PERSISTOR® CF2
API Reference

Revision 1.1 – Feb 2003

Contents

ABOUT THIS DOCUMENT

This API reference is meant to assist an experienced programmer who is familiar with the CF2 software architecture. A companion document entitled; "CF2 Programmer's Guide" discusses issues involved with application development.

ATA Device Drivers

ATACapacity -- Return the PC Card capacity in total sectors

Description: This function returns the drive or media capacity as well as additional information about the number of heads and sectors per track for LBA translation. It also optionally returns card or drive information table which has both generic and manufacturer specific fields. Refer to the device/media manuals for exact interpretation.

Prototype: **short ATACapacity(ATADvr iodvr, ulong *sectors, ushort *spt, ushort *heads, void **info);**

Inputs: **ATADvr** see following notes
iodvr pointer to the physical device driver
***sectors** pointer to hold the total number of LBA sectors available
***spt** pointer to hold sectors per track
***heads** pointer to hold the logical number of heads
****info** pointer to a ATA_SECTOR_SIZE buffer pointer that can accept the drive information table

Returns: Returns-zero for success or ATA error bits

Notes: pass zeros for everything except *sectors to just quickly determine capacity

ATAReadSectors -- Read logical drive sector(s)

Description: This function reads one or more 512 byte sectors from the card or disk into a memory buffer.

Prototype: **short ATAReadSectors(ATADvr iodvr, ulong sector, void *buffer, short count);**

Inputs: **ATADvr** see following notes
iodvr pointer to the physical device driver
sector first LBA sector to read
***buffer** pointer to memory that will contain read data
count number of ATA_SECTOR_SIZE sectors to read

Returns: Returns-zero for success or ATA error bits

Notes: Use multi-sector reads for best performance

ATAWriteSectors -- Write logical drive sector(s)

Description: This function writes one or more 512 byte sectors from a memory buffer onto the card or disk.

Prototype: **short ATAWriteSectors(ATADvr iodvr, ulong sector, void *buffer, short count);**

Inputs: **ATADvr see following notes**
iodvr pointer to the physical device driver
sector first LBA sector to write
***buffer** pointer to memory containing data to write
count number of ATA_SECTOR_SIZE sectors to write

Returns: Returns-zero for success or ATA error bits

Notes: Use multi-sector writes for best performance

ATA functions – Additional Notes

Inputs: **typedef short (*ATADvr)(void *);**

All of the functions in the ATA group make use of an anonymous structure pointer of type ATADvr to translate from generic ATA operations to actual device I/O. This first parameter to all of the ATA functions must contain a valid ATADvr pointer from at least one call to an XXGetDriver() function. () function.

Returns: All of the ATA functions return zero to indicate success, or a nonzero value to indicate some type of failure. The exact interpretation of the failure code varies depending on the physical device, but for the CompactFlash, the bits in the lower byte identify one or more of the following errors, and the upper byte may contain extended error codes that are not documented here, but can be found in the card or disk manufacturers ATA technical manuals.
0x80 card busy
0x40 card not ready
0x20 data request failure
0x10 extended error request failed
0x04 no media present
0x0F operation failed to complete
0x03 invalid argument

BigIDEA IDE Driver

BIAGetDriver -- Return the low level BigIDEA driver (for the ATA driver)

Description: Return the low level BigIDEA driver (for the ATA driver)

Prototype: **ATADvr BIAGetDriver(BIADEV device);**

Inputs: **device** is the enumeration of the device

Returns: Address of driver

BIAGetStatusString -- Return the current drive status

Description: Return the current drive status

Prototype: **char *BIAGetStatusString(void);**

Inputs: None

Returns: A pointer to a string representing drive status flags as a formatted 18 char string

BIAPowerUp -- Power up the BigIDEA and spin up the drive

Description: Apply power and spin-up the drive and, optionally, wait for completion.

Prototype: **bool BIAPowerUp(bool waitready);**

Inputs: **waitready** is TRUE if you want the program to wait for the drive to spin-up and FALSE otherwise.

Returns: If **waitready** == TRUE, returns TRUE if the drive spins up without a timeout, Otherwise returns FALSE.
If **waitready** == FALSE, immediately returns TRUE without care to the status.

BIAShutDown -- Turn off the drive and power down the BigIDEA

Prototype: **void BIAShutDown(void);**

Inputs: None

Returns: Nothing

BIOS Functions

BIOSHandlerAddress -- Return a BIOS handlers actual address

Description: The BIOSHandlerAddress macro invokes the _BIOSHandlerAddress function to return

Prototype: **vptr _BIOSHandlerAddress(short drvrid);**

Inputs: **drvrid** is the driver table id of the function whose address you seek.

Returns: Returns the address in the specified slot of the driver table.

Notes:
1. use the BIOSHandlerAddress macro instead of the function call
2. see PICOHandlerAddress for similar operations with PicoDOS functions
3. The drvrid argument to _BIOSHandlerAddress() is a non-obvious enumeration constant corresponding to the relative location of the target API function in the BIOS function list in the header. You're much better off using the BIOSHandlerAddress macro which lets you simply specify the name of the API function whose address you want to find.

BIOSPatchInsert -- Insert a new handler in the BIOS table

Description: The BIOSPatchInsert macro invokes the _BIOSPatchInsert function to let you patch specific BIOS API functions in the BIOS jump table.

Prototype: **vptr _BIOSPatchInsert(short drvrid, vptr newf);**

Inputs: **drvrid** is the driver table id of the function you wish to replace.
newf is a volatile pointer to the new address for the given BIOS routine.

Returns: Returns the former contents of the specified driver table slot for future reference or for "unpatching."

Notes:
1. The drvrid argument to _BIOSPatchInsert () is a non-obvious enumeration constant corresponding to the relative location of the target API function in the BIOS function list found in the header. You're much better off using the BIOSPatchInsert macro which lets you simply specify the name of the API function you want to patch.
2. The nature of the BIOS jump table is described in the CF2 Programmers Manual.

BIOSReset -- Reset the Persistor

Description: This function forces a hardware reset which includes assertion of the external /RESET signal. On completion, the CF2 will take whatever reset action has been ordered by the PBM boot command. If the reset action goes beyond entering PBM, the BIOS will be completely re-initialized. If PicoDOS is invoked, PicoDOS will also be completely re-initialized.

This is the cleanest way to terminate a running application if the BIOS or PicoDOS vectors have been altered or you want to guarantee the state of the hardware for the next program run.

Prototype: **void BIOSReset(void);**

BIOSResetToPicoDOS -- Reset the Persistor and force to PicoDOS

Description: This function resets the Persistor as described for the BIOSReset() function above, but forces the CF2 to jump to 0xE10000 (PicoDOS) regardless of the boot settings.

Prototype: **void BIOSResetToPicoDOS (void);**

BIOSVersionCheck -- Confirm application and BIOS compatability

Description: This function exists to help programs determine at runtime if they are compatible with the currently installed version of the BIOS. By calling this function with the version information from the initial build of the software, a developer can make a runtime determination of the ability or lack thereof of their program to run on that specific Persistor.

Prototype: **bool BIOSVersionCheck(short ver, short rel, char *id, bool reset);**

Inputs:
ver is the major release number of the BIOS at build time
rel is the minor release number of the BIOS at build time
id
reset nnn

Returns: Returns TRUE if the parameters supplied match the currently installed version of the BIOS, FALSE otherwise.

Checksums and Cyclic Redundancy Check Functions

Summary of functions	
Checksum vs. CRC	Checksums are simple and fast. CRCs provide better error detection but are slower
16 bit vs. 32 bit	Because the 68332 has a 32 bit CPU, there is virtually no performance penalty associated with using the full 32 bit routines. 16 bit CRC catches 99.998% of all errors and is appropriate for data blocks up to 4KB. 32 bit CRC catches 99.999999977% of all errors and is appropriate for blocks up to 64KB.
CheckSum16 CheckSum32	Use these to compute "on-the-fly" checksums for short data streams (like UART characters). Use checksums where speed and function pointer access are the paramount objectives.
CheckSum16Block CheckSum32Block	Use these to compute checksums for small data blocks (like the flash). Use checksums where speed and function pointer access are the paramount objectives.
CRC16	Use this routine to compute "on-the-fly" CRCs for data streams of 4kB or less.
CRC32	Use this routine to compute "on-the-fly" CRCs for data streams of 64kB or less.
CRC16Block	Use this routine to compute CRCs for data blocks of 4kB or less.
CRC32Block	Use this routine to compute CRCs for data blocks of 64kB or less.

CheckSum16 -- Update a running 16 bit checksum

Description: Computes and returns an updated unsigned short checksum derived from an unsigned byte value and an unsigned short running checksum. The running checksum is typically zero for the first call, and the latest returned value for subsequent calls. The algorithm uses simple addition primitives and has deterministic timing.

Prototype: **ushort CheckSum16(uchar value, ushort runningSum);**

Inputs: **value** is the next byte to checksum
runningSum is the running checksum from a previous call, usually initialized to zero for the first call

Returns: the updated checksum, either the final value, or the next value to pass as the running sum

Notes: Called automatically at BIOS startup

CheckSum16Block -- Compute a 16 bit checksum for a block of data

Description: Computes and returns an unsigned short checksum on a block of memory. Pass it a pointer to the start of the block, the number of bytes to compute, and a starting checksum value (typically zero). The algorithm uses simple addition primitives and has deterministic timing.

Prototype: **ushort CheckSum16Block(const void *data, ulong len, ushort runningSum);**

Inputs: **data** points to the start of the data block to CRC
len is the count in bytes to CRC
runningSum is the running checksum from a previous call, usually initialized to zero for the first call

Returns: the computed checksum

Notes: Called automatically at BIOS startup.

CheckSum32 -- Update a running 32 bit checksum

Description: Computes and returns an updated unsigned long checksum derived from an unsigned byte value and an unsigned long running checksum. The running checksum is typically zero for the first call, and the latest returned value for subsequent calls. The algorithm uses simple addition primitives and has deterministic timing as shown below.

Prototype: **ulong CheckSum32(uchar value, ulong runningSum);**

Inputs: **value** is the next byte to checksum
runningSum is the running checksum from a previous call, usually initialized to zero for the first call

Returns: the updated checksum, either the final value, or the next value to pass as the running sum

Notes: Called automatically at BIOS startup

CheckSum32Block -- Compute a 32 bit checksum for a block of data

Description: Computes and returns an unsigned short checksum on a block of memory. Pass it a pointer to the start of the block, the number of bytes to compute, and a starting checksum value (typically zero). The algorithm uses simple addition primitives and has deterministic timing.

Prototype: **ushort CheckSum16Block(const void *data, ulong len, ushort runningSum);**

Inputs: **data** points to the start of the data block to CRC
len is the count in bytes to CRC
runningSum is the running checksum from a previous call, usually initialized to zero for the first call

Returns: the computed checksum

Notes: Called automatically at BIOS startup

CRC16 -- Update a running 16 bit CCITT CRC

Description: Computes and returns an updated unsigned short cyclic redundancy check derived from an unsigned byte value and an unsigned short running CRC. The running CRC is typically zero for the first call, and the latest returned value for subsequent calls. The algorithm is table driven and has deterministic timing.

Prototype: **ushort CRC16(uchar value, ushort runningCRC);**

Inputs: **value** is the next byte to CRC
runningCRC is the running CRC from a previous call, usually initialized to zero for the first call

Returns: the updated CRC, either the final value, or the next value to pass as the running CRC

Notes: Called automatically at BIOS startup

CRC16Block -- Compute a 16 bit CCITT CRC for a block of data

Description: Computes and returns an unsigned short cyclic redundancy check on a block of memory. Pass it a pointer to the start of the block, the number of bytes to compute, and a starting CRC value (typically zero). The algorithm is table driven and has deterministic timing. This is called automatically at BIOS startup.

Prototype: **ushort CRC16Block(const void *data, ulong len, ushort runningCRC);**

Inputs: **data** points to the start of the data block to CRC
len is the count in bytes to CRC
runningCRC is the running CRC from a previous call, usually initialized to zero for the first call

Returns: the computed CRC

CRC32 -- Update a running 32 bit CCITT CRC

Description: Computes and returns an updated unsigned short long cyclic redundancy check derived from an unsigned byte value and an unsigned short running CRC. The running CRC is typically zero for the first call, and the latest returned value for subsequent calls. The algorithm is table drive and has deterministic timing as shown below. This is called automatically at BIOS startup.

Prototype: **ulong CRC32(uchar value, ulong runningCRC);**

Inputs: **value** is the next byte to CRC
runningCRC is the running CRC from a previous call, usually initialized to zero for the first call

Returns: the updated CRC, either the final value, or the next value to pass as the running CRC

CRC32Block -- Compute a 32 bit CCITT CRC for a block of data

Description: This function computes and returns an unsigned long cyclic redundancy check on a block of memory. Pass it a pointer to the start of the block, the number of bytes to compute, and a starting CRC value (typically zero). The algorithm is table drive and has deterministic timing as shown below. This is called automatically at BIOS startup.

Prototype: **ulong CRC32Block(const void *data, ulong len, ulong runningCRC);**

Inputs: **data** points to the start of the data block to CRC
len is the count in bytes to CRC
runningCRC is the running CRC from a previous call, usually initialized to zero for the first call

Returns: the computed CRC

CRCInit -- Initialize the CRC tables

Description: The 16 and 32 bit CRC routines work from table lookup algorithms rather than performing full computations for each request. These tables are setup by this routine, which is automatically called as part of the drivers initialization performed during the BIOS initialization. You will never need to explicitly call this function, and its description is included here so that when you see this function listed in the function table, you won't wonder if this is something you need to do. Calling this function more than once has no effect. This is called automatically at BIOS startup.

Prototype: **void CRCInit(void);**

Chip Select Wrapper Functions

CS10isEClock -- Define CS10 for its alternate ECLOCK function (default is chip select)

Description: When in ECLK mode, CS10 simply outputs a clock at one eighth the frequency of the system clock. This function allows the user to switch between the two functions of the CS10 pin.

Prototype: **void CS10isEClock(bool isECLK);**

Inputs: **isECLK** is a boolean that is TRUE if CS10 is to be used as ECLOCK

Notes: Persistor recommends that new users do not use CS10 as an ECLK output if sandwich cards will be used.

CS10Options -- Define the CS10 R/W access and wait states

Description: This function is called after calling CS10Setup to specify the characteristics of the device at CS10. This function tells the system whether the device can read, write or both, whether the chip select should go low with the address strobe or the data strobe and how many wait states to use during transactions.

Prototype: **void CS10Options(bool canRead, bool canWrite, bool dsSync, short waits);**

Inputs: **canRead** is a boolean that is TRUE if the device can be read from
canWrite is a boolean that is TRUE if the device can be written to
dsSync specifies whether the chip select should go low with the address strobe or the data strobe.
waits is the number of wait states to use with this device.

CS10Setup -- Setup CS10

Description: Setup CS10 address range and access width.

Prototype: **void CS10Setup(ulong baseAddr, long size, bool is16bit);**

Inputs: **baseAddr** is the address where you would like your memory mapped peripheral to reside.
size is the amount of address space above the baseAddr that you would like to reserve for the specified device.
is16bit is a boolean that is TRUE if the device is 16 bits wide.

CS10GetWaits -- Return the wait states setting for CS10

Description: Returns the number of wait states for CS10 at the current clock setting.

Prototype: **short CS10GetWaits(void);**

Inputs: None

Returns: Number of wait states

CS8Options -- Define the CS8 R/W access and wait states

Description: This function is called after calling CS8Setup to specify the characteristics of the device at CS8. This function tells the system wither the device can read, write or both, whether the chip select should go low with the address strobe or the data strobe, and how many wait states to use during transactions.

