

# Pico Computing Inc.



## **Pico.dll Documentation**

**Version 5.0.0.3. Apr 16th, 2010.**

Pico Computing, Inc.  
150 Nickerson, Suite 311  
Seattle, WA, 98109-1634  
(206) 283-2178  
[www.picocomputing.com](http://www.picocomputing.com)

## 1 Overview

This manual describes Pico.dll - a comprehensive interface to Pico Cards which enables a program in C++ to access all the facilities of the cards.

Other manuals in this help library can be located at [GuideToDocumentation.pdf](#)

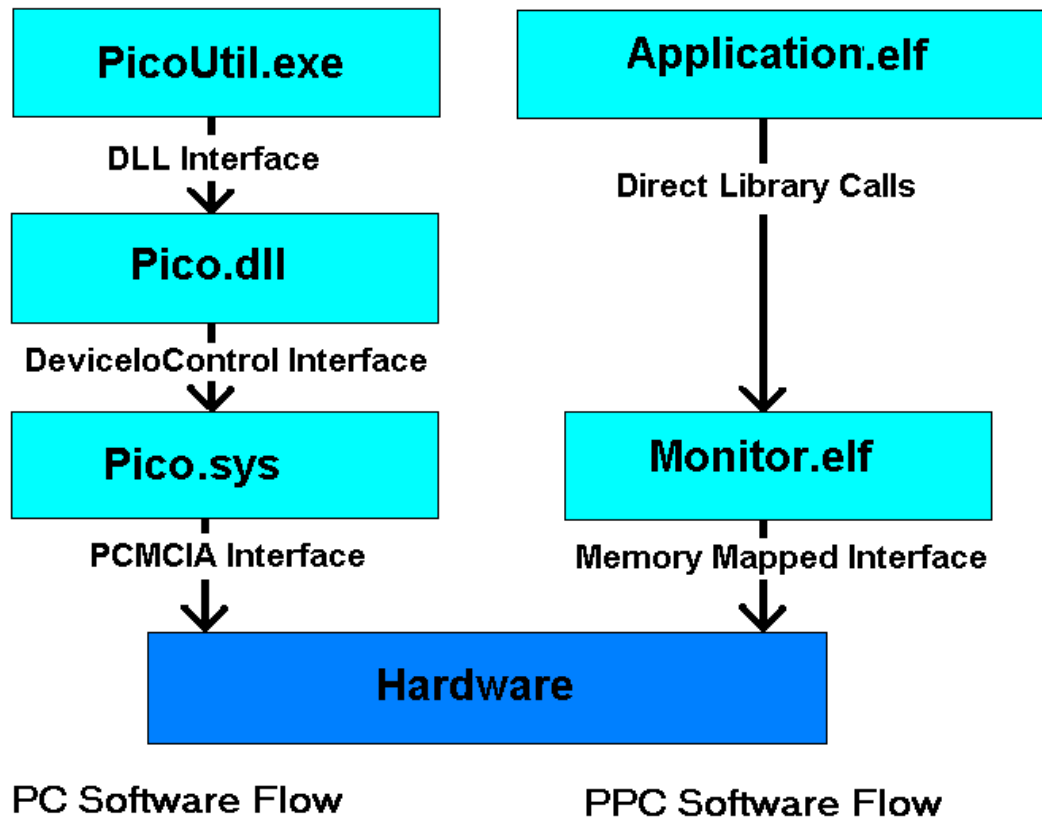
NOTE: This link will access the pdf from the PicoComputing.com Website .

## 2 Architecture

### Overview

Pico.dll provides a C++ API (Application Programming Interface) to any of the Pico Card through a class called PicoXface. Pico.dll spares the programmer from the lowest level detail of interfacing with a Pico Card. Since it is implemented as a dll it runs in the address space of the calling program. Pico.dll makes calls to a driver [Pico.sys](#). Since Pico.sys is a driver it operates in its own address space and at the innermost ring of the operating system. Application programs such as [PicoUtil.exe](#) use the PicoXface class. Pico.sys is intimately concerned with the operation of the firmware and the [hardware](#) of the Pico Cards. This manual describes the operation of Pico.dll using the PicoXface class. Updates to the Pico software and firmware can be obtained directly from the Pico Computing website ([www.picoComputing.com](http://www.picoComputing.com)).

