parameters

`-> libRef`	Library reference number
`-> slot_num`	Currently always 0
`<- width`	Current bus width of the slot (CF_8_BIT or CF_16_BIT).

Result 0 on success or error code on failure

Comments

CfGetIRQLevel

Purpose Returns the number of the IRQ line connected to the slot

Prototype

```
Err CfGetIRQLevel(UInt16 libRef,
                  UInt8  slot_num,
                  Uint8 *level);
```

Parameters

`-> libRef`	Library reference number
`-> slot_num`	Currently always 0
`<- level`	Set to the IRQ line used by the slot (currently IRQ3).

Result 0 on success or error code on failure

Comments This is currently IRQ3, and this routine will always return a 3. Future versions of the hardware may change the IRQ line, however. Programs using card interrupts should use this call as a sanity check before enabling interrupts, to gracefully handle future hardware changes.

CFGETLIBAPIVERSION

Purpose Get the CF library version

Prototype

```
Err CfGetLibAPIVersion(UInt16 libRef,
                       UInt32 *dwVerP);
```

Parameters

`-> libRef`	Library reference number
`-> dwVerP`	Pointer to UInt32 variable for library version. Version format follows the Palm OS versioning scheme.

Result 0 on success or error code on failure

Comments Other than CfLibOpen, this is the only library call that can be made without first opening the library. The format of the version UInt32 is documented in the Palm OS header files.

CfGetSleepPowerState

Purpose Returns the current value set by the function CfSetSleepPowerState. When the unit is about to enter sleep mode, this value determines whether or not the card slot remains powered or not.

Prototype

```
CfGetSleepPowerState(UInt16       libRef,
                     UInt8        slot_num,
                     cf_pwr_type *sleep_state);
```

Parameters

`-> libRef`	Library reference number
`-> slot_num`	Currently always 0
`<- sleep_state`	Returns either CF_POWER_ON or CF_POWER_OFF

Result 0 on success or error code on failure

Comments

CFLIBCLOSE

Purpose Close the CF library, freeing allocated resources.

Prototype
```
Err CfLibClose(UInt16 libRef);
```

Parameters
```
-> libRef
```
Library reference number

Result 0 on success, CF_ERR_LIB_IN_USE if another application has the library open (currently not supported).

Comments Currently, the library must be closed when the calling application exits.

CFLIBOPEN

Purpose Open the CF library

Prototype
```
Err CfLibOpen(UInt16 libRef);
```

Parameters
```
-> libRef
```
Library reference number

Result 0 on success, CF_ERR_NO_MEMORY if not enough RAM available for allocating global variables.

Comments The library must have been previously loaded with **SysLibLoad**. This function allocates the library global data block and sets up the hardware for the CF slot.

CFPOWERSLOT

Purpose Apply power or remove power from the CF slot.

Prototype

```
Err CfPowerSlot(UInt16          libRef,
                UInt8           slot_num,
                cf_pwr_type     on_off,
                data_width_type width);
```

Parameters

`-> libRef`	Library reference number
`-> slot_num`	Currently always 0
`-> on_off`	Indicates whether to turn the slot on or off (see below).
`-> width`	Current bus width of the slot (CF_8_BIT or CF_16_BIT).

Result 0 on success or error code on failure

Comments Parameter *on_off* is set to CF_POWER_ON to turn slot on, CF_POWER_OFF to turn slot off. Parameter *width* is set to CF_8_BIT to power slot in 8-bit mode, and CF_16_BIT to power slot in 16 bit mode (this parameter is ignored when removing power from the slot). The bus width may be changed later without powering down the card (see *CfSetDataWidth* for details). **Note:** When powering the card in 8-bit mode, data bus byte swapping is automatically set to the appropriate value for an 8-bit bus, and must not be changed.

CFRESET

Purpose Perform a hardware reset on the CF card. This pulses the card's physical reset pin.

Prototype

```
Err CfReset(UInt16 libRef,
            UInt8 slot_num);
```

Parameters

`-> libRef`	Library reference number
`-> slot_num`	Currently always 0

Result 0 on success or error code on failure

Comments

CFSETDATAWDITH

Purpose Sets the data bus width to the CF slot.

Prototype

```
Err CfSetDataWidth(Uint16 libRef,
                   UInt8 slot_num,
                   data_width_type *width);
```

Parameters

`-> libRef`	Library reference number
`-> slot_num`	Currently always 0
`-> width`	Data bus width to set, either CF_8_bit or CF_16_bit

Result 0 on success or error code on failure

Comments Sets the data bus width of the CF slot. Sets the width of the slot buffers and the slot chip select only. It does not set the data width of the CF card itself. The slot must be powered before the data width can be set. The application must set the proper data path width manually before a card access because the bus control signals generated by the EZ processor do not contain data width information. An application may switch the data width as needed to support a mix of 8 and 16 bit transfers, without cycling power or resetting the card. **Note**: when operating in 8-bit mode, data bus byte swapping must be enabled to read valid data. It is the responsibility of the calling application to set the byte swapping appropriately when manually changing the data path width

CfSetSleepPowerState

Purpose Sets a value which determines which action the unit will take when it enters sleep mode. If the value is set to CF_POWER_ON, the unit will not power down the CF slot when it enters sleep mode. If the value is set to CF_POWER_OFF, it will power down the slot.

Prototype

```
CfSetSleepPowerState(UInt16      libRef,
                     UInt8       slot_num,
                     cf_pwr_type sleep_state);
```

Parameters

`-> libRef`	Library reference number
`-> slot_num`	Currently always 0
`-> sleep_state`	Either CF_POWER_ON or CF_POWER_OFF

Result 0 on success or error code on failure

Comments

CFSETSWAPPING

Purpose Enable/disable byte swapping on the 16-bit CF data bus.

Prototype

```
Err CfSetSwapping(UInt16  libRef,
                  UInt8   slot_num,
                  Boolean swap_bytes);
```

Parameters

`-> libRef`	Library reference number
`-> slot_num`	Currently always 0
`-> swap_bytes`	Set to **true** to swap the byes on the 16-bit CF card, false to pass straight through.

Result 0 on success or error code on failure

Comments When disabled, 16-bit data from the CF card is passed directly across to the 16-bit processor bus. When enabled, high and low bytes from the CF card are swapped before being placed on the processor bus. In 16-bit mode, byte swapping is useful when reading cards containing 16-bit data stored in Intel format, such as the FAT data stored on ATA cards. In 8-bit mode, swapping **must** be enabled to read valid data. When powering the card in 8-bit mode, the bus swapping is set automatically by the library.

CFSLOTPOWERED

Purpose Queries the hardware to determine if the CF slot is currently powered

Prototype

```
Err CfSlotPowered(UInt16          libRef,
                  UInt8           slot_num,
                  Boolean         *powered,
                  data_width_type *data_width,
                  Boolean         *card_in_use);
```

Parameters

`-> libRef`	Library reference number
`-> slot_num`	Currently always 0
`<- powered`	Set to **true** if power is currently applied to the slot, **false** otherwise
`<- data_width`	If the slot is powered, returns the data width (CF_8_BIT or CF_16_BIT). Set to NULL if the value is not needed
`<- card_in_use`	Set **true** if the slot has been powered by an application other than the CF library. Set to NULL if the value is not needed

Result 0 on success or error code on failure

Comments If the slot is powered, checks the internal CF library card status variables to determine whether the card was powered by an outside application.

SECTION 3 – CF LIBRARY ERROR CODES

The following is a list of the error return codes from the CF library calls. These are defined in CfLib.h.

CF_ERR_BAD_PARAM	A parameter is invalid.
CF_ERR_LIB_NOT_OPEN	Trying to close an library that is not open.
CF_ERR_LIB_IN_USE	Trying to close a library opened by another application.
CF_ERR_NO_MEMORY	Not enough memory for the requested call.
CF_ERR_CARD_DISABLED	CF card has not been powered.

SECTION 4 – CF LIBRARY EXAMPLE CODE

The Cf SDK zip file contains a demonstration application that uses the Cf Library. The demonstration application is a CodeWarrior project with all source code in the *src* subdirectory. The Cf library header file, cflib.h, and the notification manager header file notify.h, are also in the *src* subdirectory.

CHAPTER FOUR –
FAT FILE SYSTEM LIBRARY

This chapter is intended to introduce the use of, and provide a reference to, the Ffs (FAT File System) Library procedures. It is directed toward Palm OS application developers who wish to access CF cards from within their applications. It is assumed that the reader is familiar with the C programming language, in particular within the context of the Palm OS.