Prototype: **void CS8Options(bool canRead, bool canWrite, bool dsSync, short waits);**

Inputs: **canRead** is a boolean that is TRUE if the device can be read from
canWrite is a boolean that is TRUE if the device can be written to
dsSync specifies whether the chip select should go low with the address strobe or the data strobe.
waits is the number of wait states to use with this device.

CS8Setup -- Setup CS8

Description: Setup CS8 address range and access width.

Prototype: **void CS8Setup(ulong baseAddr, long size, bool is16bit);**

Inputs: **baseAddr** is the address where you would like your memory mapped peripheral to reside.
size is the amount of address space above the baseAddr that you would like to reserve for the specified device.
is16bit is a boolean that is TRUE if the device is 16 bits wide.

CS8GetWaits -- Return CS8 wait states setting for CS8

Description: Return CS8 wait states for CS8 at the current clock .

Prototype: **short CS8GetWaits(void);**

Inputs: None

Returns: Number of wait states

CompactFlash Low Level Drivers

CFCardDetect -- Return true if a card is inserted

Description: This function returns TRUE if a memory card is inserted in the CompactFlash header.

Prototype: **bool CFCardDetect(void);**

Returns: TRUE if a card is in the CompactFlash header, FALSE if it is not

Notes: this is simply an electro-mechanical check and does not interrogate the card to determine its viability

CFEnable -- Enable or disable the CompactFlash card to save power

Description: Turns the CompactFlash card on and off

Prototype: **void CFEnable(bool on);**

Notes:

CFGetDriver -- Return the low level CompactFlash driver (for the ATA driver)

Description: The function returns an anonymous pointer that is used by the ATA driver to access the actual CompactFlash I/O primitives. Refer to the ATA section for more information..

Prototype: **void *CFGetDriver(void);**

Notes:

Console I/O Functions and Macros

Summary of Console I/O Functions and Macros The Console I/O functions provide patchable low level access to your routines that interact with a user through a console interface. By default, all of these functions work through the 68332 SCI (Serial Controller Interface), and all the PicoDOS command shell functions work through CIO functions, as do the default standard library console functions.

CIOdrain	Wait for all transmissions to complete
CIOgetc	Wait for, and return the next character
CIOgetq	Return non-zero count if input data is available
CIOgets	Input line with minimal editing features
CIOhexdump	Dump memory in hex and ASCII to the console
CIOiflush	Flush any pending input data
CIOoflush	Discard any queued transmit characters
CIOprintf	Simple printf to console
CIOputc	Send a character
CIOputs	Send zero terminated string
getch	Wait for and return the next byte
kbflush	Empty the input buffer and return
kbhit	Detect the availability of a character on the UART
putch	Writes a byte out the main UART
putflush	Wait for all transmission to complete
putstr	Write a NULL terminated string to the main UART
uprintf	A clone of stdio's printf without floating point support

Below are macros defined in <_cfx_console.h> along with their derivations.
Red items are documented in this Console section
Blue items are documented in the SCI section

MACRO	MAPS to MACRO	MAPS TO FUNCTION	USES BY DEFAULT
kbhit()	cgetq()	CIOgetq()	SCIRxQueuedCount()
getch()	cgetc()	CIOgetc()	SCIRxGetChar()
kbflush()	ciflush()	CIOiflush()	(SCIRxGetCharWaitIdle(1) !=
putch(c)	cputc(c)	CIOputc(c)	SCITxPutChar(c)
putflush()	cdrain()	CIOdrain()	SCITxWaitCompletion()
	coflush()	CIOoflush()	SCITxFlush()
	cgetclp(x)	CIOgetclp(x)	SCIRxGetChar() + low power
	cgetclp(x)	CIOgetclp(x)	SCIRxGetChar() + low power
getstr(s,n)	cgets(s,n)	CIOgets(s,n)	uses CIOgetc
putstr(s)	cputs(s)	CIOputs(s)	uses CIOputc with CR-LF
uprintf	cprintf	CIOprintf	uses CIOputc with CR-LF
	csprintf	CIOsprintf	uses no I/O
	cvprintf	CIOvprintf	uses CIOputc with CR-LF
hexdump	chexdump	CIOhexdump	uses CIOputc/CIOgetq
	cstructdump(st)	chexdump((void*)&st,(ulong)&st,1,sizeof(st))	
	carraydump(ar)	chexdump((void*)ar,(ulong)&ar,sizeof(*ar),sizeof(ar))	

CIOdrain – Wait for all transmissions to complete

Description: Forces and waits for all characters in the transmit buffer to be transmitted.

Prototype: **void CIOdrain(void);**

Inputs: Nothing

Returns: Nothing

CIOgetc – Wait for, and return the next character

Description: Wait for the next input character.

Prototype: **short CIOgetc(void);**

Inputs: None

Returns: The character received.

CIOgetq – Return non-zero count if input data is available

Description: Returns the number of characters or zero if no input characters are waiting.

Prototype: **short CIOgetq(void);**

Inputs: None

Returns: Number of characters available

CIOgets – Input line with minimal editing features

Description: Get a line of input.

Prototype: **short CIOgets(char *buf, short len);**

Inputs: ***buf** is a pointer to destination storage for characters
len is the maximum length for the storage

Returns: The number of characters read

CIOhexdump – Dump memory in hex and ASCII to the console

Description: Provided to enable display of hex data to the console for debugging or whatever the need.

Prototype: **bool CIOhexdump(void *mem, ulong addr, short wsize, long bytecount);**

Inputs: ***mem** is a pointer to the start address to dump
addr is the start address to display
wsize is the display format 1 for byte, 2 for word, and 4 for long word
bytecount is the total number of bytes to display

Returns: Returns TRUE only if the display was interrupted by detection of a console input character (which is gobbled with CIOgetc).

CIOiflush – Flush any pending input data

Description: Flushes the input buffer of any data.

Prototype: **short CIOiflush(void);**

Inputs: None

Returns: Returns TRUE if bytes were flushed as a result of the call, FALSE if the queue was already empty and no data was flushed.

CIOoflush – Discard any queued transmit characters

Description: Flushes the transmit buffer.

Prototype: **void CIOoflush(void);**

Inputs: None

Returns: Nothing

CIOprintf – Simple printf to console

Description: A simple printf to the console but without any floating point support.

Prototype: **short CIOprintf(const char *format, ...);**

Inputs: Args like printf

Returns: The number of characters written or negative if there was an error

CIOputc – Send a character

Description: Send a character to the console with no '\n' to '\r'-'\n' translation.

Prototype: **void CIOputc(char c);**

Inputs: The character to send

Returns: Nothing

CIOputs – Send zero terminated string

Description: Send a null (zero) terminated string to the console with no '\n' to '\r'-'\n' translation.

Prototype: **void CIOputs(const char *str);**

Inputs: ***str** is a pointer to a null terminated string

Returns: Nothing

getch -- Wait for and return the next byte

Description: This function will wait for and retrieve the next incoming byte on the main UART.

Prototype: **short getch(void);**

Returns: Returns a short integer (16 bits) instead of a char (8 bits) as you might expect. The character fetched is always in the LSB(low 8 bits) of the short.

The CIO functions (on which the CIO related macros are based) explicitly mask off the high byte to appear the same as early versions.

Notes: **getch() is a macro to SCIRxGetChar()**
See STDIOWarning at end of this section

kbflush -- Empty the input buffer and return

Description: kbflush() will empty the receive buffer and return immediately regardless of buffer contents or buffering mode.

Prototype: **void kbflush(void);**

Notes: kbflush() is a macro to (SCIRxGetCharWaitIdle(1) != 1)
See STDIOWarning at end of this section

kbhit -- Detect the availability of a character on the UART

Description: This function returns nonzero if there is a character available to receive on the main serial port. It's behavior depends also on the buffering mode currently in effect. In interrupt-driven mode, kbhit() should return the number of characters available. In polled mode, kbhit() will return either 0 or non zero if there is a character available.

Prototype: **short kbhit(void);**

Returns: Returns TRUE if there is a character available, FALSE if there is not. (see notes)

Notes: kbhit() is a macro to SCIRxQueuedCount()
See STDIO Warning at end of this section

putch -- Writes a byte out the main UART

Description: This command takes the LSB of **data** and puts it in the transmit queue of the main UART. If the SCI is in polled mode, putch() will wait for the transmit queue to be empty and then write the LSB of data to the main UART. The MSB of **data** is always ignored.

Prototype: **void putch(ushort data);**

Inputs: **data** is a ushort, the LSB of which is the character you wish to transmit.

Notes: putch(c) is a macro to SCITxPutChar(c)
See STDIO Warning at end of this section

putflush -- Wait for all transmission to complete.

Description: This will wait for the transmit buffer to be empty and then return. Behavior is the same regardless of buffering mode, however timing may vary.

Prototype: **void putflush (void);**

Notes: putflush() is a macro to SCITxWaitCompletion()
See STDIO Warning at end of this section

putstr -- Write a NULL terminated string to the main UART

Description: Writes the NULL terminated string pointed to by **str** to the transmit queue of the main UART. putstr() will not append any characters to the end of any string passed to it, however it will replace any "newline" characters (/n) already in the string with CRLF sequences before inserting it into the buffer.

Prototype: **void putstr (const char *str);**

Inputs: **str** is a pointer to a NULL terminated string.

Notes: putstr(s) is a macro to SCITxPutStr(s)
See STDIO Warning at end of this section

uprintf -- A clone of stdio's printf without floating point support

Description: Provides the same functionality as the ANSI C library function printf() with respect to the main UART, excepting floating point number display support. The types float, double and long long, cannot be output by uprintf(). Full documentation of the printf() function and its many format specifiers and conventions is beyond the scope of this manual, please see an ANSI function reference for a more detailed description of printf() usage.

Prototype: **void uprintf (const char *format, ...);**

Inputs: See ANSI C documentation of printf function arguments and behavior.

Notes: uprintf() is a macro to SCITxPrintf ()
See STDIO Warning at end of this section

STDIO Warning

The ANSI C I/O libraries provided with the CF2 are, for the most part, even higher level wrappers to these driver functions. However, in an effort of duplicate exactly the behavior of the ANSI stdio system, the stdio functions often provide software buffering and queues. While you should feel free to use and inter-mingle both these driver level functions and stdio functions, it should be noted that due to the software buffering used by the stdio library, the outcome of intermingling these two groups of functions can yield unexpected results. If you, as a developer, choose to use both libraries, you should be careful to flush the buffers of each subsystem (driver calls and stdio) before using the other. Not doing so will not cause any fatal problems, but can cause confusion. For instance, if you were using the stdio printf function, and then immediately followed it with a driver call that also output data to the serial port, it is likely that the data output by the driver call would appear on the serial port before the data printed with stdio printf.

DOS Directory Functions

DIRFindFirst -- Find first directory entry starting at path

Description: This is called first when trying to get directory information.

Prototype: **short DIRFindFirst(char *path, struct dirent *dp);**

Inputs: **path** is a pointer to the path from which to start the search
dp is a pointer to a directory entry structure

Returns: An error code which could be dirErrorStart = DIR_ERRORS or dsdEndOfDir (end of directory reached)

DIRFreeSpace -- Return free space on specified drive

Description: Return free space on specified drive

Prototype: **long DIRFreeSpace(char *drive);**

Inputs: **drive** is, for example, "c:\\"

Returns: The number of free bytes as a long

DIRFindEnd -- Conclude directory search

Description: Conclude directory search

Prototype: **short DIRFindEnd(struct dirent *dp);**

Inputs: **dp** is a pointer to a directory entry structure

Returns: An error code which could be dirErrorStart = DIR_ERRORS or dsdEndOfDir (end of directory reached)

DIRFindNext -- Find next directory entry

Description: Called after an initial call to DIRFindFirst using the same pointer to directory entry. dp is filled with the information from the next directory entry.

Prototype: **short DIRFindNext(struct dirent *dp);**

Inputs: **dp** is a pointer to a directory entry structure

Returns: An error code which could be dirErrorStart = DIR_ERRORS or dsdEndOfDir (end of directory reached)

Notes: Called after an initial call to DIRFindFirst using the same pointer to directory entry. dp is filled with the information from the next directory entry.

DIRMatchName -- Return true if filename matches ambiguous pattern

Description: Return TRUE if filename matches ambiguous pattern

Prototype: **bool DIRMatchName(char *filename, char *pattern);**

Inputs: **filename** the filename to compare
pattern is a pattern like "*.txt" or "*.*" or an exact filename match. This is usually used in conjunction with DIRFindFirst and DIRFindNext comparing a pattern against the returned name from the DIRENT structure (d_name).

Returns: TRUE is the pattern matches the filename and FALSE otherwise

DIRTotalSpace -- Return total space on specified drive

Description: Return total space on specified drive

Prototype: **long DIRTotalSpace(char *drive);**

Inputs: **drive** is, for example, "c:\\"

Returns: The total number of bytes as a long

Interrupt and Exception Vector Wrapper Functions

IEV_C_FUNCT -- Define C Interrupt Handler Function

Description: This is a macro function provided for declaring a function that you wish to install as an interrupt or exception handling routine

IEV_C_PROTO -- Declare C Interrupt Handler Function Prototype

Description: This is a macro provided for prototyping a function that you wish to install as an exception or interrupt service routine written in C.

Example:

```
IEV_C_PROTO(level7InterruptISR);
IEV_C_FUNCT(level7InterruptISR)
{
   // your ISR code written in C...
}
```

IEVInsertAsmFunct -- Install an assembler function into the VBR

Description: This function allows you to install an interrupt or exception handler written in 68332 assembly language into the vector table. This varies from the IEVInsertCFunct in that it assumes that the calling and return conventions for interrupt and exception handling have already been used when the function was written. It is not necessary to use the C handler definition macros (IEV_C_PROTO and IEV_C_FUNCT) when creating an assembly function as an interrupt or exception handler. It is assumed that you know how to write these handlers when using this function. You would also use this function to reinstall a previously de-installed interrupt or exception handler whose pointer was returned by a previous call to IEVInsertCFunct or IEVInsertAsmFunct call.

Prototype: **vfptr IEVInsertAsmFunct(vfptr afp, short vector);**

Inputs: **afp** is the pointer to the assembly routine you wish to install.
vector is the vector table "slot number" you wish to install this ISR into. (see the CPU32 documentation for more info on the vector table)

Returns: Returns a pointer to the function that was previously installed in the target slot so that you can restore it later if you so desire.

IEVInsertCFunct -- Install a C function into the vector table

Description: This function allows you to install an interrupt or exception handler written in C into the vector table. You must first prototype and declare the function using the IEV_C_PROTO and IEV_C_FUNCT macros. Normally a C compiler will return from a function using a 68000 RTS instruction, but it an interrupt or exception handler must return with an RTE instruction to avoid causing havoc with the registers and the stack. This is provided for with the aforementioned macros provided.

Prototype: **vfptr IEVInsertCFunct(IEVCWrapper *cfp, short vector);**

Inputs: **cfp** is the name of the function you wish to install. It mast have been prototyped an declared using the IEV_C_PROTO and IEV_C_FUNCT macros.
vector is the vector table "slot number" you wish to install this ISR into. (see the CPU32 documentation for more info on the vector table)

Returns: Returns a pointer to the function that was previously installed in the target slot so that you can restore it later if you so desire.

LED Signal Functions

LEDGetState -- Return the current LED state

Description: Return the current LED state

Prototype: **ushort LEDGetState(ushort lrLED);**

Inputs: LEDleft or LEDright

Returns: 0 if off, 1 if red, 2 if green

Notes: Left and Right LED is established by looking at the front of the CF2 (directly at the CompactFlash card).

LEDInit -- Setup the LED drivers (turns both off)

Description: This function is called at startup and sets up the hardware drivers for the on board LEDs. A users application would not need to call this function. This function also has the default effect of turning off all of the LEDs.