The Pico driver fits within the software stack as illustrated below:

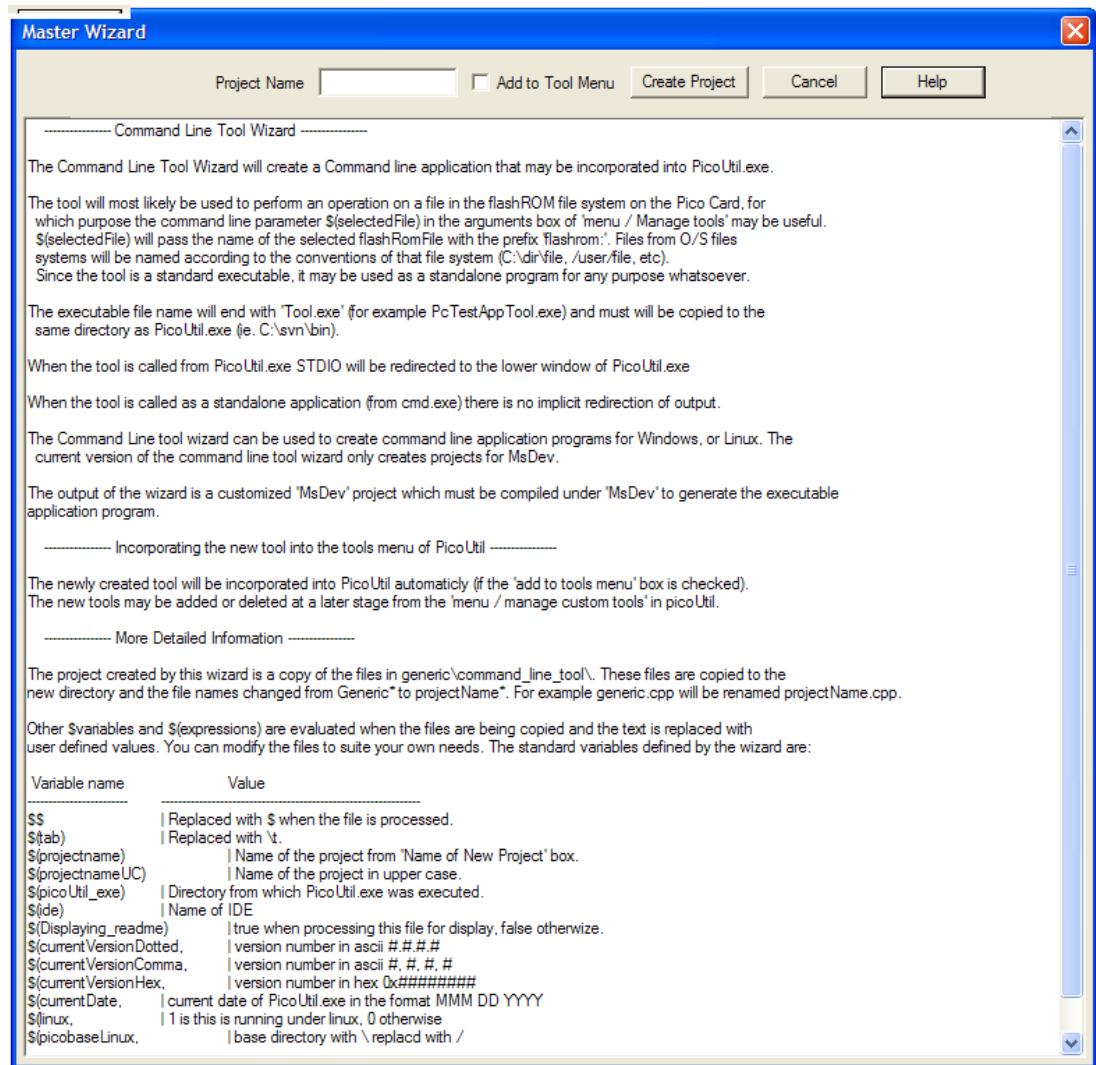


Programs on the left run on the PC host under Windows-2000, Windows-XP, or Linux 2-6. Programs

on the right run as embedded applications, under Micro COS-ii, or under Linux 2.6.