Section 1 of this document gives background detail on the FAT file system. Section 2 describes the use of shared libraries in Palm OS applications, summarizing the functionality provided by the Ffs Library. Section 3 details the shared data structures used by multiple functions in the Ffs library and describes each of the library calls, describing their function, parameters, and return value. Section 4 lists the possible error codes and their interpretation. , and Section 4 discusses a sample project.

SECTION 1 - FAT FILE SYSTEM OVERVIEW

The FAT file system in this document refers to a system of file management on a storage device. The device is divided into clusters, each of which can be composed of one or more sectors. A cluster can be in one of three states:

- Allocated to a file
- Allocated to a directory
- Unused or free

The mapping of the clusters is contained in a File Allocation Table (FAT), which is where the file system gets its name.

The use of this SDK shields developers from learning the complexities of how the FAT file system is implemented. However, if you require additional information, please refer to one of the following reference works:

- Inside the Windows 95 File System, by Stan Mitchell. 1997, O'Reilly & Associates, Inc.
- Dissecting DOS, by Michael Podanoffsky, 1995, Addison-Wesley.

TRGPRO AND THE FAT FILE SYSTEM

The TRGpro is a computing device built upon industry standards. It was designed with a slot to accept CompactFlash devices, which are rapidly becoming the standard for handheld computers. In keeping with this eye towards standards, its internal implementation for accessing CompactFlash memory cards is based upon a FAT file system.

The true advantage to using the FAT file system is that it is a standard also supported by PC's running any of the following operating systems:

- MS-DOS (all versions)
- Windows 3.1
- Windows 95
- Windows 98
- Windows NT (all versions)
- Windows 2000

For the TRGpro, the removable media is a CompactFlash memory card.

SECTION 2 - FFS LIBRARY OVERVIEW

THE PURPOSE OF THE FAT FILE SYSTEM LIBRARY

The FAT File System (Ffs) shared library provides an interface to Compact Flash (CF) cards containing a FAT File System. The interface is based upon the unbuffered file/disk system and library calls typically used with the C language. Support is provided for manipulating both files and directories, simplifying the exchange of data between the Palm device and a PC. In addition, the high-capacities of existing CF cards allow Ffs-aware applications to create, read, and modify files much larger than the total storage space available on existing Palm devices. A document reader, for example, could access documents directly on a CF card, without first having to move the documents in the Palm device RAM or Flash.

LOADING, UNLOADING, AND ACCESSING THE FFS SHARED LIBRARY

Currently, the Ffs Library is implemented as a Palm OS shared library. To access the Ffs calls, an application must search for the library, load the library if not found, then open the library. Opening the library returns a library reference number that is used to access the individual functions within the library. When the application is finished with the library, it should close the library. The current version of the library does not support the sharing of open files between applications, and only one application should have the library open at any one time (though the system may have it open, also).

The calling application must include the header file ffslib.h. This file contains the required constant definitions, structure typedefs, and function prototypes for the library. In addition, this file maps library functions calls to the corresponding system trap instructions, through which all library routines are accessed. If the caller requires notification of CF card insertion/removal events, it must also include notify.h and the PalmOS header NotifyMgr.h (requires OS 3.3 headers).

To find a loaded library, an application calls **SysLibFind**, specifying the library. If not found, an application loads the library using **SysLibLoad**, specifying the library type and creator IDs. For the Ffs library, the name, type and creator IDs are defined in ffslib.h as *FfsLibName*, *FfsLibTypeID* and *FfsLibCreatorID*, respectively. After loading the library, it must be opened with a call to **FfsLibOpen**. Opening the library allocates and initializes its global variables, and sets up the CF socket hardware.

Once the library is open, the application may make calls to the library functions. The first parameter to a library call is always the library reference number returned when the library is loaded. Most library calls return an integer result of 0 on success and –1 on failure. A more specific error code may be obtained through another library call.

The application that opens the library is responsible for closing and optionally unloading the library. The library is closed with the **FfsLibClose** call, and unloaded with the **SysLibRemove** call. The library can only be removed, however, if it is not in use by the system, as indicated by the value 0 returned from **FfsLibClose**. If still in use, **FfsLibClose** returns FFS_ERR_LIB_IN_USE. It is possible for an application to leave the library loaded when exiting.

The library may then be accessed by other applications through the **SysLibFind** call, which returns a reference to an already-loaded library. Once the reference number is obtained, the library is opened as usual with **FfsLibOpen** call. In either case, however, the caller must open the library on startup and close it on exit. The library should not be left open between applications.

Currently, the name of the Ffs library used for **SysLibFind** is “Fsf.lib,” the creator ID is “FfsL,” and the type ID is “libr.” These constants are all defined in ffslib.h.

SUMMARY OF FFS LIBRARY FUNCTIONS

The Ffs library calls may be grouped into six categories: disk management, directory management, file access, file management, library management, and error handling. The calls, grouped by category, are listed below, with brief descriptions of each call’s function. An alphabetical listing with a detailed specification of each call is given in section 3.

Disk management

- FfsCardIsATA check if inserted card is an ATA device.
- FfsCardInserted check if a CF card is inserted.
- FfsFlushdisk flush all buffers to flash.
- FfsFormat format the card.
- FfsGetdiskfree get the total size of the CF disk, and the amount of free space.
- FfsGetdrive get the current working drive number.
- FfsSetdrive set the current working drive number.

.

Directory management

- FfsChdir change the current working directory.
- FfsFinddone free resources after a directory search.
- FfsFindfirst start a directory search.
- FfsFindnext continue a directory search.
- FfsGetcwd get the current working directory.
- FfsIsDir check if the specified path is a directory or file.
- FfsMkdir create a directory.
- FfsRename rename a directory.
- FfsRmdir remove a directory.

File access

- FfsClose close a file.
- FfsCreat create a new file.
- FfsEof check if the current file pointer is at the end of the file.
- FfsLseek move a file pointer.
- FfsOpen open/create a file.
- FfsTell get the current file pointer value.
- FfsWrite write to a file.

File management

- FfsFlush — flush an open file to disk.
- FfsFstat — get information about an open file.
- FfsGetfileattr — get file attributes.
- FfsRemove — delete a file.
- FfsRename — rename a file.
- FfsSetfileattr — set file attributes.
- FfsStat — get information about a file.
- FfsUnlink — delete a file (same as FfsRemove).

Library management

- FfsGetLibAPIVersion — get the Ffs library version number.
- FfsLibClose — close the library.
- FfsLibOpen — open the library.

Error handling

- FfsGetErrno — get the current global error result code.
- FfsInstallErrorHandler — install a critical error handler callback function.
- FfsUnInstallErrorHandler — remove the critical error handler callback function.

For the most part, these functions implement the low-level unbuffered I/O functions found in the C language. The buffered stream I/O functions, such as *fopen* and *fprintf*, are not supported, though they could be built on top of the Ffs library layer.

Although many Ffs library calls accept a drive letter as part of the path string, and routines are provided to get and set the default drive, the Ffs library and Nomad hardware currently support only a single drive. This drive is signified as number 0 or 1 (0 indicates current drive, 1 indicates the first drive), and path “A:”.

THE GLOBAL ERROR RESULT CODE

Most of the Ffs library calls return an integer error indicator, set to 0 for success and –1 for failure. Library calls that return some other type of value, such as a pointer or file offset, always reserve one value to indicate an error. In either case, a specific error code is loaded into the global *errno* variable. The *errno* variable is not cleared on a successful call, so at any given time it contains the last error code generated. The current *errno* value may be retrieved by calling **FfsGetErrno**.

CRITICAL ERROR HANDLER CALLBACK

If an I/O error occurs when accessing the CF card, a critical error handler is called. The critical error handler is responsible for deciding whether to abort or retry the current operation, to mark a failed sector as bad, or to reformat the card. The actual choices available in a specific situation

are dependent on the type of critical error that occurred, and are determined by the internal critical error handler.

Regardless of the type of critical error that occurred, “abort current operation” is always a choice, and is the default action taken by the critical error handler. The calling program may supply its own critical error handler, however, to prompt the user for the desired course of action. A custom critical error handler is installed by a call to **FfsInstallErrorHandler**. The custom error handler takes as parameters a drive number, a code indicating the valid responses, and a string containing the specific error message, and returns the desired course of action. In current versions of the library, the drive number will always be 0. The codes defining the valid responses are listed below, along with corresponding course of action codes:

- **CRERR_NOTIFY_ABORT_FORMAT**: CRERR_RESP_ABORT or CRERR_RESP_FORMAT.
- **CRERR_NOTIFY_CLEAR_ABORT_RETRY**: CRERR_RESP_ABORT, CRERR_RESP_RETRY, or CRERR_RESP_CLEAR.
- **CRERR_NOTIFY_ABORT_RETRY**: CRERR_RESP_ABORT or CRERR_RESP_RETRY.