Prototype: **void LEDInit(void);**

LEDOrbit -- Orbit the LEDs on each call

Description: LEDOrbit allows you to create a circling effect with the two dual LED's on the CF2. Each LED is actually two stacked LEDs with a red one on the bottom and a green one on the top. LEDOrbit when called repeatedly with a delay in between each call makes the LEDs flash in a circle. In other words each call to LEDOrbit turns off the currently active diode and turns on the next one in the "circle."

Prototype: **void LEDOrbit(bool ccw);**

Inputs: **ccw** is a boolean that is TRUE if the LEDs should orbit in the counterclockwise direction.

LEDSetState -- Set the LED state

Description: Set the state of either LED explicitly.

Prototype: **void LEDSetState(ushort lrLED, ushort state);**

Inputs: **lrLED** is either LEDleft or LEDright (enumerated as 0 and 1 respectively)
state is one of these four enumerated values: LEDoff, LEDred, LEDgreen, and LEDbus

Notes: The LED state can also be determined using a group of enumerated variables: LEDoff, LEDred, LEDgreen.

LEDToggleRG -- Toggle LED between Red and Green

Description: This function toggles the color of the specified LED from red to green or vice versa. If the LED is off when LEDToggleRG is called it will turn on with a color of red.

Prototype: **void LEDToggleRG(ushort lrLED);**

Inputs: **lrLED** is either LEDleft or LEDright (enumerated as 0 and 1 respectively) and specifies which LED to toggle

LEDToggleRGOff -- Toggle LED between Red, Green, and Off

Description: This function toggles the color of the specified LED from red to green to off in a cyclic manner. If the LED is off when LEDToggleRGOff is called it will turn on with a color of red.

Prototype: **void LEDToggleRGOff(ushort lrLED);**

Inputs: **lrLED** is either LEDleft or LEDright (enumerated as 0 and 1 respectively) and specifies which LED to toggle

Periodic Interrupt Timer Functions

PITAddChore -- Add a periodic interrupt chore

Description: This function adds a chore to the list of chores to be executed at the interval specified with either: PITSet100usPeriod or PITSet51msPeriod.

Prototype: **bool PITAddChore(vfptr chore, ushort intReqLevel);**

Inputs: **chore** is a volatile function pointer to the chore to be installed
intReqLevel is the interrupt request level that you wish your chore to be run at. It will never be higher than the level at which the PIT was initialized.

Returns: Returns TRUE if the chore was successfully added to the list.

Notes: These chores should be kept fairly short. A good rule of thumb is to keep all of the chores to a combined maximum duration of 100µs. This chore is executed in an interrupt so the standard interrupt guidelines apply with regards to execution speed. The interrupt request level you pass to this function is the interrupt request level you wish the chore to be executed at. However, the chore will never execute at a higher priority than the PIT was initialized at.

PITInit -- Initialize the periodic interrupt timer

Description: This function is generally called automatically by the operating system before your program runs, though in special cases, you may override the OS and sequence the initialization process yourself (see Startup). This function lays all the groundwork for the Periodic Interrupt timer but does not start the timer or install any chores.

Prototype: **void PITInit(ushort intReqLevel);**

Inputs: **intReqLevel** is the interrupt request level (0-7) that the PIT will run at. (Default is 3)

PITPeriod -- Return PIT period setting in microseconds, zero if off

Description: Return PIT period setting in microseconds, zero if off

Prototype: **ulong PITPeriod(**void**);**

Inputs: Nothing

Returns: PIT period in microseconds or zero if the PIT is off

PITRemoveChore -- Remove a periodic interrupt chore (NULL vfptr for all)

Description: This function removes a chore from the PIT chore list that was perviously added by the PITAddChore function. Pass NULL (zero) for the chore parameter to remove all PIT chores.

Prototype: **bool PITRemoveChore(vfptr chore);**

Inputs: **chore** is a pointer to a function that was installed with the **PITAddChore** function.

Returns: Returns TRUE if the chore was successfully removed.

PITSet100usPeriod -- Set periodic interrupt timer period in 100us ticks

Description: This function specifies how often the PIT interrupts in units of 100 µs. Because the register that holds the period is only 8 bits, this function can only specify delays of 100µs to 25.5ms. When this function is called it not only sets the period but actually starts the timer and the chores begin.

Prototype: **void PITSet100usPeriod(uchar per100us);**

Inputs: **per100us** is the number of 100µs intervals between PIT interrupts.

PITSet51msPeriod -- Set periodic interrupt timer period in 51ms ticks

Description: This function specifies how often the PIT interrupts in units of 51 ms. This function is provided because PITSet100usPeriod can only provide delays of up to 25.5 ms. This function expands that range by allowing you to specify delays of 51ms to 13s. When this function is called, it not only sets the period but actually starts the timer and the chores begin.

Prototype: **void PITSet51msPeriod(uchar per51ms);**

Inputs: **per51ms** is the number of 51ms intervals between PIT interrupts.

PicoDOS Initialization and Coordination Functions

_PICOHandlerAddress -- Return a PICO handlers actual address

Description: Return a PICO handlers actual address

Prototype: **vptr _PICOHandlerAddress(short drvrid);**

Inputs: **drvrid** is the handler id

Returns: Returns the handlers address

_PICOPatchInsert -- Insert a new handler in the PICO table

Description: Insert a new handler in the PICO table

Prototype: **vptr _PICOPatchInsert(short drvrid, vptr newf);**

Inputs: **drvrid** is the handler id
newf is the new handler's address

Returns: Returns the address of the handler being replaced

PICOMemAllocRegister -- Give PicoDOS access to application memory

Description: Give PicoDOS access to application memory to enable advanced features

Prototype: **void PICOMemAllocRegister(Callocf *callocf, Freef *freef);**

Inputs: **callocf** is the pointer to the calloc function
freef is the pointer to the free function

Returns: Nothing

PicoZOOM Functions

PZCacheFlush -- Flush cached data to the storage media

Description: Flush cached data to the storage media

Prototype: **bool PZCacheFlush(short logdrv);**

Inputs: **logdrv** is the logical drive number ("A:" = 0, "B:" = 1, etc.)

Returns: Returns TRUE if successful.

PZCacheSetup -- Setup PicoZOOM cache and optimizations

Description: Setup PicoZOOM cache and optimizations

Prototype: **bool PZCacheSetup(short logdrv, Callocf *callocf, Freef *freef);**

Inputs: **logdrv** is the logical drive number ("A:" = 0, "B:" = 1, etc.)
callocf is a pointer to a user supplied memory allocation funcion with the same behavior as the standard C library calloc funtions, and will in fact generally be a pointer to your applications calloc.
freef isa pointer to a user supplied memory allocation funcion with the same behavior as the standard C library free funtions, and will in fact generally be a pointer to your applications free.

Returns: Returns TRUE if successful.

Notes: Because PicoZOOM requires more RAM than PicoDOS has at its disposal, you must explicitly enable PicoZOOM in your application to let PicoZOOM borrow RAM from your application's heap. Each device that you setup to use PicoZOOM will require about 10KB of RAM.

PZCacheRelease -- Conclude (flush) and free PicoZOOM cache memory

Description: Conclude (flush) and free PicoZOOM cache memory

Prototype: **bool PZCacheRelease(short logdrv);**

Inputs: **logdrv** is the logical drive number ("A:" = 0, "B:" = 1, etc.)

Returns: Returns TRUE if successful.

Pin I/O Drivers, Functions, and Macros

CF2 I/O pins

SIGNAL	PIN	DESCRIPTION	DIRECTION	FUNCTION	PULL UP	RESET 1 *	RESET 2 *
DS	1	Data Strobe/GPIO	Out	GPIO/BUS		OB	OB
PCS2	15	SPI Chip Select 2	I/O	GPIO/QSPI		I?	O+
SCK	16	SPI Serial Clock	I/O	GPIO/QSPI		I?	O+
PCS3	17	SPI Chip Select 3	I/O	GPIO/QSPI		I?	O+
MOSI	18	SPI Master Data Out	I/O	GPIO/QSPI		I?	O+
PCS1	19	SPI Chip Select 1	I/O	GPIO/QSPI		I?	O+
MISO	20	SPI Master Data In	I/O	GPIO/QSPI	1M	I+	I+
PCS0	21	SPI Chip Select 0	I/O	GPIO/QSPI		I?	O+
TPU1	22	Time Processor Pin	I/O	GPIO/TMR		I?	I?
TPU2	23	Time Processor Pin	I/O	GPIO/TMR		I?	I?
TPU3	24	Time Processor Pin	I/O	GPIO/TMR		I?	I?
TPU4	25	Time Processor Pin	I/O	GPIO/TMR		I?	I?
TPU5	26	Time Processor Pin	I/O	GPIO/TMR		I?	I?
TPU6	27	Time Processor Pin	I/O	GPIO/TMR		I?	I?
TPU7	28	Time Processor Pin	I/O	GPIO/TMR		I?	I?
TPU8	29	Time Processor Pin	I/O	GPIO/TMR		I?	I?
TPU9	30	Time Processor Pin	I/O	GPIO/TMR		I?	I?
TPU10	31	Time Processor Pin	I/O	GPIO/TMR		I?	I?
TPU11	32	Time Processor Pin	I/O	GPIO/TMR		I?	I?
TPU12	33	Time Processor Pin	I/O	GPIO/TMR		I?	I?
TPU13	34	Time Processor Pin	I/O	GPIO/TMR		I?	I?
TPU14	35	Time Processor Pin	I/O	GPIO/TMR		I?	I?
T2CLK	36	Timer Load / Clock	In	GPIO/TMR	1M	I+	I+
TPU15	37	Time Processor Pin	I/O	GPIO/TMR		I?	I?
IRQ5	39	Interrupt Request 5	I/O	GPIO/IRQ	10K	IB+	I+
IRQ7	40	Interrupt Request 7	I/O	GPIO/IRQ	10K	IB+	I+
IRQ2	41	Interrupt Request 2	I/O	GPIO/IRQ	10K	IB+	I+
MODCLK	42	Clk Source Sel / GPIO	I/O	GPIO/CLK	10K	IB+	I+
IRQ4RXD	45	IRQ / CMOS RxD	Sense In	GPIO/UART		IB?	I+
TXD	46	CMOS Serial TxD	Out	GPIO/UART		I?	OB
TXX	48	CMOS TXX / RXX Output	Out	GPIO/UART	1M	I+	O+
IRQ3RXX	50	IRQ / CMOS RXX	Sense In	GPIO/UART		IB?	I+

* The state of these I/O pins shown in the column 'RESET 1' refers to the condition of the pins before PicoDOS takes control (between 500mS to 1 second). The conditions shown in 'RESET 2' are the conditions which PicoDOS sets.

I = input
O = output
OB = three state output that includes circuitry to pull up output before high impedance is established to ensure rapid rise time
IB = type OB that can operate in open drain mode.

See the CF2 Getting Started Guide for more detailed I/O pin descriptions.

PIN macros vs. PIO functions			
PIN functions	Inline assembly macros faster require compile time literal arguments do not verify pin conditions	PIO functions	Standard C functions slower can pass variables do verify pin conditions
PinBus	Make pin perform function	PIOBusFunct	Make pin perform function
PinTestIsItBus	Is pin performing its function?		
		PIOTestAssertClear PIOTestAssertSet	Return true if pin is a low output Return true if pin is a high output
PinIO	Make pin perform I/O function		
PinRead	Configure as input and read	PIORead	Configure as input and read
PinSet	Configure as output and set high	PIOSet	Configure as output and set high
PinWrite	Configure as output low or high	PIOWrite	Configure as output as low or high
PinClear	Configure as output and set low	PIOClear	Configure as output and set low
PinMirror	Read and output level read	PIOMirror PIOMirrorList	Read and output level read Read a list of pins and output level read
PinToggle	Configure as output and toggle	PIOToggle	Configure as output and toggle

PinBus -- Make an I/O pin perform its bus function (varies)

Description: Configures one of the I/O ports to act as its bus function instead of as an I/O pin.

Prototype: **void PinBus(short pin);**

Inputs: **pin** is the pin to act on.

Notes: 1) All arguments to the macro functions must be compile-time literals
2) Pin must be in Pin I/O mode or the operation may fail.

PinClear -- Configure I/O pin as output and set low

Description: Directly clears one of the I/O ports.

Prototype: **void PinClear(PinID pin);**

Inputs: **pin** is the pin to clear

Notes: 1) All arguments to the macro functions must be compile-time literals
2) Pin must be in Pin I/O mode or the operation may fail.

PinIO* -- *Make an I/O pin perform its digital I/O function

Description: Configure one of the I/O ports to act as an I/O pin.

Prototype: **void PinIO(short pin);**

Inputs: **pin** is the pin number to act on

Notes: 1) All arguments to the macro functions must be compile-time literals
2) Pin must be in Pin I/O mode or the operation may fail.

PinMirror* -- *Read an I/O pin, then configure as an output at the level read

Description: Configures an individual bit of an I/O port as an input and reads its current state, then it reconfigures the line to an output at the level previously read. This is particularly useful for eliminating floating inputs which can cause the system to waste power.

Prototype: **void PinMirror(short pin);**

Inputs: **pin** is the pin number to act on

Notes: 1) All arguments to the macro functions must be compile-time literals
2) Pin must be in Pin I/O mode or the operation may fail.

PinRead* -- *Configure I/O pin as input and read level

Description: Reads the bit setting of one of the I/O ports.

Prototype: **short PinRead(short pin);**

Inputs: **Pin** is the pin number you wish to read

Returns: The current level if the port pin has previously been defined as an input

Notes: 1) All arguments to the macro functions must be compile-time literals
2) Pin must be in Pin I/O mode or the operation may fail.

PinSet* -- *Configure I/O pin as output and set high

Description: Directly sets one of the I/O ports.

Prototype: **void PinSet(PinID pin);**

Inputs: **pin** is the pin to set.

Notes: 1) All arguments to the macro functions must be compile-time literals
2) Pin must be in Pin I/O mode or the operation may fail.

PinTestIsItBus -- Is a pin performing its bus function

Description: This function allows you program to ascertain whether a pin is currently performing its bus function. This function is only relevant to pins which have bus functions.

Prototype: **short PinTestIsItBus (short pin);**

Inputs: **pin** is the pin number to act upon.

Returns: Returns zero if the pin is configured for its I/O function and nonzero for its bus function

Notes: All arguments to the macro functions must be compile-time literals

PinToggle -- Configure I/O pin as output and toggle current level

Description: Directly toggle one of the I/O ports.

Prototype: **void PinToggle(short pin);**

Inputs: **pin** is the pin number to act upon

Notes: 1) All arguments to the macro functions must be compile-time literals
2) Pin must be in Pin I/O mode or the operation may fail.

PinWrite -- Configure I/O pin as output and write level

Description: Sets one of the I/O ports with the specified value.

Prototype: **void PinWrite(short pin, short value);**

Inputs: **pin** is the pin number to act on
value is the value (0 or 1) that you wish to write to the pin.

Notes: 1) All arguments to the macro functions must be compile-time literals
2) Pin must be in Pin I/O mode or the operation may fail.

PIOBusFunct -- Make an I/O pin perform its alternate function (varies)

Description: This function first configures an I/O line to perform its alternate function which will vary with each pin, and only applies interrupt request (IRQn) and QSM (port Q) lines. The pin argument can be specified as numerical value between 1 and 50 corresponding to the CF2 pin out on connector

Prototype: **short PIOBusFunct(short pin);**

Inputs: **pin** is the pin number to act on

Returns: Returns the current level (after mirroring) or -1 if there's an error

PIOClear -- Configure I/O pin as output and set low

Description: This function configures an individual bit of an I/O port as an output driving low. The pin argument can be specified as numerical value between 1 and 50 corresponding to the CF2 pin out on connector C.