### 3 Creating a sample program

The following code snippet illustrates the basic techniques of access a Pico Card or cards. This program can be created from PicoUtil using wizard: From the main menu select wizard / Create Simple PicoXface:



```

1. int main(int argc, char *argv[])
2.     {int          erC;
3.     char          erBuf[100],          //for output error message from
   CreatePicoXface()
4.     *erParamP,          //for error messages
5.     *paramsP=NULL;     //parameters to CreatePicoXface
6.     cGString        cs;
7.     PicoXface       *picoXfaceP;

```

```

8.      if ((picoXfaceP=CreatePicoXface(paramsP, erBuf, sizeof(erBuf), &erC)) ==
        NULL) {erParamP = erBuf; goto xit;}

9.      //Get and print version information.
10.     cs = picoXfaceP->FormatVersion();
11.     printf("%s\n", (const char*)cs);

12.     DeletePicoXface(PicoXfaceP);

```

Line 8 creates an instance of the class PicoXface:

**paramsP** contains parameters in the general format keyword=value,....

**erBuf** will receive any error message generated by the CreatePicoXface,

the integer **err** will receive the error number: zero if successful, -negative if an error.

CreatePicoXface returns a pointer to a PicoXface class which can be used to interface to the Pico Card.

Line 10. The function FormatVersion() is called. It returns a string result with the version information for Pico.Dll (the library file loaded by CreatePicoXface).

Line 11. Converts the result to an ordinary C++ char \* and prints it.

Line 12. Closes the PicoXface and releases any resources used when the interface was active.

## 4 PicoXface class members

Access to the Pico Card is obtained by creating a PicoXface class and calling functions within that class. PicoXface is defined in the file PicoXface.h. Each function in the class is declared virtual because the implementation of the class will be a DLL (Pico.dll) and the class members are therefore pointers to the respective functions. There are a number of important data types used throughout PicoXface:

- FLASHROM\_ADDR. This is a 64bit address or size field for any location in the flash ROM or the memory mapped address space of the Pico Card.
- The class PicoXface relies upon a class cGString. This is a generic string class. The reason for yet another string class is to maintain compatibility between the Linux implementation and the Windows implementation.

Many functions returns a positive number if the operation is successful, otherwise a negative error code. The negative error code refers to codes in the file Pico\_errors.xml and can be accessed by calling the class functions InterpretError, or FullError.

A PicoXface class can be created using the function **CreatePicoXface**:

```

PicoXface *CreatePicoXface(const char *createParamsP, char
  *erMsgBufP, int erMsgSize, int *errP);

```

as follows:

```

PicoXface *g_fxP;
char      erBuf[100];
int       erC=0;
g_fxP = CreatePicoXface(NULL, erBuf, sizeof(erBuf), &erC);
if (g_fxP == NULL) printf("%s\n", InterpretError(fileNo, buf,
  sizeof(buf)));

```

Create a new instances of class PicoXface and return a pointer to it. The parameter createParamsP has the general syntax keyword=value. Acceptable keywords are:

- "simulateFile=simulationFile" operates with a file rather than a Pico Card.
- "picoCard=#" specifies card number in a multiple pico card environment
- "readOnly=1" opens in readOnly mode
- "noconnect=1" does not actually connect to the card (used by picocommand for example for /help command)
- "exebase=directory Name" source file for pico.dll etc.

## 4.1 ChangeFileProperties

```
int ChangeFileProperties
    (FLASHROM_ADDR    addr,
     const char       *pcFileNameP,
     const char       *flashNameP,
     const char       *linkedFileNameP,
     const char       *noteP);
```

This function will change the properties associate with a particular file on the flash ROM file system. The file is positively identified by its handle (which is equal to its initial segment address). The addr field is the handle \* 8192.

The fields pcFileNameP, flashNameP, linkedFileNameP, and noteP are all fields within the header of the file. Any of these parameters can be NULL to signify no change, or "" to signify a change to a zero length value, or a non zero length string. In no case can the total file header exceed the current size of the file header. This header is nominally set to 256 bytes when the file is first written.

The function returns a positive number if the operation is successful, otherwise a negative error code.