The course of action codes are interpreted by the internal critical error handler as follows:

- CRERR_RESP_ABORT: Abort the current operation.
- CRERR_RESP_RETRY: Retry the current operation.
- CRERR_RESP_FORMAT: Attempt to format the card.
- CRERR_RESP_CLEAR: Clear corrupt sector and retry the current operation.

These codes are all defined in file ffslib.h.

The custom critical error handler will typically display an alert box containing the error message text passed in from the internal critical error handler and prompting the user with the choices appropriate for the error type. For example, if the CF card is removed during an operation, the custom error handler will be called with a response code of CRERR_NOTIFY_ABORT_RETRY and an error message of “Bad card”. The error handler would then display an alert with buttons for “Abort” and “Retry”. Note that the “Abort” button should return the default value 0, in case the user presses an application launch button when the alert is displayed (in this case, the system will force the default return value from all alerts until the running application terminates).

CARD INSERTION/REMOVAL NOTIFY

When a CF card is removed while the Ffs Library is loaded, the library automatically clears all data structures associated with the card and reinitializes in preparation for the next card. Thus, if a card is removed during a library call, the library will not be able to complete the call even if the card is reinserted. This is an unfortunate side effect of the fact that most CF storage cards are missing the unique serial number in the drive ID information that is used to identify the cards. Because of this omission, the library is unable to determine if a reinserted card is identical to the previously loaded card.

The library reinitialization is invisible to the calling application. In order to notify the caller of insertion/removal events, a launch code can be sent by the system to the application. Applications “register” at startup for notification of CF insertion/removal events, and are then sent the launch code *sysAppLaunchCmdNotify* when these events occur. The *cmdPBP* points to

a *SysNotifyParamType* object, containing a field *notifyParamP* which in turn points to a UInt32 containing one of the following event types (defined in notify.h):

- CFEventCardInserted: a CF card has been inserted.
- CFEventCardRemoved: a CF card has been removed.
- CFEventPowerIsBackOn: the device just powered up, the card may have changed while the device was off.

An application registers for notification by calling **SysNotifyRegister**, defined in system header file NotifyMgr.h. The parameters, briefly, are as follows:

- cardNo: card number of calling application.
- dbID: database ID of calling application.
- notifyType: must be *sysNotifyCFEvent*, defined in notify.h.
- callbackP: not used, should be NULL.
- priority: not used, should be 0.
- userDataP: not used, should be NULL.

When an application exits, it should unregister by calling **SysNotifyUnregister**, defined in system header file NotifyMgr.h. The parameters, briefly, are as follows:

- cardNo: card number of registered application.
- dbID: database ID of registered application.
- notifyType: must be *sysNotifyCFEvent*, defined in notify.h.
- priority: not used, should be 0.

Note that an application may, in theory, remain registered after exiting. However, any system event that causes the application's database ID to change (such as hot-syncing a new copy of the application) may cause a system crash during the next CF event.

The notification manager requires PalmOS 3.3 header files or above.

SECTION 3 - FFS LIBRARY FUNCTION REFERENCE

This section contains an alphabetical listing of the functions available in the Ffs library, along with a brief description of each. Except where noted, these functions were designed to mimic common low-level file and disk functions available with most C compilers. The descriptions which follow name the closest standard C function and concentrate on the differences between the standard version (if any) and the Ffs version.

All library functions require an open library reference of type *UInt16* as their first parameter. This reference is used by the Palm OS to locate the jump table for the library, and is returned by the **SysLibLoad** call.

Note that the range of *errno* values set by a given function may be different than the standard C version (if any). In general, this difference is not noted individually.

SHARED DATA TYPES AND STRUCTURES

This section details the shared data types and structures used by multiple functions in the Ffs library. Note that the library uses the following data types, which may not be defined in the pre-OS 3.3 headers:

- Int8 formerly defined as *char*
- UInt8 formerly defined as *Byte*
- Int16 formerly defined as *Int*
- UInt16 formerly defined as *Word*
- Int32 formerly defined as *long*
- UInt32 formerly defined as *DWord*

DRIVENUM

The *drivenum* parameter is interpreted as follows:

0 Current default CF card.
1 First CF card.
2 Second CF card.
3 etc.

Currently only 1 CF card is supported. Thus, *drive* may be either 0 or 1, both indicating the first (only) card.

STRUCT FFBLK

```
typedef struct {
        char    ff_reserved[21];
        char    ff_attrib;
        Int16   ff_ftime:
        Int16   ff_fdate;
        Int32   ff_fsize;
        char    ff_name[13];
        char    ff_longname[256];
} ffblk;
```

The *ffblk* fields have the following meanings:

ff_reserved:	reserved by system, do not modify.
ff_attrib:	file attributes (see below).
ff_ftime:	file creation time (see below).
ff_fdate:	file creation date (see below).
ff_fsize:	file size in bytes.
ff_name:	file name in 8.3 format.
ff_longname:	long file name, if any.

The file attributes specified in parameter *attrib* and returned in *ff_attrib* consist of a bitwise OR of one or more of the following constants:

FA_NORMAL	"Normal" file (no attributes set).
FA_RDONLY	Read-only file.
FA_HIDDEN	Hidden file.
FA_SYSTEM	System file.
FA_LABEL	Disk volume label.
FA_DIREC	Subdirectory entry.
FA_ARCH	Archive bit (currently ignored during matches because it is always set).
FA_ALL	Only used by *attrib* parameter – matches all directory entries.

The creation time is bit-encoded in *ff_ftime* as follows:

bits 0 to 4:	Seconds divided by 2
bits 5 to 10:	Minutes
bits 11 to 15	Hours

The creation date is bit-encoded in *ff_fdate* as follows:

bits 0 to 4:	Day
bits 5 to 8:	Month
bits 9 to 15:	Year since 1980 (ie. 1 means 1981)

The long filename *ff_longname* is an empty string unless the specified file has been assigned a long file name. When creating files using the Ffs Library, a long file name is assigned only when the filename specified by the caller is longer than the 8.3 format. Long filenames created by the Ffs library are converted to all uppercase and may be up to 255 characters in length.

STRUCT STAT

```
typedef struct {
        Int16    st_dev;
        Int16    st_ino;
        UInt32   st_mode;
        Int16    st_nlink;
        Int16    st_uid;
        Int16    st_gid;
        Int16    st_rdev;
        UInt32   st_size;
        date_t   st_atime;
        date_t   st_mtime;
        date_t   st_ctime;
        UInt8    st_attr;
} stat;
```

The *stat* fields are defined as follows:

st_dev:	Card number (currently always 1).
st_ino:	Not used.
st_mode:	File mode information (see below).
st_nlink:	Always 1.
st_uid:	Not used.
st_gid:	Not used.
st_rdev:	Same as *st_dev*.
st_size:	File size in bytes.
st_atime:	Same as *st_ctime*.
st_mtime:	Same as *st_ctime*;
st_ctime:	Creation time/date (see below).
st_attr:	File attributes (see below).

The file mode gives information about the type of file and the current read/write mode. It consists of one of the file type constants bitwise OR'd with one or both of the file access constants. The file type constants are defined as:

S_IFCHR	Character special device (not used).
S_IFDIR	Subdirectory.
S_IFBLK	Block special device (not used).
S_IFREG	Regular file.

The file access constants are identical to the *mode* constants used when a file is opened with **FfsOpen**, and are defined as:

S_IREAD	File opened with read permission (always true).
S_IWRITE	File opened with write permission.

The time/date fields, *st_atime*, *st_mtime*, and *st_ctime*, are of type *date_t*, defined as:

```
typedef struct {
        UInt16   date;
```

```
        UInt16  time;
} date_t;
```

The creation date and time are bit-encoded in the *date* and *time* fields, respectively. The *date* and *time* bit-encodings are identical to the encodings used by the *ffblk* fields *ff_fdate* and *ff_ftime*, respectively. Note that the Borland C *stat* structure defines *date_t* as an Int32.

The file attributes in the *st_attr* field are defined identically to those returned in the *ffblk* field *ff_attrib*.

FFSCARDISATA

Purpose Check for the presence of an ATA storage card.