Prototype: **short PIOClear(short pin);**

Inputs: **pin** is the pin number to act on

Returns: Returns the current level (after clearing) or -1 if there's an error

PIOMirror -- Read an I/O pin, then configure as an output at the level read

Description: This function first configures an individual bit of an I/O port as an input and reads its current state, then it reconfigures the line to an output at the level previously read. This is particularly useful for eliminating floating inputs which can cause the system to waste power. The pin argument can be specified as numerical value between 1 and 50 corresponding to the CF2 pin out on connector C.

Prototype: **short PIOMirror(short pin);**

Inputs: pin is the pin number to act on

Returns: the current level (after mirroring) or -1 if there's an error

PIOMirrorList -- Read I/O pins, then configure as an output at the level read

Description: This function acts on a zero terminated list of pin numbers and invokes PIOMirror() to convert possibly floating inputs to outputs.

Prototype: **void PIOMirrorList(uchar *pinlist);**

Inputs: pinlist is a list of uchars which contain the pin numbers to mirror.

PIORead -- Configure I/O pin as input and read level

Description: This function sets up an I/O pin as an input port and returns the current level. The pin argument can be specified as numerical value between 1 and 50 corresponding to the CF2 pin out on connector C.

Prototype: **short PIORead(short pin);**

Inputs: pin is the pin number to act on

Returns: the current level or -1 if there's an error

Notes: Notes-???

PIOSet -- Configure I/O pin as output and set high

Description: Configures an individual bit of an I/O port as an output driving high. The pin argument can be specified as numerical value between 1 and 50 corresponding to the CF2 pin out on connector C.

Prototype: **short PIOSet(short pin);**

Inputs: **pin** is the pin number to act on

Returns: Returns the current level (after setting) or -1 if there's an error

PIOTestAssertClear -- Return true if I/O pin is currently an output asserting low

Description: This function tests to see if the specified pin is configured as an output and is asserted low.

Prototype: **short PIOTestAssertClear (short pin);**

Inputs: **pin** is the pin number to test.

Returns: Returns nonzero if the pin is set low, and zero if the pin is set high or is not an I/O output.

PIOTestAssertSet -- Return true if I/O pin is currently an output asserting high

Description: Return TRUE if I/O pin is currently an output asserting high

Prototype: **short PIOTestAssertSet (short pin);**

Inputs: **pin** is the pin number to test.

Returns: Returns nonzero if the pin is set high, and zero if the pin is set low or is not an I/O output.

PIOToggle -- Configure I/O pin as output and toggle current level

Description: Configures an individual bit of an I/O port as an output driving at the opposite of the current level. The pin argument can be specified as numerical value between 1 and 50 corresponding to the CF2 pin out on connector C.

Prototype: **short PIOToggle(short pin);**

Inputs: **pin** is the pin number to act on

Returns: Returns the current level (after toggling) or -1 if there's an error

PIOWrite -- Configure I/O pin as output and write level

Description: Configures an individual bit of an I/O port as an output driving at the specified level. The pin argument can be specified as numerical value between 1 and 50 corresponding to the CF2 pin out on connector C.

Prototype: **short PIOWrite(short pin, short value);**

Inputs: **pin** is the pin number to act on
value is the value (0 or 1) that you wish to write to the pin.

Returns: Returns the current level (after setting) or -1 if there's an error

Ping-Pong Buffer Functions

Summary of Ping-Pong Buffer Functions	
PPBCheckRdAvail	Return the number of bytes waiting to be read
PPBCheckWrFree	Return the free space left before a wrap
PPBClose	Close a ping-pong buffer (does not automatically flush)
PPBFlush	Flush a ping-pong buffer and force a ping-pong flip
PPBGetMemBuf	Return the read buffer and optionally zero the size
PPBOpen	Open and initialize a ping-pong buffer
PPBPutByte	Write 8-bit byte into the ping-pong buffer
PPBPutWord	Write 16-bit word into the ping-pong buffer
PPBRead	Read data from the ping-pong buffer
PPBWrite	Write data into the ping-pong buffer

PPBCheckRdAvail -- Return the number of bytes waiting to be read

Description: Return the number of bytes waiting to be read

Prototype: **long PPBCheckRdAvail(void *ppb);**

Inputs: **ppb** is the generic pointer returned by PPBOpen and used internally to manage the buffers.

Returns: Returns the count of available bytes in the read buffer

PPBCheckWrFree -- Return the free space left before a wrap

Description: Return the free space left before a wrap

Prototype: **long PPBCheckWrFree(void *ppb);**

Inputs: **ppb** is the generic pointer returned by PPBOpen and used internally to manage the buffers.

Returns: Returns the number of bytes that can be written before the buffer ping-pongs

PPBClose -- Close a ping-pong buffer (does not automatically flush)

Description: Close a ping-pong buffer (does not automatically flush)

Prototype: **void PPBClose(void *ppb);**

Inputs: **ppb** is the generic pointer returned by PPBOpen and used internally to manage the buffers.

Returns: Returns nothing

PPBFlush -- Flush a ping-pong buffer and force a ping-pong flip

Description: Flush a ping-pong buffer and force a ping-pong flip

Prototype: **short PPBFlush(void *ppb);**

Inputs: **ppb** is the generic pointer returned by PPBOpen and used internally to manage the buffers.

Returns: Returns zero for success or a non-zero error code

PPBGetMemBuf -- Return the read buffer and optionally zero the size

Description: Return the read buffer and optionally zero the size

Prototype: **void *PPBGetMemBuf(void *ppb, long *size, bool flush);**

Inputs: **ppb** is the generic pointer returned by PPBOpen and used internally to manage the buffers.
size is a pointer to a long variable to accept the count of available bytes
flush if TRUE resets the read buffer to indicate that the data has been read

Returns: Returns a direct pointer to the read data buffer

PPBOpen -- Open and initialize a ping-pong buffer

Description: Open and initialize a ping-pong buffer

Prototype: **void *PPBOpen(long totSize, void *buf, PPBRdf rdf, PPBWrf wrf, vfptr ppnotify);**

Inputs: **totSize** is the combined size in bytes of both halves of the user supplied ping-pong buffer
buf is user supplied ping-pong buffer
rdf is an optional user supplied function that reads data from the ping-pong buffer (pass zero for default memory functions)
wrf is an optional user supplied function that writes data into the ping-pong buffer (pass zero for default memory functions)
ppnotify is an optional user supplied function to call when the ping-pong buffer wraps.

Returns: Returns a generic pointer used internally by PPB to manage the buffers or zero on failure

Notes:
```
typedef long    PPBWrf(void *buf, void *wrp, ulong wrofs, ulong n);
typedef long    PPBRdf(void *buf, void *rdp, ulong rdofs, ulong n);
```

PPBPutByte -- Write 8-bit byte into the ping-pong buffer

Description: Write 8-bit byte into the ping-pong buffer

Prototype: **short PPBPutByte(void *ppb, uchar byte);**

Inputs: **ppb** is the generic pointer returned by PPBOpen and used internally to manage the buffers.
byte is 8 bit value to write

Returns: Returns zero for success or a non-zero error code

PPBPutWord -- Write 16-bit word into the ping-pong buffer

Description: Write 16-bit word into the ping-pong buffer

Prototype: **short PPBPutWord();**

Inputs: **ppb** is the generic pointer returned by PPBOpen and used internally to manage the buffers.
word is the 16 bit value to write

Returns: Returns zero for success or a non-zero error code

PPBRead -- Read data from the ping-pong buffer

Description: Read data from the ping-pong buffer

Prototype: **long PPBRead(void *ppb, void *buf, long nbyte);**

Inputs: **ppb** is the generic pointer returned by PPBOpen and used internally to manage the buffers.
buf is a pointer to a block of data to read from the FIFO
nbyte is the number of bytes to write

Returns: Returns nbyte for complete success or the number of bytes read

PPBWrite -- Write data into the ping-pong buffer

Description: Write data into the ping-pong buffer

Prototype: **long PPBWrite(void *ppb, void *buf, long nbyte);**

Inputs: **ppb** is the generic pointer returned by PPBOpen and used internally to manage the buffers.
buf is a pointer to a block of data to write data into the FIFO
nbyte is the number of bytes to write

Returns: Returns nbyte for complete success or the number of bytes written

Power Management Drivers and Functions

Summary of Power Management Drivers and Functions	
LPStop	Execute LPSTOP with options previously setup.
LPStopCSE	Execute LPSTOP and set the passed bits to control powering down other modules.
QSMRun	Start the Queued Serial Module.
QSMStop	Stop the Queued Serial Module.
PWRLPStopSetup	Setup LPSTOP CLKOUT driven.
PWRSuspendSecs	Delay in suspend mode for a number of seconds.
PWRSuspendUntil	Delay in suspend mode until a future time.
PWRPreChgAddChore	Add a power pre-change chore.
PWRPreChgRemoveChore	Remove a power pre-change chore.
PWRPostChgAddChore	Add a power post-change chore.
PWRPostChgRemoveChore	Remove a power post-change chore.

LPStop – Executes LPSTOP with options previously setup

Description: Executes LPSTOP with options previously setup

Prototype: **void LPStop(void);**

Inputs: None

Returns: Nothing

LPStopCSE – Execute LPSTOP and set the passed bits to control powering down

Description: Execute an LPSTOP and set the passed bits to control powering down other modules.

Prototype: **void LPStopCSE(uchar csebits);**

Inputs: LPStopCSE modifies SYNCR with bits specifying:

bit 1 (0x02) = 1 is VCO running and driving SIMCLK
bit 0 (0x01) = 1 is external clock driven as determined by STSIM

You will use one of the following:

Lowest Power - FullStop
Fast IRQ Response – FastStop
Submodules Running – CPUStop

Returns: Nothing

QSMRun – Start the Queued Serial Module

Description: Start the Queued Serial Module

Prototype: **void QSMRun(void);**

Inputs: None

Returns: Nothing

Notes: macro inserts in-line code for fast operation

QSMStop – Stop the Queued Serial Module

Description: Stop the Queued Serial Module

Prototype: **void QSMStop(void);**

Inputs: None

Returns: Nothing

Notes: macro inserts in-line code for fast operation

PWRLPStopSetup – Setup LPSTOP CLKOUT driven

Description: Setup LPSTOP CLKOUT driven

Prototype: **void PWRLPStopSetup(bool stcpu, bool stsim, bool stext);**

Inputs: **stcpu** TRUE stops just cpu
stsim TRUE clock stays VCO
stext TRUE CLKOUT driven

Returns: Nothing

PWRSuspendSecs – Delay in suspend mode for a number of seconds

Description: Delay in suspend mode for a number of seconds

Prototype: **WhatWokeSuspend PWRSuspendSecs(ulong delaysecs, bool resume, short WhatWakesSuspend);**

Inputs: **delaysecs** is the delay in seconds as a long value
resume is a Boolean value where TRUE means continue execution at the line following the call and FALSE will force a RESET
WhatWakesSuspend is a flag to indicate what is allowed to wake up early (e.g. the wake pin)

Returns: An unermerated value (WhatWakesSuspend) indicating what really woke us.

PWRSuspendUntil – Delay in suspend mode until a future time

Description: Delay in suspend mode until a future time

Prototype: **WhatWokeSuspend PWRSuspendUntil(ulong waketime, bool resume, short WhatWakesSuspend);**

Inputs: **waketime** is the future wake time in seconds as a long value (based on RTC)
resume is a Boolean value where TRUE means continue execution at the line following the call and FALSE will force a RESET
WhatWakesSuspend is a flag to indicate what is allowed to wake up early (e.g. the wake pin)

Returns: An unermerated value (WhatWakesSuspend) indicating what really woke us.

PWRPreChgAddChore – Add a power pre-change chore

Description: Add a power pre-change chore

Prototype: **bool PWRPreChgAddChore(vfptr chore, ushort priority);**

Inputs: **chore** is a volatile function pointer to the chore to be executed at interrupt time
priority is the CPU priority for the chore

Returns: TRUE if success FALSE otherwise

PWRPreChgRemoveChore – Remove a power pre-change chore

Description: Remove a power pre-change chore

Prototype: **bool PWRPreChgRemoveChore(vfptr chore);**

Inputs: **chore** is a pointer to the chore

Returns: TRUE if success and FALSE otherwise.

PWRPostChgAddChore – Add a power post-change chore

Description: Add a power post-change chore

Prototype: **bool PWRPostChgAddChore(vfptr chore, ushort priority);**

Inputs: **chore** is a volatile function pointer to the chore to be executed at interrupt time
priority is the CPU priority for the chore

Returns: TRUE if success and FALSE otherwise.

PWRPostChgRemoveChore – Remove a power post-change chore

Description: Remove a power post-change chore

Prototype: **bool PWRPostChgRemoveChore(vfptr chore);**

Inputs: **chore** is a pointer to the chore

Returns: TRUE if success and FALSE otherwise.

Query/Reply Functions

The query/reply functions provide a variety of useful functions for interacting with an operator using console I/O.

Summary of Query / Reply Functions	
QRchar	Query/Reply for character using: %c
QRconfirm	Query/Reply for Y/N confirmation
QRdate	Query/Reply for date
QRdatetime	Query/Reply for date and time
QRdouble	Query/Reply for double using: %lf %le %lE %lg %lG
QRfloat	Query/Reply for float using: %f %e %E %g %G
QRlong	Query/Reply for long using: %li %ld %lu %lo
QRshort	Query/Reply for short using: %i %hi %d %hd %u %hu %o %ho
QRstring	Query/Reply for string using: %s
QRtime	Query/Reply for time
QRulong	Query/Reply for ulong using: %li %ld %lu %lo
QRushort	Query/Reply for ushort using: %i %hi %d %hd %u %hu %o %ho

QRchar -- Query/Reply for character using: %c

Description: Query/Reply for character using: %c

Prototype: **bool QRchar(char *prompt, char *fmt, bool crok, char *reply, char *instr, bool uc);**

Inputs: **prompt** is a zero terminated C string to display as a prompt for the user.
fmt is a standard C libary printf/scanf format string
crok is set TRUE to accept the default reply with just a carriage-return (enter key)
reply is a pointer to a character variable to hold the reply
instr if non-zero must contain a set of characters that the reply must match to be valid
uc is set TRUE to force all replies to upper case.

Returns: Returns TRUE for all but query cancelled (CTRL-C typed).

QRconfirm -- Query/Reply for Y/N confirmation

Description: Query/Reply for Y/N confirmation

Prototype: **bool QRconfirm(char *prompt, bool defyes, bool crok);**

Inputs: **prompt** is a zero terminated C string to display as a prompt for the user.
defyes is set TRUE to make 'Y' the default reply.
crok is set TRUE to accept the default reply with just a carriage-return (enter key)

Returns: Returns TRUE Yes replies.

QRdate -- Query/Reply for date

Description: Query/Reply for date

Prototype: **bool QRdate(char *prompt, DateFieldOrder dfo, bool crok, struct tm *tm);**

Inputs: **prompt** is a zero terminated C string to display as a prompt for the user.
crok is set TRUE to accept the default reply with just a carriage-return (enter key)
tm is a standard C library tm structure pointer from <time.h> with fields tm_year, tm_mon, and tm_mday filled in from the scan.

Returns: Returns TRUE for all but query cancelled (CTRL-C typed).