The notesP field in nominally a string and is stored in the Flash ROM file system in the file header. The low order 12 bits of the addr field must be zero to logically address a file, however, if the least significant bit of the addr field is set to one it signals a multiple note field. The least significant bit is ignored in the addr parameters, however, the noteP field may then be in the format:

```
noteP = "variable1=value1\n"
        "variable2=value2\n"
        "variable3=value3\n\x00"
```

This enables multiple notes to be stored in the file header.

Available on:  
Pico E-12, E-14, E-15.

## 4.2 Connect/Disconnect

The functions in the class connect a PicoXface class to a specific Pico Card or to disconnect from the card.

```
int PicoXface::ConnectToDriver
    (int          dvcNo = 0,
     bool         disableKeyhole = false,
     int         dcmWaitMillisecs = 1000
    )

void PicoXface::DisconnectFromDriver(void);
```

The first parameter to Connect device number can be zero and Pico.dll will choose any available card. In a multi card system the dvcNo parameter can be used to specify a particular card. This function is automatically called when the dll function CreatePicoXface is called. Connect and disconnect would only be called independently to connect to or disconnect from a PicoXface object that already existed.

Parameters:

dvcNo	is the Pico Card number (1 thru 32), or zero (any card)
disableKeyhole	if true, all keyhole accesses are disabled
dcmWaitMilliSecs	specifies the time to wait for the dcm's to lock on the Pico Card.

Returns:

0 on success, negative error code on failure

### 4.3 Error Reporting Functions

void AddErrorContext	(const char *msgP);	Used by Pico.dll to amplify the context of an error
int Error1	(int erC, const char* contextP="", const char* parmP="");	Log an error
int ErrorSeverity	(int erC);	Returns the severity of the error as defined in the following enum
const char *InterpretError	(int erC, char *resultP, int resultSize);	Return a string summarizing the error with proper parameter substitution from the <b>Error1</b>
const char *FullError	(int erC, char *resultP, int resultSize);	Return a string describing in full the error with proper parameter substitution from the <b>Error1</b>
int GetErrorFuncsP	(void**addContextP, void**error1P, void**fullErrorP, void**interpretErrorP, void**errorSeverityP);	Used to attach internal error handlers to PicoXface
enum {	<b>XSEVERITY_NOERROR,</b> <b>XSEVERITY_INFORMATION,</b> <b>SEVERITY_HINT,</b> <b>XSEVERITY_WARNING,</b> <b>XSEVERITY_DANGEROUS,</b> <b>XSEVERITY_ERROR,</b> <b>XSEVERITY_SYSTEM,</b> <b>XSEVERITY_CATASTROPHIC</b>	
};		

These functions are used to manage errors. All Pico Software uses a consistent set of error numbers, documented in the file source\Pico\_Errors.xml. When the function **Error1** is called (usually from within Pico.dll) it will record context and possibly a parameter for the error message. Subsequent calls to **InterpretError**, or **FullError** will reformat the error message to include the parameters specified when **Error1** was called.

## 4.4 Debug Flags

The functions in this section read and write the debug flags.

```
uint32_t PicoXface::GetDebugFlags (void);
```

```
int PicoXface::SetDebugFlags
    (uint32_t    flags,
     bool        on = true
    )
```

Parameters:

flags		defines flags to set or reset
on	true	adds flags to current settings.
on	false	removes flags from current settings.

The flags may be | (OR-ed) together.

Switch name	Meaning	Applies to
BUG_DRIVER_SUMMARY	enables summary level debugging in Pico.sys	Pico.sys
BUG_DRIVER_DETAIL	enables detail level debugging in Pico.sys	Pico.sys
BUG_DRIVER_JTAG	enables JTAG debugging in Pico.sys	Pico.sys
BUG_DRIVER_KEYHOLE	enables Keyhole debugging in Pico.sys	Pico.sys
BUG_DRIVER_INTERRUPT	enables display of interrupt activity in Pico.sys	Pico.sys
BUG_DRIVER_POWER	enables display of power events in Pico.sys	Pico.sys
BUG_DRIVER_BM	enables display of Bus Mastering activity in Pico.sys	Pico.sys
BUG_ELF_LOAD	enables display of segment addresses when an elf file is executed from flash	Monitor.elf
BUG_ELF_PROGRAM	enables debugging flag for use inside an ELF program	User program
BUG_ELF_RFU	flags reserved for user	User program

The debug flags apply across program and subsystem boundaries.