Prototype

```
Boolean FfsCardIsATA(UInt16 libRef,
                     UInt8  drivenum);
```

Parameters

`-> libRef`	The library reference number.
`-> drivenum`	Card number to check (See Shared Data Types).

Result

`true`	An ATA card is in the slot.
`false`	No ATA card detected in slot.

Comments This routine is much slower than **FfsCardIsInserted**, as if requires powering the slot (200 msec delay if slot is not already powered).

FFSCARDISINSERTED

Purpose Checks for presence of a CF card in the slot.

Prototype

```
Boolean FfsCardIsInserted(UInt16 libRef,
                          UInt8  drivenum);
```

Parameters

`-> libRef`	The library reference number.
`-> drivenum`	Card number to check (See Shared Data Types).

Result

`true`	A CF card is inserted in the slot
`false`	No CF card is inserted in the slot.

Comments The card does not need to be powered for detection. This routine does not differentiate between ATA and non-ATA cards. See also **FfsCardIsATA**.

FFSCHDIR

Purpose Changes the current working directory.

Prototype

```
Err FfsChdir(UInt16 libRef,
             char   *path);
```

Parameters

`-> libRef`	The library reference number.
`-> Path`	New working directory.

Result

`0`	Success.
`-1`	Failure.

Comments All paths not starting at root are assumed relative to the current working directory. Use **FfsGetcwd** to get the current working directory. Similar to: Borland C getcwd.

FFSCLOSE

Purpose Closes a file previously opened by **FfsOpen** or **FfsCreat**

Prototype

```
Err FfsClose(Uint16 libRef,
             Int16  handle);
```

Parameters

`-> libRef`	The library reference number.
`-> Handle`	open file handle.

Returns

`0`	Success
`-1`	Failure

Comments Similar to: Borland C *close*, K&R *close*

FFSCREAT

Purpose Create a new file, or truncate an existing file.

Prototype

```
Int16 FfsCreat(Uint16 libRef,
               char *path,
               Int16 mode);
```

Parameters

`-> libRef`	The library reference number.
`-> path`	Name of file to create.
`-> mode`	Read/write permission for the file (see **FfsOpen** for bit definitions).

Result

`-1`	Failure
`Nonzero`	Handle for new file on success

Comments This is actually the same as calling **FfsOpen** with *flags* set to PO_RDWR | PO_CREAT | PO_TRUNC. See **FfsOpen** for more details. Similar to: Borland C *creat*, K&R *creat*.

FFSEOF

Purpose Checks if the current file pointer for a specified file is at the end of the file.

Prototype

```
Err FfsEof(Uint16 libRef,
           Int16  handle);
```

Parameters

`-> libRef`	The library reference number.
`-> handle`	Open file handle.

Result

`0`	File pointer is not at the end of the file
`1`	File pointer is at the end of the file
`-1`	Error has occurred

Comments Similar to: Borland C *eof*.

FFSFINDDONE

Purpose Frees the resources used during a **FfsFindfirst**/**FfsFindnext** directory search. Must be called before starting another search. See **FfsFindfirst** for details.

Prototype

```
Err FfsFinddone(UInt16 libRef,
                ffblk  *ff_blk);
```

Parameters

`-> libRef`	The library reference number.
`-> ff_blk`	Pointer to an ffblk structure previously used with **FfsFindfirst**/**FfsFindnext**.

Result

`0`	Success
`-1`	Failure

Comments Similar to: Borland C *eof*.

FFSFINDFIRST

Purpose Starts a directory search. Searches for all directory entries matching the specified path with the specified attributes

Prototype

```
Err FfsFindfirst(UInt16 libRef,
                 char   *path,
                 Int16  attrib,
                 ffblk  *ff_blk);
```

Parameters

`-> libRef`	The library reference number.
`-> path`	Directory and/or file specification to search. Wildcards allowed.
`-> attrib`	File attributes to match.
`<- ff_blk`	Pointer to a *ffblk* structure for returning directory entry information.

Result

`0`	Success
`-1`	Failure

Comments Note that the attributes must match exactly except for the FA_ARCH attribute (which is ignored), unless FA_ALL is specified (see below). Directory entries include not only files and subdirectories, but also disk volume labels. Similar to: Borland C *_dos_findfirst* (except *ffblk* field names differ), Borland C *findfirst* (except last two parameters reversed).

If a matching file is found, information about the file is returned in the *ffblk* structure:

Once the first matching entry in a directory is found, additional matching entries are returned by calls to **FfsFindnext**, until all matching entries are found. Once the search is complete, **FfsFinddone** must be called to free allocated resources, before the next call to **FfsFindfirst**. Because the *attrib* parameter must match an entry exactly (except as noted above), it is normally easiest to use FA_ALL when searching a directory and then perform the attribute filtering manually.

Note on wildcard matches:

If you do not understand the concept of long and short filename equivalence under the FAT File System, please refer to one of the following books:

- Inside the Windows 95 File System
- Windows 98 Resource Kit

If the wildcard is “*.*”, it will match all filenames. The following are examples of matches:

- “*.*” = “ABC.PDB”
- “*.*” = “Abcdefghijklmnopqrstuvwyz.pdb”

If the wildcard request contains either of the wildcard characters (‘?’ or ‘*’) and fits the short filename format (8.3), the search will be performed against the short filenames. Case will be ignored. The following are examples of matches:

- “a*.pdb” = “ABC.PDB”
- “a?.pdb” = “AB.PBD”

If the wildcard request is only for the extension, the search will be performed against short filenames. Case will be ignored. The following are examples of matches.

- “*.pdb” = “ABC.PDB”
- “*.pdb” = “abcdefghijklmnopqrstuvwxyz.pdb”

This function does not support long filenames with wildcard characters. The only successful search will be an exact match.

- “abcdefghijkl.pdb” = “abcdefghijkl.pdb”

The following searches will not match any filenames:

- “abcdefghijkl*.pdb”
- “abcdefghijkl?.pdb”

FFSFINDNEXT

Purpose Continue a directory search started by a call to **FfsFindfirst**. See **FfsFindfirst** for details. If a match is found, file information is returned in the *ffblk* structure.

Prototype

```
Err FfsFindnext(UInt16 libRef,
                ffblk *ff_blk);
```

Parameters

`-> libRef`	The library reference number.
`-> ff_blk`	Pointer to a *ffblk* structure previously used with **FfsFindfirst**.

Result

`0`	Success
`-1`	Failure

Comments Similar to: Borland C *_dos_findnext* (except *ffblk* field names differ), Borland C *findnext*.

FFSFLUSH

Purpose Flush open file to flash. All dirty buffers associated with the file are written to the CF card. Same general function as Borland C *fflush* and K&R *fflush*, though parameters and return values are different.

Prototype

```
Err FfsFlush(Uint16 libRef,
             Int16  handle);
```

Parameters

`-> libRef`	The library reference number.
`-> handle`	Open file handle.

Result

`0`	Success
`-1`	Failure

Comments Similar to: Borland C *_dos_findnext* (except *ffblk* field names differ), Borland C *findnext*.

FFSFLUSHDISK

Purpose Flush buffers associated with all open files to CF card. This is done automatically when the library is closed with **FfsLibClose**.

Prototype

```
Err FfsFlushdisk(Uint16 libRef,
                 Uint16 drivenum);
```

Parameters

`-> libRef`	The library reference number.
`-> drivenum`	Card number to flush (See Shared Data Types).

Result

`0`	Success
`-1`	Failure

Comments Same general function as Borland C *flushall* (but does not return number of open files, and only operates on specified drive), K&C *fflush* with parameter NULL (but only operates on specified drive).

FFSFORMAT

Purpose Formats the specified card.

Prototype
```
Err FfsFormat(UInt16 libRef,
              UInt16 drivenum);
```

Parameters

`-> libRef`	The library reference number.
`-> drivenum`	card to format (See Shared Data Types).

Result

`0`	Success
`-1`	Failure

FFSFSTAT

Purpose Return information on an open file, specified by the file's handle. Information is returned in a *stat* structure.

Prototype

```
Err FfsFstat(Uint16 libRef,
             Int16  handle,
             stat   *pstat);
```

Parameters

`-> libRef`	The library reference number.
`->Handle`	Open file handle.
`->Pstat`	Pointer to a *stat* structure for returned file information.

Result

`0`	Success
`-1`	Failure

Comments Similar to: Borland C *fstat* (except *stat* differences noted above), K&R *fstat* (except some *stat* fields have differing types).

FFSGETCWD

Purpose Returns the current working directory of the current default drive. If the current working directory path is longer than (*numchars* – 1), NULL is returned and *errno* is set.