Notes: Replies are accepted with:

YEAR<delim>MONTH<delim>DAY (specifying enum YYMMDD)
MONTH<delim>DAY<delim>YEAR (specifying enum MMYYDD)
DAY<delim>MONTH<delim>YEAR (specifying enum DDMMYY)

<delim> may be any single comma, space, dash, period, or slash.
YEAR may a full 4 digits or 2 digits assumed between 1970 and 2069
MONTH may be 1-12, full text (January, February, ..., December) or three character abbreviations.

```
typedef enum {  YYMMDD   // ISO
              , MDDYY    // US
              , DDMMYY   // European
              } DateFieldOrder;
```

QRdatetime -- Query/Reply for date and time

Description: Query/Reply for date and time

Prototype: **bool QRdatetime(char *prompt, DateFieldOrder dfo, bool crok, struct tm *tm);**

Inputs: **prompt** is a zero terminated C string to display as a prompt for the user.
crok is set TRUE to accept the default reply with just a carriage-return (enter key)
tm is a standard C library tm structure pointer from <time.h> with fields tm_year, tm_mon, tm_mday, m_hour, tm_min, and tm_secs filled in from the scan.

Returns: Returns TRUE for all but query cancelled (CTRL-C typed).

Notes: Replies are accepted with:

YEAR<delim>MONTH<delim>DAY (specifying enum YYMMDD)
MONTH<delim>DAY<delim>YEAR (specifying enum MMYYDD)
DAY<delim>MONTH<delim>YEAR (specifying enum DDMMYY)

<delim> may be any single comma, space, dash, period, or slash.
YEAR may a full 4 digits or 2 digits assumed between 1970 and 2069
MONTH may be 1-12, full text (January, February, ..., December) or three character abbreviations.

```
typedef enum {  YYMMDD   // ISO
              , MDDYY    // US
              , DDMMYY   // European
              } DateFieldOrder;
```

QRdouble -- Query/Reply for double using: %lf %le %lE %lg %lG

Description: Query/Reply for double using: %lf %le %lE %lg %lG

Prototype: **bool QRdouble(char *prompt, char *fmt, bool crok, double *value, double min, double max);**

Inputs:
prompt is a zero terminated C string to display as a prompt for the user.
fmt is a standard C libary printf/scanf format string
crok is set TRUE to accept the default reply with just a carriage-return (enter key)
value is a pointer to a double variable that will hold the reply
min if not equal to max, min is the minimum valid value to accept for a reply
max if not equal to min, max is the maximum valid value to accept for a reply

Returns: Returns TRUE for all but query cancelled (CTRL-C typed).

QRfloat -- Query/Reply for float using: %f %e %E %g %G

Description: Query/Reply for float using: %f %e %E %g %G

Prototype: **bool QRfloat(char *prompt, char *fmt, bool crok, float *value, float min, float max);**

Inputs:
prompt is a zero terminated C string to display as a prompt for the user.
fmt is a standard C libary printf/scanf format string
crok is set true to accept the default reply with just a carriage-return (enter key)
value is a pointer to a float variable that will hold the reply
min if not equal to max, min is the minimum valid value to accept for a reply
max if not equal to min, max is the maximum valid value to accept for a reply

Returns: Returns TRUE for all but query cancelled (CTRL-C typed).

QRlong -- Query/Reply for long using: %li %ld %lu %lo

Description: Query/Reply for long using: %li %ld %lu %lo

Prototype: **bool QRlong(char *prompt, char *fmt, bool crok, long *value, long min, long max);**

Inputs:
prompt is a zero terminated C string to display as a prompt for the user.
fmt is a standard C libary printf/scanf format string
crok is set true to accept the default reply with just a carriage-return (enter key)
value is a pointer to a long variable that will hold the reply
min and **max** same as described above

Returns: Returns TRUE for all but query cancelled (CTRL-C typed).

QRshort -- Query/Reply for short using: %i %hi %d %hd %u %hu %o %ho

Description: Query/Reply for short using: %i %hi %d %hd %u %hu %o %ho

Prototype: **bool QRshort(char *prompt, char *fmt, bool crok, short *value, short min, short max);**

Inputs:
prompt is a zero terminated C string to display as a prompt for the user.
fmt is a standard C libary printf/scanf format string
crok is set true to accept the default reply with just a carriage-return (enter key)
value is a pointer to a short variable that will hold the reply
min and **max** same as described above

Returns: Returns TRUE for all but query cancelled (CTRL-C typed).

QRstring -- Query/Reply for string using: %s

Description: Query/Reply for string using: %s,

Prototype: **bool QRstring(char *prompt, char *fmt, bool crok, char *strbuf, short len);**

Inputs: **prompt** is a zero terminated C string to display as a prompt for the user.
fmt is a standard C libary printf/scanf format string
crok is set TRUE to accept the default reply with just a carriage-return (enter key)
strbuf is a pointer to a character buffer to hold the reply
len is the maximum number of character to place in the reply buffer

Returns: Returns TRUE for all but query cancelled (CTRL-C typed).

QRtime -- Query/Reply for time

Description: Query/Reply for time

Prototype: **bool QRtime(char *prompt, bool crok, struct tm *tm);**

Inputs: **prompt** is a zero terminated C string to display as a prompt for the user.
crok is set TRUE to accept the default reply with just a carriage-return (enter key)
tm is a standard C library tm structure pointer from <time.h> with fields tm_hour, tm_min, and **tm_secs** filled in from the scan.

Returns: Returns TRUE for all but query cancelled (CTRL-C typed).

Notes: Replies are accepted in the form: HOURS<delim>MINUTES<delim>SECONDS
<delim> may be any single comma, space, dash, period, or slash.

QRulong -- Query/Reply for ulong using: %li %ld %lu %lo

Description: Query/Reply for ulong using: %li %ld %lu %lo

Prototype: **bool QRulong(char *prompt, char *fmt, bool crok, ulong *value, ulong min, ulong max);**

Inputs: **prompt** is a zero terminated C string to display as a prompt for the user.
fmt is a standard C libary printf/scanf format string
crok is set TRUE to accept the default reply with just a carriage-return (enter key)
value is a pointer to a ulong variable that will hold the reply
min if not equal to max, min is the minimum valid value to accept for a reply
max if not equal to min, max is the maximum valid value to accept for a reply

Returns: Returns TRUE for all but query cancelled (CTRL-C typed).

QRushort -- Query/Reply for ushort using: %i %hi %d %hd %u %hu %o %ho

Description: Query/Reply for ushort using: %i %hi %d %hd %u %hu %o %ho

Prototype: **bool QRushort(char *prompt, char *fmt, bool crok, ushort *value, ushort min, ushort max);**

Inputs: **prompt** and **fmt** and **crok** same as described above
value is a pointer to a ushort variable that will hold the reply
min and **max** same as described above

Returns: Returns TRUE for all but query cancelled (CTRL-C typed).

Queued PicoBUS (QSPI) Drivers and Functions

Summary of QSPI Drivers and Functions	
QPBClearBusy	Clear busy flag
QPBClearInterrupt	Clear the QPB interrupt flag
QPBFreeSlot	Remove a PicoBUS device slot
QPBInitSlot	Initialize a PicoBUS device slot
QPBLockSlot	Lock and return true if slot available for exclusive use
QPBRepeatAsync	Execute a pre-set-up asynchronous transfer
QPBSetup	Set up the QPB for future asynchronous transfers
QPBTestBusy	Set up the QPB for future asynchronous transfers
QPBTestLocked	See if the QPB is locked
QPBTransact	Conduct a PicoBUS session
QPBUnlockSlot	Unlock PicoBus for unrestricted use

QPBClearBusy -- Clear busy flag

Description: Clear busy flag

Prototype: **void QPBClearBusy (void);**

Inputs: None

Returns: Nothing

QPBClearInterrupt -- Clear the QPB interrupt flag

Description: Clear the QPB interrupt flag

Prototype: **void QPBClearInterrupt (void);**

Inputs: None

Returns: Nothing

Notes: When operating asynchronously, the function specified by the **asynchf** argument to either QPBSetup or QPBTransact, will be called upon completion of the transaction. This function must be called by the handler function so that the interrupt flag that forced the execution of the completion routine is cleared. If the flag is not cleared, then the completion routine will be executed for ever.

QPBFreeSlot -- Remove a PicoBUS device slot

Description: Remove a PicoBUS device slot.

Prototype: **void QPBFreeSlot(QPBDev *qbpd);**

Inputs: None

Returns: Nothing

Notes: QPBFreeSlot removes a device installed previously by QPBInitSlot. It also frees all memory allocated in relation to the device's QPBDev structure.

QPBInitSlot -- Initialize a PicoBUS device slot

Description: Initialize a PicoBUS device slot.

Prototype: **QPB *QPBInitSlot(QPBDev *qbpd);**

Inputs: **qbpd** is a QPBDev structure filled in by the user that will be assigned a slot for use in the QPB.

Returns: Returns a pointer to a QPB structure that will be used when transacting with the device.

Notes: This function takes device information that you provide in the QPBDev structure and initializes one of the QPB's 14 slots for use with that specified device. It is important to bear in mind however that using all 14 slots or arbitrarily picking slot numbers requires external multiplexing hardware.

QPBLockSlot -- Lock and return true if slot available for exclusive use

Description: Lock and return TRUE if slot available for exclusive use.

Prototype: **bool QPBLockSlot(QPB *qpb);**

Inputs: **qpb** is the QPB structure that corresponds to the slot of the desired device. The device must either be unlocked or the lock must belong to the QPB struct passed here.

Returns: Returns TRUE if lock was successful FALSE otherwise

Notes: This function takes device information that you provide in the QPBDev structure and initializes one of the QPB's 14 slots for use with that specified device. It is important to bear in mind however that using all 14 slots or arbitrarily picking slot numbers requires external multiplexing hardware.

QPBRepeatAsync -- Execute a pre-set-up asynchronous transfer

Description: Execute a pre-set-up asynchronous transfer

Prototype: **void QPBRepeatAsync (void);**

Notes: This function executes a transaction on the PicoBus. The parameters of the transfer must have been first setup with QPBSetup or the call will fail and because it is an inline assembly macro with no return values or parameters, you will have little or no non-catastrophic indication of failure. The advantage of using QPBSetup with QPBRepeatAsync is that it allows the developer to reach the highest data rates on the QPB. The other calls have much overhead but also provide more functionality. Furthermore the slot you wish to use QPBRepeatAsync with must be locked first with QPBLockSlot.

QPBSetup -- Set up the QPB for future asynchronous transfers

Description: This function sets up all of the internal registers of the PicoBus and prepares for an asynchronous transaction. QPBTransact calls this function every time it is called, but in a synchronous mode.

Prototype: **bool QPBSetup(QPB *qpb, vfptr asynchf, ushort count, ushort *spidata);**

Inputs: **qpb** is the QPB structure that corresponds to the slot of the desired device. The device must either be unlocked or the lock must belong to the QPB struct passed here.
asynchf is a volatile function pointer to the handler to be called when an asynchronous read completes.
count is the number of 16 bit words that will be passed in the variable arguments. The maximum number of of words to be passed is 16, making count's maximum 16 also.
spidata is a pointer to the data for the transaction.

Returns: Returns TRUE if the requested setup could be performed.

QPBTestBusy -- Set up the QPB for future asynchronous transfers

Description: This function allows you to test whether or not a transaction (obviously an asynchronous one) is currently occurring on the QPB.

Prototype: **QPB *QPBTestBusy (void);**

Returns: Returns a pointer to the QPB structure that is currently transacting or NULL if the bus is free.

QPBTestLocked -- See if the QPB is locked

Description: This function allows you to test whether or not the QPB is currently locked.

Prototype: **QPB *QPBTestLocked (void);**

Returns: Returns a pointer to the QPB structure that is currently locking or NULL if the bus is free.

QPBTransact -- Conduct a PicoBUS session

Description: This function executes a transaction on the QPB.

Prototype: **short *QPBTransact(QPB *qpb, vfptr asynchf, ushort count, ushort *spidata);**

Inputs: **qpb** is the QPB structure that corresponds to the slot of the desired device. The device must either be unlocked or the lock must belong to the QPB struct passed here.
asynchf is a volatile function pointer to the handler to be called when an asynchronous read completes. If it is 0 then the call will be synchronous.
count is the number of 16 bit words that will be passed in the variable arguments. The maximum number of of words to be passed is 16, making count's maximum 16 also.
spidata is a pointer to the data for the transaction.

Returns: ??

Notes: This function takes data as an array of up to 16 words (which are 16 bits wide in the 332 but each word will be truncated to your device's word size before transacting. In other words if you want to send 8 words to a device with an 8 bit word length, you would put 8 shorts with padded MSBs into spidata.) The QPB structure returned by QPBInitSlot designates which device is to receive the transaction and the count is the number of 16 bit words (up to 16) to read off the stack to be transmitted. You may also make this call asynchronous by providing a pointer to a completion routine in the **asynchf** parameter.

QPBUnlockSlot -- Unlock PicoBus for unrestricted use.

Description: This function allows you to unlock the PicoBus after it has been locked by a specific slot.

Prototype: **bool QPBUnlockSlot (QPB *qpb);**

Inputs: **qpb** is a pointer to the slot structure that owns the current lock

Returns: Returns TRUE if the bus was successfully unlocked or FALSE if it was unable to unlock or the given QPB structure was not the owner of the lock..

Notes: See QPBLockSlot for more information. It is necessary to lock the bus when performing asynchronous reads using QPBRepeatAsynch.

Real Time Clock Drivers and Functions

Real Time Clock Operation

The 68332 has no onboard Real Time Clock (RTC). The RTC function is provided by a Texas Instruments MSP430 microcontroller which lives on the CF2 board with the 68332. The MSP430 is clocked by a 40 KHz crystal, and powered by either the main supply or an off-board backup battery. The RTC crystal is a tuning fork resonator with an initial accuracy of +/-20ppm with the parabolic temperature versus frequency curve typical of these types of crystals. The MSP430 feeds this same signal to the 68332 PLL for system clock generation.

Low-level software in the 68332 maintains communications with the MSP430. While the 68332 is on it maintains its own internal clock which is synchronized with the MP430. When power is removed the MSP430 maintains time and continues to run from the lithium battery (this assumes you are using a Recipe Card). When power is re-applied, the 68332 restarts and synchronizes its internal clock from the MSP430.

Elapsed and Countdown Timers

The countdown and elapsed time functions provide convenient methods for measuring short intervals or waiting for a timeout. These functions work with and return values normalized to microseconds. Time is measured with a fundamental unit of 40000 Hz.. Thus, the timer is internally counted as an integer number of these 'ticks'. One tick is approximately 25 µsecs and with a calling overhead for the functions that interact with these counters of about 20 µsecs, the timers have an effective resolution of no less than 50 µsecs. The elapsed and countdown timers have a maximum span of about 1.2 hours before overflowing. Below is an example of usage:

```
RTCTimer        tmtest;
 ...
RTCElapsedTimerSetup(&tmtest);
 < code block being measured >
printf("Elapsed time %ld us \n", RTCElapsedTime(&tmtest));
```

RTCDelayMicroSeconds -- Delay for microseconds

Description: Provides a simple means to kill a specific amount of time, specified in microseconds. This performs much the same function as RTCDelayTicks but takes its argument in microseconds. The resolution of the calculation from microseconds to ticks which the real time clock can measure is about 25 µsecs

Prototype: **bool RTCDelayMicroSeconds (ulong d);**

Inputs: **d** is a ulong containing the number of microseconds to wait before returning.

Returns: Returns a boolean that is TRUE unless the clock is stopped at the time **RTCDelayMicroSeconds** is called.

Notes: RTCDelayMicroSeconds behavior in the event that the RTC is stopped is dependent on the action specified by RTCEnableErrTrap. Normally, it will simply return FALSE if the clock is stopped but if error trapping is enabled by RTCEnableErrTrap then the machine will crash and print debug information. Because internally the real time clock can only measure time in units of ticks, the countdown is counted in an integer number of ticks. Because one tick is approximately 25 µsecs and the calling overhead of the functions that interact with the clock is around 20 µsecs, delays of less than 55 µsecs are meaningless and that RTCDelayMicroSeconds has an effective resolution of 55 µsecs.