The release version of Pico.sys has no debug capability. In order to activate debugging in Pico.sys the following steps must be performed:

1. copy c:\pico\bin\picoD.sys over c:\windows\system32\drivers\pico.sys
2. Start a program such as DebugView.exe available from [www.sysinternals.com](http://www.sysinternals.com)

## 4.5 CreateChannel and Bus Mastering

The Pico E-14 uses the cardbus interface which is capable of operating at PCI speeds – namely 33 Mwords/sec or 1Gbit/sec. In order to achieve speeds approaching this theoretical limit, the device must use 'burst mode DMA' mode. The Pico-E14 supports this mode and provides a file level interface. The actual speed depends upon the design of the PCI back plane and will vary significantly from machine to machine. DMA is an acronym for direct memory access. This technology is properly called called Bus Mastering.

The particular firmware which handles data sent over the DMA interface is referred to as a **(peripheral) device**. The core logic in the Pico Card handles the data and provides it to the device. Devices are

instantiated in the module **memoryPeripheral.v** within the Pico firmware. The bus mastering logic in the core of the Pico E-14 generates 32Bit address, 32Bit data signal, and some control signals for a DMA device. As each 32bit word is written to or read from the FPGA the address is incremented (by four). These signals are referred to as **picoAddr**, **picoData** etc. and collectively form the **PicoBus**.

The PicoBus interface also wraps the single word mode of carbus operation on both the E-14 and E-12 cards. (NOTE: the Pico E-12 core logic implements this single word mode, but does not implement a high speed DMA mode. The E-12 also operates in 16bit mode).

The transfer of data between the host and this device necessarily involves Pico.sys. This is because the lowest level hardware requires absolute machine memory addresses. These addresses are not available to an application program. These host addresses should not be confused with the picoAddr. Host addresses refer to buffers in the PC host memory, Pico addresses refer to memory mapped device addresses in the device.

```
cPicoChannel *PicoXface::CreateChannel(int channelNum=10, const char
*selectorP=NULL, const char *bitFileNameP=NULL);
```

The function **CreateChannel** will create a DMA stream between the host PC and the Pico Card. ChannelNum should be an integer in the range [1,1791]. The pointer returned refers to a cPicoChannel integer returned will be either a negative error code, or a positive file number. for example:

```
PicoXface      *g_fxP;
cPicoChannel  *channelP;
int            erC;
g_fxP         = CreatePicoXface(...);
.....
channelP      = g_fxP->CreateChannel(2);
if ((erC=channelP->GetError) < 0) printf("%s\n", InterpretError(erC,
buf, sizeof(buf)));
```

The channel number (2 in this case) must correspond to the memory addresses of the device on the Pico Card, and the pacing registers implemented in that device.

The channel functions Read() and Write() can then be used to transfer application data to/from the Pico Card, as follows:

```
channelP->Write(buf32P, byteCount)
```

will transfer data to the Pico Card, and

```
channelP->Read(buf32P, byteCount)
```

will transfer data from the Pico Card.

The address presented to the firmware (Pico Address) during an I/O operation can be specified using:

```
channelP->SetPicoAddress(newPicoAddress)
```

For a comprehensive discussion of Pico Channels and the closely related Pico Streams refer to [PicoChannel.pdf](#).

## 4.6 Defragment

```
int PicoXface::Defragment(bool allowNoPrimaryBoot=false);
```

Fragmentation is a fact of life in any storage system for which some files (bit image files in this case) require large contiguous blocks of storage. Other files (ELF files, general data files) can be assigned to completely discontinuous space and without compromise to performance. Fragmentation is exacerbated by the random allocation of files (see [Write Flash File](#)).

This command is used to reorganize the files on the Flash ROM system into tightly packed, sequential order. The command is useful if the Flash ROM system becomes so fragmented that it is not possible to write a contiguous bit file on the Flash ROM but there is still space available in the Flash ROM system.