Prototype

```
char *FfsGetcwd(Uint16 libRef,
                char   *path,
                Int16  numchars)
```

Parameters

`-> libRef`	The library reference number.
`-> path`	If *path* is non-NULL, and the working directory path is shorter than *numchars*, the working directory path will be copied into *path* and the returned pointer will point to *path*. If *path* is NULL and the current working directory path is shorter than *numchars*, a buffer will be allocated of size *numchars*, the current working directory path will be copied into the buffer, and a pointer to the buffer will be returned to the caller. In this case, it is the responsibility of the caller to free the buffer using **MemPtrFree** when finished with it.
`-> numchars`	Maximum length of path buffer, including terminator

Result

`NULL`	Error
`Non NULL`	Pointer to buffer containing path, or NULL if error (see below).

Comments Similar to Borland C getcwd.

FFSGETDISKFREE

Purpose Returns free space and total space available on card. Total and free space is returned as number of clusters, along with sector-per-cluster and bytes-per-sector information. Thus, the caller may translate free and total size to sectors or bytes, if desired.

Prototype

```
Err FfsGetdiskfree(Uint16      libRef,
                   UInt8       drivenum,
                   Diskfree_t *dtable);
```

Parameters

`-> libRef` The library reference number.

`-> drivenum` Card to use. Although an *unsigned char* rather than an *unsigned int* this is the same value as is in the Shared Data Types section.

`Dtable:` Pointer to *diskfree_t* structure for return of disk information. The *diskfree_t* structure is identical to Borland's *diskfree_t* structure and, except for the field names, to Borland's *dfree* structure. It is defined as follows:

```
Typedef struct {
        UInt32  avail_clusters;
        UInt32  total_clusters;
        UInt16  bytes_per_sector;
        UInt16  sectors_per_cluster;
} diskfree_t;
```

The *diskfree_t* fields are defined as follows:

Avail_clusters: Number of free clusters on the card.
Total_clusters: Total number of free and used clusters.
Bytes_per_sector: Number of bytes in a sector.
Sectors_per_cluster: Number of sectors in a cluster.

The total card size in bytes may thus be found my multiplying togather *total_clusters*, *sectors_per_cluster*, and *bytes_per_sector*.

Result

`0` Success

`-1` Failure

Comments This function follows Borland's convention. Similar to: Borland C *dos_getdiskfree*, Borland C *getdfree* (except for structure and field names).

FFSGETDRIVE

Purpose Return the current default drive number. A value of 1 indicates the first drive, 2 the second drive, etc. Currently, only one (the first) drive is supported, and it is always the default.

Prototype

```
void FfsGetdrive(UInt16 libRef,
                 UInt16 *drivenum);
```

Parameters

`-> libRef`	The library reference number.
`<- drivenum`	Pointer to variable for return of card number (See Shared Data Types).

Result `None`

Comments Similar to: Borland C *_dos_getdrive*, Borland C *_getdrive* (except drive is returned in parameter, not by function), Borland C *getdisk* (except drive number starts at 1 and is returned in parameter, not by function).

FFSGETERRNO

Purpose Get the current value of the *errno* variable.

Prototype

```
Int16 FfsGetErrno(UInt16 libRef);
```

Parameters

`-> libRef`	The library reference number.

Result

`0`	If no errors have occurred since library was opened.
`Nonzero`	Current value of the global *errno* variable, or 0

Comments This internal library global is set to a specific error code any time a library call fails. It is not reset when a library call succeeds, so it always has the value of the last error. It is set to 0 when the library is opened. The *errno* error codes are listed in section 4.

FFSGETFILEATTR

Purpose Given the name of a directory entry, return its attributes. Usually the entry will be a filename, but it could be a subdirectory name or a volume label

Prototype

```
Err FfsGetfileattr(Uint16 libRef,
                   char   *name,
                   UInt16 *attr);
```

Parameters

`-> libRef`	The library reference number.
`-> name`	Name of directory entry.
`-> attrib`	Pointer to variable for return of attributes (see **FfsFindfirst** for attribute definitions).

Result

`0`	Success
`-1`	Failure

Comments Similar to: Borland C *getfileattr* (except attribute names preceded by “F” rather than underscore, following the Borland C *findfirst*/*findnext* convention).

FFSGETLIBAPIVERSION

Purpose Get the Ffs library version. Other than FfsLibOpen, this is the only library call that can be made without first opening the library.

Prototype

```
Err FfsGetLibAPIVersion(UInt16 libRef,
                        UInt32 *dwVerP);
```

Parameters

`-> libRef`	The library reference number.
`<- dwVerP`	Pointer to UInt32 variable for library version. Version format follows the Palm OS versioning scheme.

Result

0	Success
FFS_ERR_BAD_PARAM	DwVerP is a NULL pointer.

Comments The format of the version Uint32 is documented in the Palm OS header files.

FFSINSTALLERRORHANDLER

Purpose Installs a user-supplied critical error handler. The user critical error function must take an integer *choices* parameter and an error message string parameter, and return an integer *action* value. The *choices* parameter indicates the valid actions allowed; the user error handler chooses one of the allowed actions and Result it. The error message string indicates specifically what type of error occurred, and may be displayed to the user. See section 2.4 for details.

If a user error handler is not installed, all critical errors will cause the current operation to abort.

Prototype

```
id FfsInstallErrorHandler(UInt16 libRef,
                          Int16 (*CritErr)(Int16, Int16, char *));
```

Parameters

`-> libRef`	The library reference number.
`-> CritErr`	Pointer to a user critical error handler function (see below).

Result

`0`	Success
`-1`	Failure

Comments The critical error handler *must* be uninstalled when the application exits.

FFSISDIR

Purpose Checks if the specified path points to a directory.

Prototype

```
Err FfsIsDir(Uint16  libRef,
             char    *path,
             Boolean *is_dir);
```

Parameters

`-> libRef`	The library reference number.
`-> path`	Path to test
`<- is_dir`	Set to **true** if the path is a directory, **false** otherwise

Result

`0`	Success
`-1`	Failure

FFSLIBCLOSE

Purpose Close the Ffs library, freeing allocated resources.

Prototype

```
Err FfsLibClose(UInt16 libRef);
```

Parameters

`-> libRef`	The library reference number.

Result

`0`	Success
FFS_ERR_LIB_IN_USE	Another application has the library open (currently not supported)

Comments The library should be closed when the calling application exits. This function also flushes all dirty buffers to the CF card.

FFSLIBOPEN

Purpose Opens the Ffs library. The library must have been previously loaded with **SysLibLoad**. This function allocates the library global data block, sets up the hardware for the CF slot, and resets the CF card, if present.

Prototype

```
Err FfsLibOpen(UInt16 libRef);
```

Parameters

`-> libRef`	The library reference number.

Result

`0`	Success
FFS_ERR_NO_MEMORY	Failure if not enough RAM available for allocating global variables

Comments The library must have been previously loaded with **SysLibLoad**.

FFSLSEEK

Purpose Move the current file pointer to the specified offset. The offset is relative to either the beginning of the file, the end of the file, or the current file pointer, as specified by *origin*

Prototype

```
Int32 FfsLseek(UInt16 libRef, Int16 handle, Int32 offset,
Int16 origin);
```

Parameters

`-> libRef` The library reference number.

`-> handle` Handle for an open file.

`-> offset` New file offset to seek to (relative to *origin*).

`-> origin` Constant indicating seek offset origin:

SEEK_SET:	offset from beginning of file.
SEEK_CUR:	offset from current file pointer.
SEEK_END:	offset from end of file.

Result

`-1` Failure

`!= -1` New file pointer offset

Comments The new file pointer is returned. This is an offset relative to the beginning of the file, so it will not necessarily match the *offset* parameter even if the seek is successful. Similar to: Borland C *lseek*, K&R *lseek*.

FFSMKDIR

Purpose Creates a subdirectory. Parameter *path* may specify a complete or partial path.

Prototype

```
Err FfsMkdir(UInt16 libRef,
             char   *path);
```

Parameters

`-> libRef`	The library reference number.
`-> path`	Name (path) of new directory to create

Result

`0`	Success
`-1`	Failure

Comments Similar to: Borland C *mkdir*.

FFSOPEN

Purpose To open or create a file.

Prototype

```
Int16 FfsOpen(UInt16 libRef,
              char   *path,
              Int16  flags,
              Int16  mode);
```

Parameters

`-> libRef` The library reference number.

`-> path` Name of the file to open.