RTCElapsedTime -- Read the elapsed time (us)

Description: Read the elapsed time in microseconds

Prototype: **ulong RTCElapsedTime(RTCTimer *rt);**

Inputs: **rt** is a pointer to an RTCTimer struct that is preallocated and ready to be filled.

Returns: Returns the elapsed time in microseconds.

Notes: Because internally the real time clock can only measure time in units of ticks, the timer is internally counted as an integer number of ticks then normalized to microseconds. Because one tick is approximately 25 µsecs and the calling overhead of the functions that interact with these counters is at least 20 µsecs, the timers have an effective resolution of no less than 55 µsecs.

RTCElapsedTimerSetup -- Setup and start an elapsed timer (us)

Description: Tells the real time clock to set up and start an ascending counter. The timer will immediately begin counting up from zero in microseconds. Once RTCElapsedTimerSetup returns, use the function RTCElapsedTime with the same RTCTimer struct to watch the counter.

Prototype: **void RTCElapsedTimerSetup(RTCTimer *rt);**

Inputs: **rt** is a pointer to an RTCTimer struct that is preallocated and ready to be filled.

Notes: See the Elapsed and Countdown Timers description at the top.

RTCGetTime -- Get both seconds and ticks

Description: The real time clock maintains two registers. One that keeps ticks (1/40000'ths of a second (for approximately 25 µsec resolution) and one that keeps seconds.

Prototype: **ulong RTCGetTime(ulong *seconds, ushort *ticks);**

Inputs: **seconds** is a pointer to a ulong into which the current seconds counter will be placed.
ticks is a pointer to a ushort into which the current ticks count will be placed.

Returns: Returns a ulong containing just the seconds count. This is useful when you do not wish to create placeholding variables.

Notes: If either or both of the pointers are NULL, the function will not write into the base of memory, it will ignore the parameter and fill the other one, or if both are NULL, it will simply return the seconds count. See RTCSetTime for UNIX epoch note.

RTCSetTime -- Set both seconds and ticks

Description: RTCSetTime loads values into the ticks and seconds registers that are provided in the argument to the function. For standard operation these values should reflect the number of seconds and the number of ticks since the UNIX epoch which is midnight Jan 1, 1970, although there is no restriction that forces this to be true. RTCSetTime can be called regardless of whether the real time clock is running or stopped. If the clock is running, RTCSetTime will set the time and keep the clock running, as could be expected. If the clock is stopped, for instance if you wish to set the clock based on an external event, RTCSetTime will load the given values into the registers and then when/if the clock is started, it will continue to count from the values loaded.

Prototype: **void RTCSetTime(ulong secs, ushort ticks);**

Inputs: **secs** is a ulong that contains the value intended for the seconds register of the Real Time Clock
ticks is a ushort that contains the value intended for the ticks register of the Real Time Clock

RTCtime -- ANSI standard C library time() equivalent function

Description: This function returns the contents of the real time clock's seconds register. In a system with a properly set clock, this will be equal to the number of seconds since midnight Jan 1, 1970. This number is copied into the ulong pointed to by **tp**.

Prototype: **ulong RTCtime(ulong *tp);**

Inputs: **tp** is a pointer to a ulong where you wish the time to be placed on return.

Returns: Returns the contents of the real time clock's seconds register.

Notes: If **tp** is NULL, the function will not fail or write to the base of memory, it will simply do nothing with the argument and return the contents of the real time clock's seconds register.

Serial Controller Interface Drivers and Functions

Summary of Serial Interface Drivers and Functions
The Serial Controller Interface is the hardware subsystem that deals with all transactions over the main serial port (UART).

SCIConfigure	Set the baud rate and parity
SCIGetConfig	Get the baud rate and parity
SCIRxBreak	Return true if break is seen for at least **millisecs**
SCIRxFlush	Delete any data in the receive queue
SCIRxGetByte	Return the next word, wait if block is true
SCIRxGetChar	Wait for, and return the next word
SCIRxGetCharWithTimeout	Return next byte from receive queue with timeout
SCIRxHandshake	Set receive flow control
SCIRxQueuedCount	Return the number of characters in the receive queue
SCIRxSetBuffered	Select buffered (true) or non-buffered receive
SCIRxTxIdle	Return true if all Rx and Tx flags indicate idle
SCITxBreak	Start (-1) , stop (0) , or send timed break (+ val)
SCITxFlush	Delete any data in the transmit queue
SCITxHandshake	Set transmit flow control
SCITxPutByte	Transmit byte, wait if block is true
SCITxPutChar	Transmit byte
SCITxQueuedCount	Return the number of words in the transmit queue
SCITxSetBuffered	Select buffered (true) or non-buffered transmit
SCITxWaitCompletion	Wait for all transmission to complete
EIAAssertTXX	Assert /TXX
EIACheckRXX	Get State of /RXX
EIAEnableRx	Enable RS232 receivers
EIAForceOff	Force RS232 transmitters off

STDIO Warning

The ANSI C I/O libraries provided with the CF2 are, for the most part, even higher level wrappers to these driver functions. However, in an effort of duplicate exactly the behavior of the ANSI stdio system, the stdio functions often provide software buffering and queues. While you should feel free to use and inter-mingle both these driver level functions and stdio functions, it should be noted that due to the software buffering used by the stdio library, the outcome of intermingling these two groups of functions can yield unexpected results. If you, as a developer, choose to use both libraries, you should be careful to flush the buffers of each subsystem (driver calls and stdio) before using the other. Not doing so will not cause any fatal problems, but can cause confusion. For instance, if you were using the stdio printf function, and then immediately followed it with a driver call that also output data to the serial port, it is likely that the data output by the driver call would appear on the serial port before the data printed with stdio printf.

Error Codes

Many of the driver calls, particularly those which deal with fetching a byte from the receive portion of the main UART, return a short integer (16 bits) instead of a char (8 bits) as you might expect. The reason for this is so that error information can be passed back with the character. The character fetched is always in the LSB(low 8 bits) of the short so that if it is automatically typecast into a char, your character will survive and only the error codes will be lost. Similarly, if a char is passed to a function that expects a short for the purpose of holding error codes in the MSB, there should not be any unexpected behavior. In the MSB (high 8 bits) are error codes that are tested using the following masks:

```
enum {
        RxD_OR_MASK         = 0x8000    // Overrun Error Flag
      , RxD_NF_MASK         = 0x4000    // Noise Error Flag
      , RxD_FE_MASK         = 0x2000    // Framing Error Flag
      , RxD_PF_MASK         = 0x1000    // Parity Error Flag
      , RxD_BOV_MASK        = 0x0800    // Buffer Overflow Flag
      , RxD_TOSS_MASK       = 0x0400    // Throw Away Flag (user sets)
      , RxD_RPTF_MASK       = 0x0200    // Repeat Filter Call Flag (user sets)
      , RxD_ERR_MASK        = 0xF800    // Any Error Flags
      , RxD_DATA9_MASK      = 0x01FF    // Receive Data, 9 Bits
      , RxD_DATA8_MASK      = 0x00FF    // Receive Data, 8 Bits
      , RxD_DATA7_MASK      = 0x007F    // Receive Data, 7 Bits (ASCII)
      , RxD_NO_DATA         = 0xFCFF    // Return value for no data available
      };
```

By AND'ing any of these masks with the short that is returned with a given function, the user's application can determine what, if any, error occurred during the operation.

_cfx_sercomm.h contains the error code enumeration.

SCIConfigure -- Set the baud rate and parity

Description: Sets the baud rate, parity, stop bits and auto-recalculation settings for the main UART. This is the primary means by which a user's application can change the serial port settings.

Prototype: **long SCIConfigure(long baud, char parity, bool autoTiming);**

Inputs: **baud** is a long integer containing the desired baud rate
parity is a character containing a code to specify the parity and stop bits settings desired. The codes are listed below.
autoTiming is a Boolean indicating whether or not you would like the SCI to recalculate the baud rate in the event of a clock speed change. TRUE means do recalculate.

Returns: Returns a copy of the argument baud.

Notes: Parity codes are as follows:
'N' = '0' = '1' = No Parity and 1 Stop Bit
'2' = No Parity and 2 Stop Bits
'E' = Even Parity and 1 Stop Bit 'O' = Odd Parity and 1 Stop Bit

SCIGetConfig -- Get the baud rate and parity

Description: SCIGetConfig is used to extract the current settings of the SCI including baud rate and parity. Parity and stop bit information is returned in the form of a character that corresponds to a code which is described below. If either of the pointers passed to SCIGetConfig are NULL, the function will not write to the base of memory, but rather ignore that parameter. It will always return the baud rate, even if **baudPtr** is NULL.

Prototype: **long SCIGetConfig(long *baudPtr, char *parityPtr);**

Inputs: **baudPtr** is a pointer to a long where SCIGetConfig should put the returned baud rate.
parityPtr is a pointer to a character where SCIGetConfig should put the code which denotes the current parity/stop bit setting.

Returns: Returns a long equal to the baud rate (equal to *baudPtr)

Notes: Parity is denoted by a character containing a code to specify the parity and stop bits settings. The codes are as follows:
'N' = '0' = '1' = No Parity and 1 Stop Bit
'2' = No Parity and 2 Stop Bits
'E' = Even Parity and 1 Stop Bit 'O' = Odd Parity and 1 Stop Bit

SCIRxBreak -- Return true if break is seen for at least milliseconds

Description: This function can be called to determine if a long break signal is being received.

Prototype: **bool SCITxBreak(short millisecs)**

Inputs: **millisecs** is the length of the break for which you wish to test.

Returns: Returns a Boolean that is TRUE if a break of the specified length was received.

Notes: This function will return TRUE if the break signal is received for at least duration **millisecs** ms. If the serial line was not in a break condition when the function is called, it will return FALSE immediately.

SCIRxFlush -- Delete any data in the receive queue

Description: SCIRxFlush simply purges the receive queue and throws away any data contained therein.

Prototype: **void SCIRxFlush(void)**

SCIRxGetByte -- Return the next word, wait if block is true

Description: Retrieves one byte from the main UART.

Prototype: **ushort SCIRxGetByte(bool block);**

Inputs: **block -** a Boolean, TRUE if the call should wait until a byte has been received to return

Returns: Returns a short, the LSB of which contains the byte gotten or -1 if block was FALSE and there was not a byte available in the receive queue.

Notes: If **block** is TRUE and no characters are available, the function will wait until a character is available before returning. If **block** is FALSE, and there are no characters are available it will return -1. If a byte was received, the function will return that byte in the LSB of the short returned. The MSB will contain error codes if applicable.

SCIRxGetChar -- Wait for, and return the next word

Description: This command waits for and returns the next byte from the UART. In the high byte of the short returned are error codes, if applicable.

Prototype: **short SCIRxGetChar(void);**

Returns: The character gotten from the UART is returned in the LSB of the short and the MSB contains error codes.

Notes: See Error Codes.

SCIRxGetCharWithTimeout -- Return next byte from receive queue with timeout

Description: This command waits up to **millisecs** ms for the next character to arrive on the main UART.

Prototype: **short SCIRxGetCharWithTimeout(short millisecs);**

Inputs: **millisecs -** the number of milliseconds to wait for a character before returning an error.

Returns: A short, the LSB of which contains the character gotten, or which equals -1 if the function times out. In the high byte of the short returned are error codes, if applicable.

Notes: See Error Codes.

SCIRxHandshake -- Set receive flow control

Description: The BIOS SCI functions provide both hardware and software flow control options for both input and output operations with the setup functions SCIRxHandshake() and SCITxHandshake() and three enumerated setup constants.

Prototype: **void SCIRxHandshake(short hshk, char xon, char xoff)**

Inputs: **hshk** is the enumerated handshake selector (hshkOff by default)
xon is the character sent from the CF2 to resume transmission (CTRL-Q by default).
xoff is the character sent from the CF2 to pause transmission (CTRL-S by default)

Notes: Flow control can be OFF (hshkOff), ON using the auxiliary RS-232 control signals (hshkCtsRts), or ON using definable XON/XOFF characters. Flow control is only supported when the input or output drivers are working in buffered (interrupt driven) modes.

Input flow control, when enabled, tells the connected serial transmitting device to stop sending data when the CF2's 2048 word input buffer gets to within 256 words of overflowing. This gives the sender a minimum of 10ms at 230,400 BAUD to recognize the request and stop transmitting. For hardware flow control, the stop mechanism is the negation of the RSTXX signal (EIA negative). For software flow control, the stop mechanism is the transmission by the CF2 of an XOFF character, which can be user defined, but is typically a control-S (0x13) character.

When the buffer empties to within 512 words of overflowing, the CF2 tells the transmitting device that it's ok to resume sending data. This 256 words of hysteresis keeps from tying up both the receiver and transmitter with flow control signals possibly accompanying each character. For hardware flow control, the resume mechanism is the assertion of the RSTXX signal (EIA positive). For software flow control, the stop mechanism is the transmission by the CF2 of an XON character, which can be user defined, but is typically a control-Q (0x11) character.

SCIRxQueuedCount -- Return the number of characters in the receive queue

Description: This function allows you to find out how many characters are waiting in the receive queue to be received.

Prototype: **short SCIRxQueuedCount(void);**

Returns: Returns the number of characters currently in the receive queue.

Notes: In polled buffering mode this function will always return either 1 or 0 depending on the existence of a character waiting in the UART receive register. In interrupt-driven buffering mode, this function returns the number of characters waiting in the receive queue, up to the maximum, 2048.

SCIRxSetBuffered -- Select buffered (true) or non-buffered receive

Description: This function allows you to change the buffering scheme for the receive line of the main UART.

Prototype: **void SCIRxSetBuffered(bool buffered);**

Inputs: **buffered -** a Boolean used to specify the buffering mode desired. TRUE selects interrupt-driven buffering mode and FALSE selects polled mode or non-buffered mode.

Notes: By specifying interrupt driven buffering mode, the CF2's internal BIOS buffering scheme is activated providing an transparent receive buffer of 2048 characters. In polled buffering mode the receive queue still "exists" but have a size of one character reflecting the fact that only the internal UART registers are being used for buffering.

SCIRxTxIdle -- Return true if all Rx and Tx flags indicate idle

Description: Return TRUE if all Rx and Tx flags indicate idle

Prototype: **bool SCIRxTxIdle(void);**

Returns: Returns a boolean that is TRUE if both the transmit and receive sections of the UART are idle and both queues are empty.

Notes: This could be used to determine if the UART is ready to be shut down.

SCITxBreak -- Start (-1) , stop (0) , or send timed break (+ val)

Description: Sets the UART to transmit a break condition for the specified duration or to start or stop an indefinite break.

Prototype: **void SCITxBreak(short millisecs)**

Inputs: **millisecs** is the duration of the desired break in milliseconds. If it is -1 it will start an indefinite break and return, if it is 0 it will stop an indefinite break and return.

SCITxFlush -- Delete any data in the transmit queue

Description: Empties the transmit buffer and throws away any data not yet transmitted out the UART.

Prototype: **void SCITxFlush(void);**

Notes: **See STDIO Warning.**

SCITxHandshake -- Set transmit flow control

Description: Set transmit flow control

Prototype: **void SCITxHandshake(short hshk, char xon, char xoff)**

Inputs: **hshk** is the enumerated handshake selector (hshkOff by default)
xon is the character sent to the CF2 to resume transmission (CTRL-Q by default).
xoff is the character sent to the CF2 to pause transmission (CTRL-S by default)

Notes: Flow control can be OFF (hshkOff), ON using the auxiliary RS-232 control signals (hshkCtsRts), or ON using definable XON/XOFF characters. Flow control is only supported when the input or output drivers are working in buffered (interrupt driven) modes.