### Theory of operation:

The files are copied from the flash ROM to a PC-host file, the flash ROM is erased, and the files are copied back from the PC-host file. The temporary file is stored in the directory `c:\pico\tmp`.

Since the operation erases the entire content of the flash ROM, however, because of optimizations in the write flash functions files which do not in fact move, will not be erased.

### Available on:

Pico E-12, E-14, E-15.

## 4.7 DeleteFiles

This functions is used to delete one or more files from the flash ROM file system. When a file is deleted the space is erased and the flash ROM sector appears to be 0xFF.

```
int PicoXface::DeleteFiles
    (const char  **filesP,
     int         fileCount,
     int         *errorsP
    );
```

DeleteFiles deletes the specified file(s) from the flash ROM on the card. Because a file is allocated in logical sectors, and the erase operation on a flash ROM erases a physical sector, the delete file function must restore surrounding files when deleting a specific file. There is a danger that the restore operation will not succeed in which case damage will occur to the surrounding files. When multiple files are deleted, Pico.dll groups the erase sectors to optimize the erase operation.

### Parameters:

filesP	array of filenames as strings
fileCount	number of filenames in the filesP array
errorsP	an array of ints to receive individual error codes for each file in filesP. Must have fileCount elements. May be NULL

### Returns:

0 on success, or negative if any of the files had an error

## 4.8 File Properties

Functions in this section return fields from the file header. The FLASHROM\_ADDR parameter is the address of the first record of the file and uniquely identifies the file

```
cGString PicoXface::LinkedFileFromAddr      (FLASHROM_ADDR addr);
cGString PicoXface::NoteFromAddr           (FLASHROM_ADDR addr);
cGString PicoXface::PcNameFromAddr         (FLASHROM_ADDR addr);
cGString PicoXface::FlashNameFromAddr      (FLASHROM_ADDR addr);
cGString PicoXface::DateFromAddr           (FLASHROM_ADDR addr);
private: bool PicoXface::FirstSectorOfFile (FLASHROM_ADDR addr);
```

Returns true if the specified flash ROM address is the first sector of a file.

```
private: int PicoXface::GetAllocation        (FLASHROM_ADDR addr,
FLASHROM_ADDR **allocationPP);
```

Allocates an array of sector addresses and returns the array in allocationPP. For bit file (which are allocated contiguously) the array contains the first and last addresses of the file only. For all other types of files allocationP contains as many entries as there are sectors in the file. The function returns the number of entries in allocationPP. The caller must free the allocationPP array.

## 4.9 File Size

Functions in the group return the size of the file. The file can be uniquely specified by its initial sector address, or by its name (which is also unique).

```
FLASHROM_ADDR PicoXface::FileAddrFromFlashName(const char *filenameP);
FLASHROM_ADDR PicoXface::FileSizeFromAddr (FLASHROM_ADDR addr);
int PicoXface::FileSizeInSectors (FLASHROM_ADDR addr);
```

## 4.10 File Type

Functions in the group return the type of the file. The file can be uniquely specified by its initial sector address, or by its name (which is also unique). The type returned in one of the constants:

<b>SECTOR_FREE=0,</b>	Sector is not assigned to any file
<b>SECTOR_ELF</b>	Sector is assigned to an ELF file
<b>SECTOR_BIT</b>	Sector is assigned to a bit file
<b>SECTOR_RESERVED</b>	Sector is reserved for system use
<b>SECTOR_DATA</b>	Sector is assigned to a data files (ie not an ELF or BIT file)
<b>SECTOR_STARTUP</b>	Sector is reserved for startup information (used when card is inserted)
<b>SECTOR_UNUSABLE</b>	Sectors has an error which makes it unusable

```
FILE_TYPE PicoXface::FileTypeFromAddr (FLASHROM_ADDR addr);
```

If the file is of type data and the last letters of the file name are .txt or .log the function will return 'TXT' or 'LOG'.

```
FILE_TYPE PicoXface::FileTypeFromName (const char *fNameP, uint8_t
*dataP=NULL);
```

If the file is of type data and the last letters of the file name are .txt or .log the function will return 'TXT' or 'LOG'.

```
cGString PicoXface::TypeNameFromType (FILE_TYPE type);
```

TypeNameFromType returns the text equivalent of the type:

```
"free", "elf", "bit", "tuple", "data", "startup", "unusable", "unknown", "txt", or "log";
```

```
FILE_TYPE PicoXface::SectorTypeFromAddr (FLASHROM_ADDR addr);
```

If the file is of type data and the last letters of the file name are .txt or .log the function will return 'TXT' or 'LOG'.