`-> flags` File access mode. The file is opened with permissions set by the *flags* parameter, which consists of a bitwise OR of one or more of the following constants

O_RDONLY	Open for read access only.
O_WRONLY	Open for write access only.
O_RDWR	Open for read/write access.
O_APPEND	Append all writes to end of file.
O_CREAT	Create the file if it does not already exist.
O_TRUNC	If the file exists, truncate it to 0 size.
O_EXCL	If O_CREAT is also specified, and file already exists, fail
O_TEXT	Open in text mode (not implemented).
O_BINARY	Open in binary mode (always true).
O_NOSHAREANY	Once open, do not allow other opens of this file.
O_NOSHAREWRITE	Once open for writing, do not allow other opens of this file.

`-> mode` File creation mode. If the O_CREAT flag is not specified, the *mode* parameter is ignored and may be set to 0. Otherwise, *mode* determines the read/write attribute on the newly created file:

S_IREAD	Read permitted (always true).
S_IWRITE	Write permitted.

Result

`!= -1` Open file handle

`-1` Failure

Comments Similar to: Borland C *open* (but *fmode* global is not supported), K&R *open* (but allows non-existent files to be created by specifying O_CREAT).

FFSREAD

Purpose Read the specified number of bytes from an open file.

Prototype

```
Int16 FfsRead(UInt16 libRef,
              Int16  handle,
              Void   *buffer,
              Int16 num_bytes);
```

Parameters

`-> libRef`	The library reference number.
`-> handle`	Open file handle.
`<- buffer`	Pointer to the buffer to store the read data
`-> num_bytes`	The *num_bytes* parameter is interpreted as an UInt16, allowing up to 65534 (0xFFFE) bytes to be read. The return value is also interpreted as an UInt16 unless set to –1 (65535, or 0xFFFF), which indicates an error. For this reason, a full 65535 (0xFFFF) bytes cannot be read, and if *num_bytes* is set to this value, no data will be read and *errno* will be set to indicate an invalid parameter

Result

`-1`	Failure
`!= -1`	Actual number of bytes read

Comments Similar to: Borland C *_read* (except *num_bytes* is a signed integer), Borland C *read* (except *num_bytes* is a signed integer and CTRL-Z is ignored), K&R *read* (except *buffer* is a void pointer).

FFSREMOVE

Purpose Removes a file. Parameter *path* must point to a simple file that is not open.

Prototype

```
Err FfsRemove(UInt16 libRef,
              char   *path);
```

Parameters

`-> libRef`	The library reference number.
`-> path`	File to remove.

Result

`0`	Success
`-1`	Failure

Comments This function call is identical to **FfsUnlink**. Similar to: Borland C *remove*, K&R *remove*.

FFSRENAME

Purpose Changes the name of a directory entry (file, subdirectory, or volume label). Parameter *new_name* must be a simple name, not a path (thus files cannot be moved with this function).

Prototype

```
Err FfsRename(UInt16 libRef,
              char   *path,
              char   *new_name);
```

Parameters

`-> libRef`	The library reference number.
`-> path`	Path to directory entry to rename
`-> new_name`	New name for entry.

Result

`0`	Success
`-1`	Failure

Comments Similar to: Borland C *rename* (but does not move files), K&R *rename*.

FFSRMDIR

Purpose Remove a subdirectory. Directory must be empty and must not be the current working directory.

Prototype

```
Err FfsRmdir(UInt16 libRef,
             char   *dirname);
```

Parameters

`-> libRef`	The library reference number.
`-> dirname`	Directory to remove

Result

`0`	Success
`-1`	Failure

Comments Similar to: Borland C *rmdir*.

FFSSETDRIVE

Purpose Set the current default card (drive), and return the total number of available cards. Currently, only one card is supported.

Prototype

```
void FfsSetdrive(Uint16 libRef,
                 Uint16 drivenum,
                 UInt16 *ndrives);
```

Parameters

`-> libRef`	The library reference number.
`-> drivenum`	Card number to set as default (See Shared Data Types).
`-> ndrives`	Pointer to variable for return of the number of drives available.

Result `None`

Comments Similar to: Borland C *_dos_setdrive*, Borland C *setdisk* (except drive numbers start at 1, and number drives is returned in additional parameter).

FFSSETFILEATTR

Purpose Set the attributes for a file.

Prototype

```
Err FfsSetfileattr(UInt16 libRef,
                   char   *name,
                   UInt16 attr);
```

Parameters

`-> libRef` The library reference number.

`-> name` Name of the file

`-> attr` New attributes to set. The *attr* parameter is a bitwise OR of one or more of the following constants:

FA_ATTRIB	Set archive bit.
FA_NORMAL	“Normal” file.
FA_RDONLY	Read only.
FA_HIDDEN	Hidden file.
FA_SYSTEM	System file.

Result

`0` Success

`-1` Failure

Comments Similar to: Borland C *_dos_setfileattr* (except attribute constants prefixed by “F” rather than underscore, following Borland C *findfirst*/*findnext* convention).

FFSSTAT

Purpose Similar to **FfsFstat**, except file is specified by name, and thus need not be open. See **FfsFstat** for details.

Prototype

```
Err FfsStat(UInt16 libRef,
            char   *path,
            stat   *pstat);
```

Parameters

`-> libRef`	The library reference number.
`-> path`	Directory entry to query
`-> pstat`	Pointer to a stat structure for return of statistics

Result

`0`	Success
`-1`	Failure

Comments Similar to: Borland C *stat* (same differences as noted for *fstat*), K&R *stat* (same differences as noted for *fstat*).

FFSTELL

Purpose Returns the current offset position within the open file.

Prototype Int32 FfsTell(UInt16 libRef, Int16 handle);

Parameters

`-> libRef`	The library reference number.
`-> handle`	Open file handle.

Result

`-1`	Failure
`!= -1`	Current file pointer

Comments Similar to: Borland C *tell*.

FFSUNINSTALLERRORHANDLER

Purpose Uninstall the current user critical error handler, if any. After uninstall, the system critical error handler will always abort the current operation when a critical error occurs.

Prototype void FfsUnInstallErrorHandler(UInt16 libRef);

Parameters `-> libRef` The library reference number.

Result `None`

Comments Applications must call this when exiting if an error handler was installed. See **FfsInstallErrorHandler**.

FFSUNLINK

Purpose Removes a file. Parameter *path* must point to a simple file that is not open.

Prototype

```
Err FfsUnlink(UInt16 libRef,
              Char   *path);
```

Parameters

`-> libRef`	The library reference number.
`-> path`	File to remove.

Result

`0`	Success
`-1`	Failure

Comments This function call is identical to **FfsRemove**. Similar to: Borland C *remove*, K&R *remove*.

FFSWRITE

Purpose Writes the specified number of bytes to a file.

Prototype

```
Int16 FfsWrite(UInt16 libRef,
               Int16 handle,
               void *buffer,
               Int16 num_bytes);
```

Parameters

`-> libRef`	The library reference number.
`-> handle`	Open file handle.
`<- buffer`	Pointer to the buffer of the data to write
`-> num_bytes`	Note that *num_bytes* is interpreted as UInt16, and may range up to 65534 (0xFFFE). The return value is similarly treated as UInt16 except for the error case of –1 (0xFFFF). Thus, if *num_bytes* is set to 65535 (0xFFFF), it is treated as an error and no data is written.

Result

`-1`	Write failed
`!= -1`	Number of bytes written. If fewer bytes are actually written than were requested, it probably indicates a full CF card.

Comments Similar to: Borland C *_write* (except *num_bytes* is signed), Borland C *write* (except *num_bytes* is signed and no CR/LF translation performed), K&R *write* (except *buffer* is a void pointer).

SECTION 4 – FFS LIBRARY ERROR CODES

When an error occurs during a Ffs library call, some indication of the error is returned by the function to the caller (usually a value of –1, versus a value of 0 on success). However, specific error information is stored in an internal *errno* variable, and must be obtained by calling **FfsGetErrno**. The *errno* variable may take on any of the following values after a failed function call. It is initialized to 0 when the library is opened, and is not cleared after a successful call.

These constants are defined in ffslib.h.

ENOENT	File or path not found.
ENOMEM	Not enough memory available.
EBADF	Invalid file handle.
EACCESS	File access violation (ie. writing to a read-only file).
EINVDRV	Invalid drive number specified.
EEXIST	Trying to create a file with O_EXCL, but file exists.
EINVAL	Invalid parameter.
ENFILE	No free file handles available.
ENOSPC	No space on CF card.
ESHARE	Open failed due to sharing violation.
ENODEV	No CF card found.
ERANGE	Result is out of range.
EIOERR	CF card IO error occurred.