The BIOS SCI functions provide both hardware and software flow control options for both input and output operations with the setup functions SCIRxHandshake() and SCITxHandshake() and three enumerated setup constants.

Output flow control, when enabled, lets the connected serial receiving device tell the CF2 to stop sending data when the receiver's senses it is near overflow. For hardware flow control, the stop mechanism is the detection of a negated RSRXX signal (EIA negative). For software flow control, the stop mechanism is the receipt of an an XOFF character, which can be user defined, but is typically a control-S (0x13) character.

The CF2 resumes transmitting when the receiving device tells the CF2 it's ok to resume sending data. For hardware flow control, the resume mechanism is the detection by the CF2 of an asserted RSRXX signal (EIA positive). For software flow control, the resume mechanism is the receipt by the CF2 of an XON character, which can be user defined, but is typically a control-Q (0x11) character.

SCITxPutByte -- Transmit byte, wait if block is true

Description: Transmits one byte out the main UART. If **block** is TRUE and the transmit buffer is full, the function will wait until it is able to enqueue the byte before returning. If **block** is FALSE, and the function cannot enqueue the byte it will return FALSE without enqueuing. If the byte was enqueued or transmitted, the function will return TRUE.

Prototype: **bool SCITxPutByte(ushort data, bool block)**

Inputs: **data -** a short, the LSB of which should contain the byte you wish to send.
block - a Boolean, TRUE if the call should wait until byte has been sent to return

Returns: Returns TRUE if byte was transmitted or FALSE if block was set to FALSE and the byte could not be transmitted or enqueued (depending on buffering mode) immediately.

Notes: See Error Codes.

SCITxPutChar -- Transmit byte

Description: This function takes the LSB of **data** and puts it in the transmit queue of the main UART. If the SCI is in polled mode, SCITxPutChar() will wait for the transmit queue to be empty and then write the LSB of data to the main UART. The MSB of **data** is always ignored.

Prototype: **void SCITxPutChar(ushort data);**

Inputs: **data** is a ushort, the LSB of which is the character you wish to transmit.

SCITxQueuedCount -- Return the number of words in the transmit queue

Description: This function allows you to find out how many characters are waiting in the transmit queue.

Prototype: **short SCITxQueuedCount(void);**

Returns: In polled buffering mode this function will always return either 1 or 0 depending on the existence of a character waiting in the UART transmit register. In interrupt-driven buffering mode, this function returns the number of characters waiting in the transmit queue, up to the maximum, 512.

SCITxSetBuffered -- Select buffered (true) or non-buffered transmit

Description: This function allows you to change the buffering scheme for the transmit line of the main UART.

Prototype: **void SCITxSetBuffered(bool buffered);**

Inputs: **buffered -** a Boolean used to specify the buffering mode desired. TRUE selects interrupt-driven buffering mode and FALSE selects polled mode or non-buffered mode.

Notes: By specifying interrupt driven buffering mode, the CF2's internal BIOS buffering scheme is activated providing an transparent transmit buffer of 512 characters. In polled buffering mode the transmit queue still "exists" but has a size of one character reflecting the fact that only the internal UART registers are being used for buffering.

SCITxWaitCompletion -- Wait for all transmission to complete

Description: Waits for the transmit buffer to finish sending all characters in the buffer and then returns.

Prototype: **void SCITxWaitCompletion(void);**

EIAAssertTXX – Assert /TXX

Description: Assert /TXX

Prototype: **bool EIAAssertTXX(bool set);**

Inputs: TRUE = CMOS low and EIA pos., FALSE = CMOS high and EIA neg

Returns: ??

EIACheckRXX – Get State of /RXX

Description: Get State of /RXX

Prototype: **bool EIACheckRXX(void);**

Inputs: None

Returns: TRUE if /RXX is asserted (CMOS low, EIA positive)

EIAEnableRx – Enable RS232 receivers

Description: Enable RS232 receivers

Prototype: **bool EIAEnableRx(bool enable);**

Inputs: TRUE to enable FALSE to disable

Returns: returns the previous state

EIAForceOff – Force RS232 transmitters off

Description: Force RS232 transmitters off

Prototype: **bool EIAForceOff(bool forceoff);**

Inputs: Pass TRUE to turn the drivers off and FALSE to turn them on

Returns: Returns the previous state of the forced condition

System Clock Timing Functions

TMGGetSpeed – Get the system clock frequency

Description: Returns the system clock frequency in kHz.

Prototype: **ushort TMGGetSpeed(void);**

Inputs: None

Returns: Clock frequency in kHz.

TMGSetSpeed – Set the system clock frequency

Description: Changes the system clock to the frequency passed to it.

Prototype: **ushort TMGSetSpeed(ushort kHz);**

Inputs: **kHz** is Clock frequency in kHz.

Returns: The actual clock frequency set.

Notes: The system is clocked from a voltage-controlled oscillator controlled by a phase-locked loop which is built into the 68332.

TMGSetSysClock – Set the system clock frequency with option lock wait disable

Description: Set the system clock frequency with option lock wait disable

Prototype: **short TMGSetSysClock(ushort kHz, bool dontWaitLock);**

Inputs: **kHz** is Clock frequency in kHz.
dontWaitLock is TRUE if you wish execution to return immediately before the PLL re-locks on the new frequency

Returns: Previous speed in kHz

TMGSetupCLKOUTPin – Setup the CLKOUT pin

Description: Setup the CLKOUT pin

Prototype: **void TMGSetupCLKOUTPin(bool onRunning, bool onLPSTOP);**

Inputs: **onRunning** is a boolean that indicates whether CLKOUT should be on while the machine is running

onLPSTOP is a boolean that indicates whether CLKOUT should be on while the machine is in LPSTOP mode

Returns: Nothing

Table Driven Command Processor Functions

CmdConfirm –Prompt for confirmation

Description: Prompt the user for confirmation. Any prompt and any character. Example:

Prototype: **bool CmdConfirm(char *prompt, char trueReply);**

Inputs: ***prompt** is a string to display
***trueReply** is the character that will return a TRUE

Returns: TRUE or FALSE depending on if ***trueReply** is received

Notes: Handles upper and lower case automatically. Example:
cprintf("CmdConfirm returned %s\n",
(CmdConfirm("'Y' or 'y'? for TRUE otherwise FALSE: ", 'Y'))?("TRUE"):("FALSE"));

CmdDispatch – Dispatch command

Description: Dispatch command with the options all contained in the CmdInfoPtr.

Prototype: **char *CmdDispatch(CmdInfoPtr cip);**

Inputs: ***cip** is a pointer to the CmdInfoPtr structure

Returns: Pointer to command string

CmdExpectRange – Validate Range

Description: Return non-zero if the next two arguments specified by the argc index are numeric values, and it places copies of the range in start and end.

Prototype: **int CmdExpectRange(CmdInfoPtr cip, short index, long *start, long *end);**

Inputs: ***cip** is a pointer to the CmdInfoPtr structure
***index** is the index to the arguments
***start** is the start range
***end** is the end range

Returns: Returns non-zero if the next two arguments specified by the argc index are numeric values

Notes: Note this function only works correctly if CmdExtractArgValues() has already been called.

CmdExpectValue – Validate and Get Value

Description: Returns non-zero if the specified argument index is a numeric value, and it places a copy of that number in the variable value.

Prototype: **int CmdExpectValue(CmdInfoPtr cip, short index, long *value);**

Inputs: ***cip** is a pointer to the CmdInfoPtr structure
***index** is the index into the arguments
***value** get the argument

Returns: Returns 1 if the argument was extracted and zero otherwise

Notes: Note this function only works correctly if CmdExtractArgValues() has already been called.

CmdExtractArgValues – Say what

Description: This function converts the presumably numeric strings into values for direct manipulation. It begins at the argument specified in first and continues through (inclusive) to the argument specified in last. It uses the value in radix (2, 8, 10, 16) for the default number base, though explicit prefixes will override the default. It returns non-zero only if all of the arguments in range were numeric.

Prototype: **int CmdExtractArgValues(CmdInfoPtr cip, short first, short last, short radix);**

Inputs: **cip** is a pointer to the CmdInfoPtr structure
first is the first value
last is the last value
radix is the number base

Returns: Zero if there was an error and non-zero otherwise

CmdExtractAVDosSwitches – Extract DOS switches from command line

Description: Extracts DOS switches from a standard argc/**argv.

Prototype: **short CmdExtractAVDosSwitches(short argc, char **argv, char *fmt, ...);**

Inputs: ??

Returns: ??

CmdExtractCIDosSwitches – Say what

Description: Extracts DOS switches from CmdInfoPtr.

Prototype: **short CmdExtractCIDosSwitches(CmdInfoPtr cip, char *fmt, ...);**

Inputs: ??

Returns: ??

CmdIsDigit – Is passed digit valid under number base

Description: Determines if a digit passed is valid under a given number base . If the digit is valid it is converted to its base 10 representation and saved at the location passed.

Prototype: **int CmdIsDigit(short c, short base, short *val);**

Inputs: **c** is the character to test
base is the number base to test the character with
***val** is a pointer the converted value is written

Returns: 1 if the digit is valid and was converted zero otherwise

Notes:

CmdIsNumber– Is Command a Number

Description: Determines if argument is a number and returns it in *value.

Prototype: **int CmdIsNumber(char **s, long *value, short base);**

Inputs: ****s** is argv
***value** is where the extracted number goes
base is the number base to test the character with

Returns: Returns TRUE if *s points to a valid number in any of the four common number bases and leaves *s pointing to the terminating character else returns FALSE with s unchanged. If the number is valid it is written to *value.

CmdParse – Parse command structure

Description: Parse command structure. This is used, for example, prior to a call to extract arguments and switches.

Prototype: **char *CmdParse(CmdInfoPtr cip);**

Inputs: **cip** is a pointer to the CmdInfoPtr structure

Returns: Returns NULL if no errors, or a pointer to a string with the appropriate error message.

CmdSetNextCmd – Kill repeat commands with CR

Description: This is used inside of a command to prevent the command from being repeat-called with an inadvertent <enter>. You do this by calling CmdSetNextCmd(cip, 0);.

Prototype: **void CmdSetNextCmd(CmdInfoPtr cip, char *nextcmd);**

Inputs: **cip** is a pointer to the CmdInfoPtr structure
***nextcmd** is a pointer to the next command when just a CR is entered.

Returns: Nothing

CmdStdBreak – Send a BREAK

Description: Returns a CMD_BREAK

Prototype: **char *CmdStdBreak(CmdInfoPtr cip);**

Inputs: **cip** is a pointer to the command info

Returns: This routine simply returns the special code CMD_BREAK

CmdStdCmdTest – Test Commands

Description: This is a great routine for debugging your own command handlers. If your custom handler is misbehaving, feed this routine the same parameter strings, and it will decode and display the various fields that your routine is working with.

Prototype: **char *CmdStdCmdTest(CmdInfoPtr cip);**

Inputs: **cip** is a pointer to the command info

Returns: Returns NULL if no errors, or a pointer to a string with the appropriate error message.

CmdStdErrText – Lookup Error Text

Description: Look up the error text associated with an error code.

Prototype: **char *CmdStdErrText(short errID);**

Inputs: **errID** is the error code to lookup

Returns: A pointer to the error text associated with the error code.

CmdStdHelp – Display a help menu

Description: This routine walks through the command table and generates a one or two column help menu from the two text fields in each entry. It skip over entries with the empty string for the help field, and it also skips over entries which have a lower privilege level than the current default.

Prototype: **char *CmdStdHelp(CmdInfoPtr cip);**

Inputs: **cip** is a pointer to the command info

Returns: Returns NULL if no errors, or a pointer to a string with the appropriate error message.

CmdStdLPGets – Line Input

Description: Line input function with edit.

Prototype: **short CmdStdLPGets(char *linebuf, short linelen);**

Inputs: ***linebuf** is the pointer to the buffer for input characters
linelen is the maximum length of linebuf

Returns: The number of characters entered into linebuf

CmdStdRun – Run the command processor

Description: This function simply makes an indirect call to the handler attached to the first entry in the command table. This is almost always PDCCmdStdPicoRun.

Prototype: **char *CmdStdRun(CmdInfoPtr cip);**

Inputs: **cip** is a pointer to the command info

Returns: Returns NULL if no errors, or a pointer to a string with the appropriate error message.

CmdStdSetup – Sets up the Command Table prior to a CmdStdRun

Description: Given a pointer to a CmdInfo structure, this function sets up all of the fields with safe default values in preparation for further standard interactive mode calls, using information from the required pointer to the target command table. The altgets field allows you to choose the standard C library gets() function by simply passing zero, or specify a more appropriate line input function, perhaps with better line editing capabilities.

Prototype: **void CmdStdSetup(CmdInfoPtr cip, CmdTablePtr ctp, short (*altgets)(char *, short));**

Inputs: **cip** is a pointer to the command info
ctp is the command table
altgets is the get line function for the command processor to use

Returns: Nothing

Time Processing Unit

Summary of TPU Drivers and Functions	
TUBlockDuration	Return expected block duration in ms at current baud
TUClose	Close the specified port and release its memory
TUGetDefaultParams	Return the default TPU UART open parameters
TUInit	Initialize the TPU UART module
TUOpen	Open a TPU UART port for serial communications
TURelease	Close all ports then release all memory and resources allocated to TPU UARTs
TURxFlush	Delete any data in the receive queue
TURxGetByte	Wait for, and return the next word
TURxGetByteWithTimeout	Return next word
TURxGetBlock	Receive a block of bytes with timeout
TURxPeekByte	Fetch Nth byte in receive queue without deleting
TURxQueuedCount	Return the number of words in the receive queue
TUSetDefaultParams	Setup new default TPU UART open parameters
TUTxFlush	Delete any data in the transmit queue
TUTxPrintf	Transmit using standard printf conventions
TUTxPutByte	Send byte
TUTxPutBlock	Transmit a block of bytes with timeout
TUTxQueuedCount	Return the number of words in the transmit queue
TUTxWaitCompletion	Wait for all transmission to complete

TUBlockDuration – Return expected block duration in ms at current baud

Description: word

Prototype: **long TUBlockDuration(TUPort *tup, long bytes);**

Inputs: ***tup** is a pointer to the port
bytes is number of bytes for computation

Returns: Transmission duration in milliseconds

Notes:

TUClose – Close the specified port and release its memory

Description: word

Prototype: **void TUClose(TUPort *tup);**

Inputs: ***tup** is a pointer to the port

Returns: Nothing

Notes:

TUGetDefaultParams – Return the default TPU UART open parameters

Description: word

Prototype: **TUChParams *TUGetDefaultParams(void);**

Inputs: None

Returns: A pointer to a TUChParams struct containing the TPU UART parameters

Notes:

```
typedef struct
        {
        short bits;             // data bits exclusive of start, stop, parity
        short parity;           // parity: 'o','O','e','E', all else is none
        short autobaud;         // automatically adjust baud when clock changes
        long baud;              // baud rate
        short rxpri;            // receive channel TPUPriority
        short txpri;            // transmit channel TPUPriority
        short rxqsz;            // receive channel queue buffer size
        short txqsz;            // transmit channel queue buffer size
        short tpfbsz;           // transmit channel printf buffer size
        } TUChParams;
```

TUInit – Initialize the TPU UART module

Description: Call this at the start of your program before opening TPU UART ports.

Prototype: **void TUInit(Callocf *callocf, Freef *freef);**

Inputs: References to calloc and free functions used so that the TPU UART software can manage the memory used by the port.

Returns: Nothing

TUOpen – Open a TPU UART port for serial communications

Description: Specify separate valid TPU channels (1 to 15) for receive and transmit. Specify -1 for rxch for a transmit only port, -1 for txch for receive only.