## 4.11 GetCurrentImage

```
cGString PicoXface::GetCurrentImage  
(int *errP = NULL);
```

Get the name of the current FPGA program, also known as FPGA image. The firmware provides the address used to load the FPGA file into the FPGA fabric. This function reads this information and returns the file corresponding to this address.

Parameters:

errP     an optional pointer to the return error code

Returns:

filename of the current FPGA program

Available on:

Pico E-12, E-14, E-15.

## 4.12 GetDriver

Functions in this group provide information about the underlying driver. This driver is usually the interface logic to Pico.sys, however, it may be the interface logic to the simulation, or may differ slightly under different operating systems.

```
PicoDrv * PicoXface::GetDriver (void) ;
```

Returns:     pointer to the current driver adapter

```
protected: cGString PicoXface::GetDriverDesc (void);
```

Returns a textual description of the driver.

## 4.13 Flash Size

Functions in this group return information about the flash ROM itself independent of the flash ROM file system imposed upon it.

```
FLASHROM_ADDR PicoXface::GetFlashSize (void );
```

Returns: The size in bytes of the flash ROM on the Pico Card.

```
int PicoXface::GetLogicalSectorCount(void);
```

Returns: The size in bytes of the flash ROM on the Pico Card divided by the logical sector size (8192 bytes).

```
int PicoXface::GetPhysicalSectorCount(void);
```

Returns: The size in physical sectors of the flash ROM on the Pico Card.

```
int PicoXface::GetPhysicalSectorSize();
    Returns: The size of the physical sector of the flash ROM on the Pico Card.

int PicoXface::GetUnusedSectorNum (bool physical);
    Returns: A sector on the flash ROM that is unused. This call is used by the SelfTest function
    and the program WriteSector to select an unused sector on which testing can be done. If the
    physical parameter is true the address returned will refer to a physical sector.

int PicoXface::LogicalSectorsPerPhysicalSector(void);
    Returns: The size number of logical sectors per physical sector.

private: FLASHROM_ADDR PicoXface::MaxFreeSpace(FILE_TYPE type);
    Returns: The number of bytes in various categories:
    type == SECTOR_ELF      return total unused space.
    type == SECTOR_UNUSABLE return free space that is located between two files.
    type == SECTOR_BIT      return largest contiguous allocation.
```

## 4.14 GetFlashSpecs

```
cGString PicoXface::GetFlashSpecs (void );
```

Returns: A formatted containing information about the Pico Card's flash ROM.

## 4.15 GetNextFile

Functions in this group provide a mechanism to step through the files on the flash ROM system one at a time.

```
cGString PicoXface::GetNextFile (int *tokenP, int *errP = NULL);
```

```
private: FLASHROM_ADDR PicoXface::GetNextFile
    (FLASHROM_ADDR addr);
```

Internal utility used to step through the flash ROM directory.

```
private: cGString* PicoXface::ListFileFlashNames(cGString dir, int *tokenP );
private: cGString* PicoXface::ListFileDates (cGString dir, int *tokenP );
private: cGString* PicoXface::ListFileNotes (cGString dir, int *tokenP );
private: int64_t* PicoXface::ListFileSizes (cGString dir, int *tokenP );
```

These functions get the first filename by calling with a token value of -1. For example, int myToken = -1; cGString f1 = GetNextFile(&myToken); Subsequent filenames are obtained by calling this without changing the token. A zero-length string returned indicates the end of the list, at which point the token is invalid. An error code is returned in \*errP. Despite the implication of tokens, this only supports one iteration at a time - that is, one valid token at a time.

Parameters:

tokenP a pointer to the token  
errP an optional pointer to the return error code

Returns:

the filename, or a zero-length string to finish the list

## 4.16 HasCapability

```
bool PicoXface::HasCapability
(uint32_t   cap,
 bool      refresh = false
);
```

This functions returns true if the FPGA image currently loaded has the specified capability.