If the error occurs during initialization of the library (either when it is opened or when a new card is detected), the following low-level driver error codes may be set:

BUS_ERC_DIAG	Drive diagnostic failed
BUS_ERC_ARGS	Bad argument during initialization
BUS_ERC_DRQ	Drive DRQ is not valid.
BUS_ERC_TIMEOUT	Timeout during an operation
BUS_ERC_STATUS	Controller reported an error
BUS_ERC_ADDR_RANGE	LBA out of range
BUS_ERC_CNTRL_INIT	Fail to initialize controller
BUS_ERC_IDDRV	Identify drive info error
BUS_ERC_CMD_MULT	Read/Write Multiple Command error
BUS_ERC_BASE_ADDR	Base Address not valid
BUS_ERC_CARD_ATA	Card is not ATA

SECTION 5 – FFS LIBRARY EXAMPLE CODE

The Ffs SDK zip file contains a demonstration application that uses Ffs Library. The demonstration application is a CodeWarrior project with all source code in the *src* subdirectory. The Ffs library header file, ffslib.h, and the notification manager header file notify.h, are also in the *src* subdirectory.

CHAPTER FIVE – AUDIO LIBRARY

This chapter is intended to introduce the use of, and provide a reference to, the Audio Library procedures. It is directed toward Palm OS application developers who wish to access enhanced audio functionality of the TRGpro. It is assumed that the reader is familiar with the C programming language, in particular within the context of the Palm OS. Section 1 of this document describes the use of shared libraries in Palm OS applications, summarizing the functionality provided by the Audio Library. Section 2 details each of the library calls, describing their function, parameters, and return value. Section 3 lists the possible error return codes, and Section 4 discusses a sample project.

SECTION 1 - AUDIO LIBRARY OVERVIEW

THE PURPOSE OF THE AUDIO LIBRARY

The Audio library provides an interface to the enhanced audio TRGpro. It includes routines to control the master volume as well as routines to play DTMF tones.

LOADING, UNLOADING, AND ACCESSING THE AUDIO SHARED LIBRARY

Currently, the Audio Library is implemented as a Palm OS shared library. To access the Audio calls, an application must search for the library, load the library if not found, then open the library. Opening the library returns a library reference number that is used to access the individual functions within the library. When the application is finished with the library, it should close the library. The current implementation of the Audio library supports only single-user access; applications cannot share an open library. If one application leaves the Audio library open upon exit and another application finds the library and begins using library calls, the results are unpredictable.

The calling application must include the header file AudioLib.h. This file contains the required constant definitions, structure typedefs, and function prototypes for the library. In addition, this file maps library functions calls to the corresponding system trap instructions, through which all library routines are accessed.

To find a loaded library, an application calls **SysLibFind**, specifying the library. If not found, an application loads the library using **SysLibLoad**, specifying the library type and creator IDs. For the Audio library, the name, type and creator IDs are defined in AudioLib.h as Audio*LibName*, Audio*LibTypeID* and Audio*LibCreatorID*, respectively. After loading the library, it must be opened with a call to **AudioLibOpen**.

Once the library is open, the application may make calls to the library functions. The first parameter to a library call is always the library reference number returned when the library is loaded. Most library calls return an integer result of 0 on success and a non-zero error code on failure. The application that opens the library is responsible for closing and unloading the library. The library is closed with the **AudioLibClose** call, and unloaded with the **SysLibRemove** call. It

is possible for an application to leave the library loaded when exiting. The library may then be accessed by other applications through the **SysLibFind** command, which returns a reference to an already-loaded library. Once the reference number is obtained, the library is opened as usual with **AudioLibOpen** call. In either case, however, the caller must open the library on startup and close it on exit. The library cannot be left open between applications.

Currently, the name of the Audio library used for **SysLibFind** is "Audio.lib," the creator ID is "trgA", and the type ID is "libr". These constants are all defined in AudioLib.h.

SUMMARY OF AUDIO LIBRARY FUNCTIONS

The Audio library calls may be grouped into two categories: audio management and library management. The calls, grouped by category, are listed below, with brief descriptions of each call's function. An alphabetical listing with a detailed specification of each call is given in section 2.

Audio management

- AudioSetMasterVolume — Set the master volume.
- AudioGetMasterVolume — Get the master volume setting.
- AudioSetMute — Turn muting on and off.
- AudioGetMute — Get current mute setting.
- AudioPlayDTMFChar — Play ASCII char as DTMF tone
- AudioPlayDTMFStr — Play ASCII string and DTMF tones.

Library management

- AudioLibClose — Close the Audio library.
- AudioGetLibAPIVersion — Get the library version number.
- AudioLibOpen — Open the Audio library.

SECTION 2 – AUDIO LIBRARY FUNCTION REFERENCE

This section contains an alphabetical listing of the functions available in the Audio Library, along with a brief description of each.

All library functions require an open library reference of type *UInt16* as their first parameter. This reference is used by the Palm OS to locate the jump table for the library, and is returned by the **SysLibLoad** call.

SHARED DATA TYPES

This section details data types used by multiple functions in the Audio library. Note that the library uses the following data types, which may not be defined in the pre-OS 3.3 headers:

- Int8 formerly defined as *char*
- UInt8 formerly defined as *Byte*
- Int16 formerly defined as *Int*
- UInt16 formerly defined as *Word*
- Int32 formerly defined as *long*
- UInt32 formerly defined as *Dword*

MUTE

The mute parameter can have one of the following values:

- AUDIO_MUTE_OFF, //Mute is set to off and volume is set to current setting.
- AUDIO_MUTE_ON //Mute is set on and therefore the volume is set to zero.

VOLUME

The volume parameter can be set between AUDIO_VOLUME_MIN and AUDIO_VOLUME_MAX.

FUNCTIONS

AUDIOGETMASTERVOLUME

Purpose Get the master volume setting.

Prototype

```
Err AudioGetMasterVolume(UInt16 libRef,
                         UInt8 *volume)
```

Parameters

`->libRef`	Library reference number
`->volume`	Pointer to Uint8 variable for returning the volume setting (see Shared Data Types)

Result Always returns 0

Comments

AUDIOSETMASTERVOLUME

Purpose Set the master volume setting.

Prototype

```
Err AudioSetMasterVolume(UInt16 libRef,
                         UInt8  volume)
```

Parameters

`-> libRef`	Library reference number
`-> volume`	New master volume setting (see Shared Data Types)

Result Always returns 0.

Comments

AUDIOSETMUTE

Purpose Turn mute on or off. This sets the volume to 0 or restores this to the previous setting.

Prototype

```
Err AudioSetMute(UInt16    refNum,
                 mute_type mute)
```

Parameters

`-> libRef`	Library reference number
`-> mute`	AUDIO_MUTE_OFF restores the volume to the store preference setting. AUDIO_MUTE_ON saves the current volume to the store preferences and sets the volume to 0. (See Shared Data Types).

Result 0 on success or error code on failure

Comments

AUDIOGETMUTE

Purpose Get the current mute setting.

Prototype

```
Err AudioGetMute(Uint16     refNum,
                 Mute_type *mute)
```

Parameters

`-> libRef`	library reference number
`<- mute`	Current mute setting returned in mute (See Shared Data Types).

Result 0 on success or error code on failure

Comments

AUDIOPLAYDTMFCHAR

Purpose Play the ASCII char as a DTMF tone.

Prototype

```
void AudioPlayDTMFChar(UInt16 refNum,
                       char   ascChar,
                       Int16  toneLength)
```

Parameters

`-> libRef`	Library reference number
`-> ascChar`	ASCII char to play. Valid characters are '0', '1', '2', '3', '4', '5', '6', '7', '8', '9', 'A', 'B', 'C', 'D', '#', and '*'.
`-> toneLength`	Length of tone defined in 1/16ths of a second

Result 0 on success or error code on failure

Comments

AUDIOPLAYDTMFSTR

Purpose Play the ASCII string as DTMF tones.

Prototype

```
void AudioPlayDTMFStr(UInt16 refNum,
                      char   *ascStr,
                      Int16  toneLength,
                      Int16  toneGap)
```

Parameters

`-> libRef`	Library reference number
`-> ascStr`	The string must be comprised of the following characters: ‘0’, ‘1’, ‘2’, ‘3’, ‘4’, ‘5’, ‘6’, ‘7’, ‘8’, ‘9’, ‘A’, ‘B’, ‘C’, ‘D’, ‘#’, and ‘*’.
`-> toneLength`	Length of tone defined in 1/16ths of a second.
`-> toneGap`	Time to delay between each char in 1/10ths of a second.