Prototype: **TUPort *TUOpen(short rxch, short txch, long baud, TUChParams *tp);**

Inputs:
rxch is the receive TPU Channel Number
txch is the transmit TPU Channel Number
baud is the desired baud
***tp** is a pointer to the TUChParams containing the channel parameters

Returns: A pointer to the port or NULL if an error occurs

Notes:

TURelease – Close all ports then release all memory and resources allocated to TPU UARTs

Description: This is done automatically when your program quits. You use it if you need to dynamically reconfigure your systems TPUs.

Prototype: **void TURelease(void);**

Inputs: None

Returns: Nothing

TURxFlush – Delete any data in the receive queue

Description: Delete any data in the receive queue

Prototype: **void TURxFlush(TUPort *tup);**

Inputs: ***tup** is a pointer to the port

Returns: Nothing

TURxGetByte – Wait for, and return the next word

Description: Wait for, and return the next word

Prototype: **short TURxGetByte(TUPort *tup, bool block);**

Inputs:
***tup** is a pointer to the port
block is TRUE if you want to wait for a character and FALSE if you do not wish to wait

Returns: The character

TURxGetByteWithTimeout – Return next word

Description: Return next word

Prototype: **short TURxGetByteWithTimeout(TUPort *tup, short millisecs);**

Inputs:
***tup** is a pointer to the port
millisecs is the timeout value in milliseconds

Returns: Returns the next word or -1 on timeout

TURxGetBlock – Receive a block of bytes with timeout

Description: Receive a block of bytes with timeout

Prototype: **long TURxGetBlock(TUPort *tup, uchar *buffer, long bytes, short millisecs);**

Inputs:
***tup** is a pointer to the port
***buffer** is a pointer to the buffer to hold the bytes
bytes is the maximum number of bytes
millisecs is the number of milliseconds to wait before timing out

Returns: The number of bytes read

Notes:

TURxPeekByte – Fetch Nth byte in receive queue without deleting

Description: Fetch Nth byte in receive queue without deleting

Prototype: **short TURxPeekByte(TUPort *tup, short index);**

Inputs:
***tup** is a pointer to the port
index is the offset (index) within the input queue from which to extract the byte

Returns: The byte at the offset (index)

Notes:

TURxQueuedCount – Return the number of words in the receive queue

Description: Return the number of words in the receive queue

Prototype: **short TURxQueuedCount(TUPort *tup);**

Inputs: ***tup** is a pointer to the port

Returns: The number of words (bytes) in the receive queue

Notes:

TUSetDefaultParams – Setup new default TPU UART open parameters

Description: Setup new default TPU UART open parameters

Prototype: **void TUSetDefaultParams(TUChParams *rp);**

Inputs: ***rp** is a pointer to a TUChParams struct containing the new parameters

Returns: Nothing

Notes: The new setting will only apply to ports opened after the set call.

TUTxFlush – Delete any data in the transmit queue

Description: Delete any data in the transmit queue

Prototype: **void TUTxFlush(TUPort *tup);**

Inputs: ***tup** is a pointer to the port

Returns: Nothing

Notes:

TUTxPrintf – Transmit using standard printf conventions

Description: Transmit using standard printf conventions

Prototype: **short TUTxPrintf(TUPort *tup, char * str, ...);**

Inputs: ***tup** is a pointer to the port
str is a standard printf format string and args

Returns: The number of characters written or negative if there was an error

Notes:

TUTxPutByte – Send byte

Description: Send byte

Prototype: **bool TUTxPutByte(TUPort *tup, ushort data, bool block);**

Inputs: ***tup** is a pointer to the port
data is the byte to send
TRUE if you want to wait for a character and FALSE if you do not wish to wait

Returns: Returns TRUE if the byte was sent and FALSE if the queue was full and if the block parameter (ok to block/wait) was FALSE. In other words, it returns TRUE unless block is FALSE and the queue is full preventing eventual transmission.

Notes:

TUTxPutBlock – Transmit a block of bytes with timeout

Description: Transmit a block of bytes with timeout

Prototype: **long TUTxPutBlock(TUPort *tup, uchar *buffer, long bytes, short millisecs);**

Inputs: ***tup** is a pointer to the port
buffer is a pointer to the block of data to send
bytes is the number of bytes to send
millisecs is the number of milliseconds to wait before the function times-out

Returns: The number of bytes actually sent

Notes:

TUTxQueuedCount – Return the number of words in the transmit queue

Description: Return the number of words in the transmit queue

Prototype: **short TUTxQueuedCount(TUPort *tup);**

Inputs: ***tup is a pointer to the port**

Returns: The number of words (bytes) in the transmit queue

Notes:

TUTxWaitCompletion – Wait for all transmission to complete

Description: Wait for all transmission to complete

Prototype: **void TUTxWaitCompletion(TUPort *tup);**

Inputs: ***tup** is a pointer to the port

Returns: Nothing

Notes:

Utility Functions

execstr -- Pass command string from running application to PicoDOS command shell

Description: Pass a command to the PicoDOS command shell.

Prototype: **short execstr(char *cmdstr);**

Inputs: **cmdstr** is a zero terminated C string containing the command and parameters just as would be typed at the PicoDOS command prompt.

Returns: Returns enum { execstrNoCmdMatch = -1, execstrNoError = 0, execstrGeneralFailure = 1 };

Notes: Notes.

flogf -- printf to console and log file (after Initflogf())

Description: This function works like printf, except that it sends the formatted data to either, neither, or both an append file and the stdout console. When working to the console, the flogf takes care of checking the current state of the EIA driver and if it's off, turns it on while sending then turns it back off when complete. Console writes also blocks until all characters have been sent. Similarly, when writing to a file, flogf checks the current CF enable state and if it's off, turns it on while writing, then back off. Each flogf targeting a file opens the file for append mode on entry and closes and flushes the file on exit.

Prototype: **short flogf(char *format, ...);**

Inputs: **format** is identical to standard printf formats and variable arguments.

Returns: Returns the number of characters written.

Notes: Notes.

Initflog -- Setup for future flogf operations

Description: Setup for printf-like runtime logging to a file and/or the console.

Prototype: **void Initflog(char *fname, bool echoToConsole);**

Inputs: **fname** is a zero terminated C string contiaining the file or full path name to log/append future flogf messages to.
echoToConsole is TRUE if flogf messages are to be echoed to the stdout console.

Returns: Returns nothing.

Notes: Pass zero for fname and TRUE for echoToConsole to just show messages on the console.

pdcfinfo -- Return PicoDOS file system size information (legacy support code)

Description: Return PicoDOS file system information with a CF8/AT8 compatable function call.

Prototype: **short pdcfinfo(char *drive, long *size, long *free)PICO_CALL(pdcfinfo);**

Inputs: **drive** is a string pointer in the form of "A:", "B:", etc.
size is a pointer to a long variable to hold the size of the media in bytes
free is a pointer to a long variable to hold the available free space in bytes

Returns: Returns zero on success or 1 if the named drive is not available.

Notes: New programs should use the PicoDOS DIR functions DIRFreeSpace and DIRTotalSpace.

picodosver -- Return a string containing PicoDOS version information (legacy support code)

Description: Return PicoDOS version information with a CF8/AT8 compatable function call.

Prototype: **char * picodosver(void);**

Inputs: Nothing

Returns: Returns a C string in the form "nnnnn-v.rs" where nnnnn is the serial number, v is the current version of PicoDOS, r is the release level, and s is the sub-release level.

Notes: New programs should use the BIOS global variables for this information:
BIOSGVT.CF2SerNum
BIOSGVT.BIOSVersion
BIOSGVT.BIOSRelease
BIOSGVT.PICOVersion
BIOSGVT.PICORelease

sscandate -- scan date string into year, mon, and mday fields of struct tm

Description: This is a general purpose date string parser that accomodates a variety of date specification styles.

Prototype: **short sscandate(const char *str, struct tm *ptm, DateFieldOrder dfo);**

Inputs: **str** is a zero terminated C string containing date in the form:

YEAR<delim>MONTH<delim>DAY (specifying enum YYMMDD)
MONTH<delim>DAY<delim>YEAR (specifying enum MMYYDD)
DAY<delim>MONTH<delim>YEAR (specifying enum DDMMYY)

<delim> may be any single comma, space, dash, period, or slash.
YEAR may a full 4 digits or 2 digits assumed between 1970 and 2069
MONTH may be 1-12, full text (January, February, ..., December) or three character abbreviations.

ptm is a standard C library tm structure from <time.h> with fields tm_year, tm_mon, and tm_mday filled in from the scan.

Returns: Returns the lenght of the scanned string or zero if the string is invalid.

Notes: This routine is used by the PicoDOS QRdate and QRdatetime query/reply functions.

```
typedef enum {  YYMMDD      // ISO
              , MMDDYY      // US
              , DDMMYY      // European
              } DateFieldOrder;
```

sscantime -- scan time string into hour, min, and sec fields of struct tm

Description: This is a general purpose time string parser.

Prototype: **short sscantime(const char *str, struct tm *ptm);**

Inputs: **str** is a zero terminated C string containing time in the form:
HOURS<delim>MINUTES<delim>SECONDS and optional AM or PM for 12 hour time.
<delim> may be any single comma, space, dash, period, or slash.

ptm is a standard C library tm structure from <time.h> with fields tm_hour, tm_min, and tm_secs filled in from the scan.

Returns: Returns the lenght of the scanned string or zero if the string is invalid.

Notes: This routine is used by the PicoDOS QRtime and QRdatetime query/reply functions.

Virtual EEPROM Functions

Summary of VEE Functions	
VEECheck	Check the Virtual EEPROM and return free size if OK
VEEClear	Clear the entire Virtual EEPROM
VEEDelete	Delete a Virtual EEPROM variable
VEEFetchFloat	Return float value from Virtual EEPROM
VEEFetchLong	Return long value from Virtual EEPROM
VEEFetchNext	Find the next valid VEE entry (NULL to start)
VEEFetchStr	Return C string from Virtual EEPROM
VEEFetchVar	Fetch a Virtual EEPROM variable
VEEGetData	Return pointer to VEE variable data field or zero
VEEGetName	Return VEE variable name or null string pointer ("\0")
VEEStoreFloat	Store float data to Virtual EEPROM
VEEStoreLong	Store long data to Virtual EEPROM
VEEStoreStr	Store string data to Virtual EEPROM

VEECheck -- Check the Virtual EEPROM and return free size if OK

Description: Confirms VEE internal data structures and returns the free space in bytes or -1 if there is an error

Prototype: **short VEECheck(void);**

Inputs: None.

Returns: Returns zero or one of the enumerated VEE error codes.

VEEClear -- Clear the entire Virtual EEPROM

Description: Delete the entire virtual eeprom.

Prototype: **bool VEEClear(void);**

Inputs: None.

Returns: Returns TRUE if successful.

Notes: This erases the entire VEE except for a single byte that is copied and retained for the PBM startup vector. Use this only to fix unrecoverable problems that could happen if an applicaition program corrupts the VEE flash.

VEEDelete -- Delete a Virtual EEPROM variable

Description: Delete a virtual eeprom variable.

Prototype: **bool VEEDelete(char *name);**

Inputs: **name** is a pointer to a zero terminated C string of up to 15 characters that locates the entry.

Returns: Returns TRUE if found and successful, FALSE if not found or the VEE is locked.

VEEFetchFloat -- Return float value from Virtual EEPROM

Description: Return either a float value found in the virtual eeprom or a default value.

Prototype: **float VEEFetchFloat(char *name, float fallback);**

Inputs: **name** is a pointer to a zero terminated C string of up to 15 characters that identifies the entry.
fallback is a float used as a default fallback if the vee search fails to find the veename entry

Returns: Returns a four byte float value.

Notes: The eeprom entry may be either a float value or a string that evaluates to a float value (uses atof())

VEEFetchLong -- Return long value from Virtual EEPROM

Description: Return either a signed long value found in the virtual eeprom or a default value.

Prototype: **long VEEFetchLong(char *name, long fallback);**

Inputs: **name** is a pointer to a zero terminated C string of up to 15 characters that identifies the entry.
fallback is a signed long used as a default fallback if the vee search fails to find the veename entry

Returns: Returns a signed long value.

Notes: The eeprom entry may be either a long value or a string that evaluates to a long value (uses atol())

VEEFetchNext -- Find the next valid VEE entry (NULL to start)

Description: Return the next sequential VEEVar entry.

Prototype: **VEEVar VEEFetchNext(VEEVar *prev);**

Inputs: **prev** is a pointer to a VEEVar structure, or zero to find the first.

Returns: Returns a pointer to the next VEEVar structure or zero if there are no more.

Notes: Used to iterate through the VEE for a list of directory of VEE entries.

VEEFetchStr -- Return C string from Virtual EEPROM

Description: Return either a signed long value found in the virtual eeprom or a default value.

Prototype: **char *VEEFetchStr(char *name, char *fallback);**

Inputs: **name** is a pointer to a zero terminated C string of up to 15 characters that identifies the entry.
fallback is a C string used as a default fallback if the vee search fails to find the veename entry

Returns: Returns a C string.

Notes:

VEEFetchVar -- Fetch a Virtual EEPROM variable

Description: Return a pointer to a VEEVar structure.

Prototype: **VEEVar VEEFetchVar(char *name);**

Inputs: **name** iis a pointer to a zero terminated C string of up to 15 characters that locates the entry.

Returns: Returns a pointer to a VEEVar structure or zero if the requested name can't be found.

Notes: You can determine the type of data from the type field, and access the name and stored data using the companion VEEGetName and VEEGetData functions.

VEEGetData -- Return pointer to VEE variable data field or zero

Description: Given a VEEVar structure pointer, return a pointer to its data field, which is guaranteed to begin on an even boundary.

Prototype: **void* VEEGetData(VEEVar *vvp, short *size);**

Inputs: **vvp** is a pointer to a VEEVar structure from VEEFetchVar, VEELookup, or VEEFetchNext.
size is an optional (not used if its zero) pointer to a short to hold the size of the data field in bytes.

Returns: Returns a non-zero pointer or zero if there is a problem.

Notes:

VEEGetName -- Return VEE variable name or null string pointer ("\0")

Description: Given a VEEVar structure pointer, return a pointer to its zero terminated C string.

Prototype: **char* VEEGetName(VEEVar *vvp);**

Inputs: **vvp** is a pointer to a VEEVar structure from VEEFetchVar, VEELookup, or VEEFetchNext.

Returns: Returns a pointer to a zero terminated C string or zero if there is a problem.

Notes: Notes.

VEEStoreFloat -- Store float data to Virtual EEPROM

Description: Store a 32 bit IEEE floating point binary value (C float) into the virtual eeprom.

Prototype: **bool VEEStoreFloat(char *name, float fvalue);**

Inputs: **name** is a pointer to a zero terminated C string of up to 15 characters that identifies the entry. **fvalue** is

Returns: Returns TRUE if successful.

Notes: If a VEE variable with the same name already exists, it will be deleted to make room for the new one.

VEEStoreLong -- Store long data to Virtual EEPROM

Description: Store a 32 bit long binary value (C long or ulong) into the virtual eeprom.

Prototype: **bool VEEStoreLong(char *name, long lvalue);**

Inputs: **name** is a pointer to a zero terminated C string of up to 15 characters that identifies the entry. **lvalue** is the 32 bit binary value to store

Returns: Returns TRUE if successful.

Notes: If a VEE variable with the same name already exists, it will be deleted to make room for the new one.

VEEStoreStr -- Store string data to Virtual EEPROM

Description: Store a zero terminated C string into the virtual eeprom.

Prototype: **bool VEEStoreStr(char *name, char *str);**

Inputs: **name** is a pointer to a zero terminated C string of up to 15 characters that identifies the entry. **str** is zero terminated C string

Returns: Returns TRUE if successful.

Notes: If a VEE variable with the same name already exists, it will be deleted to make room for the new one.