Result 0 on success or error code on failure

Comments

AUDIOGETLIBAPIVERSION

Purpose Get the Audio library version

Prototype

```
Err AudioGetLibAPIVersion(Uint16 libRef,
                          UInt32 *dwVerP);
```

Parameters

`->libRef`	Library reference number
`->dwVerP`	Pointer to UInt32 variable for library version. Version format follows the Palm OS versioning scheme.

Result 0 on success or error code on failure

Comments Other than AudioLibOpen, this is the only library call that can be made without first opening the library.

AUDIOLIBCLOSE

Purpose Close the Audio library, freeing allocated resources.

Prototype
```
Err AudioLibClose(UInt16 libRef);
```

Parameters

`->libRef`	Library reference number

Result 0 on success, AUDIO_ERR_LIB_IN_USE if another application has the library open (currently not supported).

Comments Currently, the library must be closed when the calling application exits.

AUDIOLIBOPEN

Purpose Open the Audio library

Prototype
```
Err AudioLibOpen(UInt16 libRef);
```

Parameters

`->libRef`	Library reference number

Result Always returns 0.

Comments The library must have been previously loaded with **SysLibLoad**.

SECTION 3 – AUDIO LIBRARY ERROR CODES

The following is a list of the error return codes from the Audio library calls. These are defined in AudioLib.h.

AUDIO_ERR_BAD_PARAM	A parameter is invalid.
AUDIO_ERR_LIB_NOT_OPEN	Trying to close an library that is not open.
AUDIO_ERR_LIB_IN_USE	Trying to close a library opened by another application.

SECTION 4 – AUDIO LIBRARY EXAMPLE CODE

The Audio SDK zip file contains a demonstration application that uses Audio Library. The demonstration application is a CodeWarrior project with all source code in the *src* subdirectory. The audio library header file and the notification manager header file are also in the *src* subdirectory.

CHAPTER SIX - ENTERPRISE DEVELOPMENT SOLUTIONS

SUPPORTING CUSTOMERS THROUGH STANDARDS

The TRGpro computer and its supporting products are designed to allow enterprise customers and developers to develop and deploy solutions with minimal effort. TRG has been in the business of developing Palm hardware and software products that resolve key hurdles in the acceptance of Palm computers into enterprise applications since the Palm computer was introduced. Our software product line enables unique solutions for all Palm computers that have Flash memory. This currently includes the TRGpro, Palm III, IIIx, V, and Vx computers.

IMAGEPRO

TRG has created a utility that performs a complete Flash load for Palm devices. ImagePro loads Flash memory on Palm devices with the selected OS and the selected application programs. Each enterprise or developer can customize the Palm device load to their unique requirements.

The enterprise central support staff can define and test a few standard enterprise Palm images using the ImagePro BUILD process.

Then the Palm device user can install the new enterprise Palm image on their Palm device using a standard Palm HotSync® process without involvement by the enterprise support staff. The user process performs the following:

- backups all programs and databases in the user's Palm (RAM and Flash) to the user's PC using the Backup Buddy program
- loads the Palm device Flash with the new Palm OS version or custom OS version, and with the new or additional application programs
- reloads to the Palm device (RAM) all original user programs and databases

ImagePro can be used to download Flash image files to a Palm III, IIIx, V, or Vx devices or IBM WorkPad 20x, 30x, or 40x devices via a PC running Windows 95, 98, or NT.

SFAPRO (SELF FLASHING APPLICATIONS)

SFApro is a TRG Products program that that will allow application developers to install their application programs in non-volatile Flash at the time of initial program installation. SFA will work similar to the way that the PC Install Shield works, in that the application developer has only to "wrap" the SFA process code around the application program in order to have the program reside in Flash memory on each user's Palm device.

All application developers, whether commercial developers or enterprise developers, view their programs as very important and they want to insure the user that in case of a hard reset, their program will still be available on the Palm device.

FLASHPRO

FlashPro is a TRG product that installs and runs on each Palm device. FlashPro provides a user menu that allows the user to move or copy any program or database between Flash and RAM in either direction. Users can increase the available memory, but more importantly they can backup their applications to Flash and can backup their data to Flash any time they choose via an explicit FlashPro user action on their Palm device.

In case of a hard reset requirement, FlashPro provides functions to recover to RAM the programs and the latest database backups that are in Flash.

While this FlashPro functionality provides a manual means to back up data and programs, it is not sufficient for most enterprise applications, because IT support often does NOT want the power of FlashPro in their user's hands. Hence the requirement for FlashPro APIs and other Flash solutions.

FLASHPRO API

FlashPro is a single user program, but it is also a ***platform*** for application developers. FlashPro APIs allow programmers to read from Flash and write to Flash whenever it makes sense in their application scenario. For example, in a medical application for doctors and nurses, at the end of each patient session, the Palm application program could call a FlashPro API to take a Flash backup of the patient data recorded and the prescriptions entered.

Application developers can implement Flash data safety in their proprietary palm application programs by using the FlashPro API functions. The APIs are provided to developers at no charge. Each Palm device that involves a FlashPro API must have FlashPro installed.

FLASHPACK

FlashPack V.2 is a TRG commercial program that is written using the FlashPro APIs, and provides each user with an option to backup any database to non-volatile Flash on a scheduled basis, such as 8:00 daily, or weekly. Databases such as calendar, address book, or memo pad or OEM databases work perfectly with this data safety mechanism. Users can also explicitly backup up any database whenever they want.

In case of a hard reset, FlashPack V.2 asks the user if he wants to restore the databases backed up in Flash to RAM.

CUSTOMIZING THE TRGPRO

OVERVIEW

TRG supports customers who require a customized version of the TRGpro computer through a comprehensive design and engineering support staff. TRG engineers have worked on dozens of product development projects for clients around the world. Custom solutions based on the TRGpro can range from simple modifications to the standard product to fully customized hand

held computers designed to meet specific customer requirements. Our broad hardware and software experience enables us to bring a unique perspective to each client's needs.

TRG is interested in bringing a new level of commitment to enterprise clients. By enabling customers to pick and choose from a set of well-defined modifications to the base product, a semi-custom solution can be easily formulated. Semi-custom solutions enable enterprise clients to deploy TRGpro computers to their workforces in exactly the format that is best for them.

MODIFICATIONS TO THE STANDARD TRGPRO

Modifications to the standard TRGpro product generally take the form of minor alterations to the existing product. The list below illustrates the "standard" modifications that can be applied to the product. The modifications have a standard cost, lead time, and minimum order quantity associated with them, and these parameters are ranked on a scale of 1-10. This list of available modifications will change with time. Check the TRG web site (www.trgpro.com) to see what is currently available. Enterprise customers will help us to further define a set of desirable modifications.

Modification	Description	Lead Time	Minimum Order Quantity	Engineering Cost
Custom Flash memory sizing	The amount of on-board Flash memory can be sized at 2, 4, 8, 10, 16 MB	Short-Long	Small-Large	Moderate
Customer specific Flash memory loads	A customer can have TRG pre-load a custom application in the TRGpro Flash memory.	Short	Moderate	Small
Delete premium sound	Removes the premium audio amplifier, filter, and speaker. Adds standard Palm speaker.	Moderate	Moderate	Zero
Customer specific logo applied to the unit	Customer supplied artwork can be added to the unit on the top right side of the unit.	Short	Small	Small
Customer specific booklet	Customer supplied booklet content can be added.	Large	Large	Moderate
Delete cradle	TRGpro units can be supplied without the cradle.	Short	Small	Zero
Delete CF connector	Removes the CF connector, and replaces CF back/door with standard back.	Short	Moderate	Zero
Delete retail packaging	TRGpro units can be supplied in bulk packaging to OEMs.	Short	Small	Zero
Change to enhanced screen	TRGpro can be delivered with enhanced LCD options	Long	Moderate	Zero

CUSTOMIZATION

Customization of the TRGpro product entails a change that is not one of the “standard” modifications that are available for the computer. Customizations require heavy engineering involvement in software and/or hardware. This type of change must be quoted on an individual basis to the client. The scope of the change will need to be discussed with TRG to determine the best method of approach.

There is really no limit to what can be changed in the TRGpro design. New housings can be developed, different hardware can be added to or deleted from the design, OS extensions and modifications can be made, power sources can be enhanced, etc. TRG’s engineering staff has many years of product design experience, and is familiar with working in a geographically diverse design environment with varied clients.

Please call TRG to discuss your particular TRGpro product customization needs.