Parameters:

cap must be one of the values in the following table:

refresh if true, this will query the current FPGA, otherwise the values will be returned from the cached values

PICO_CAP_FLASH	Image has logic to interface with flash ROM
PICO_CAP_KEYHOLE	PPC "keyhole" register and monitor.elf running.
PICO_CAP_BUSMASTERING	Image has bus mastering
PICO_CAP_ADC	Image has A/D and D/A
PICO_CAP_PIC	Image has PIC access
PICO_CAP_JTAG_SPY	Image has JTAG spy (it always has JTAG support)
PICO_CAP_ETHERNET	Image has Ethernet
PICO_CAP_HOBbled	E-12 is using configuration memory to set AER (hobbled mode).
PICO_CAP_KEYHOLE_RAW	raw keyhole hardware ignoring presence of monitor.elf
PICO_CAP_MONITOR	monitor is running.
PICO_CAP_TURBOLOADER	allows us to reload the fpga and get the current image (peekaboo)
PICO_CAP_UARTLITE	Image has uart lite port
PICO_CAP_GPIO	Image has GPIO support
PICO_CAP_MPORT_RAM	Image has multiport RAM support and is accessible directly from PC_host

## 4.17 IsOpen

```
bool PicoXface::IsOpen (void) const;
```

Returns true if there's an error-free connection to a driver adapter, and hence, a card.

Returns:

true if the current driver adapter reports no errors

## 4.18 JTAG operations

```
int PicoXface::JtagDetectDevices (void);
```

Returns the number of devices on the JTAG chain. Should always be three.

```
int PicoXface::JtagGetIdCode
(JTAG_SCANLOC scanLoc,
 uint32_t *idCodeP);
```

Returns the JTAG ID of the device a scanLoc 1, 2, 3,... on the JTAG chain. The device ID is 32bit value assigned cooperatively between the FPGA manufacturer and the JTAG (ie the political group).

```
private: int PicoXface::WriteReadJTAG
```

```

    (bool          tms,
     uint8_t      *dataInP,
     uint8_t      *tdoP,
     int          bitCount,
     int          dvcsBfor,
     int          dvcsAft,
     const char   *nameP="");

```

Write byte to the JTAG port and read back responses.

Parameters:

tms = true. dataInP points to TMS data,

tms = false. dataInP points to TDI data.

dataInP = array of bits to output on TDI (0x01 bit of the JTAG port) or TMS line (0x04 bit of the JTAG port).

tdoP = array to receive bits from JTAG line(0x10 bit of JTAG port) (NULL if not required)

bitCount= number of bits to transit read.

dvcsBfor=number of devices before the target device.

dvcsAft = number of devices after the target device.

The target device will be embedded in a sea of other devices. The function WriteReadJTAG presume that the non target devices are in bypass mode. In bypass mode a device emits a single bit when data is read, so that the bulk of the data returned is from the active (target) device. WriteReadJTAG automatically compensates for the position of the device in the chain.

Returns:

number of bits read, or a negative error code.

bits are packed in memory in the low end of each byte, eg. (bufP=0x01, bits=1) is a single one bit.

```

int PicoXface::JtagValid(void);
int PicoXface::JtagSpyStart(...);
int PicoXface::JtagSpyStop(...);

```

## 4.19 Keyhole Access

Functions in this group are used to prepare, read, and write the keyhole port.

```

protected: int PicoXface::KeyholeInitialize (bool enable=true);
    Initializes the keyhole and flushes any pending messages.

```

```

protected: int PicoXface::KeyholeReceive (char *bufP, int bufSize);
    Receives data into bufP to a maximum of bufSize characters. Returns -7155 if
    there is no data in the receive que.

```

```

protected: int PicoXface::KeyholeSend (char *bufP, int bufSize);
    Sends data over the keyhole port.

```

```

private: int PicoXface::GetRawKeyholeData (KEYHOLE_BUF *);
    Gets keyhole data in units of KEYHOLE_BUF.

```

```

private: int PicoXface::SendKeyholeCmd (KEYHOLE_BUF *pCodeP, int count);
    Receives keyhole data in units of KEYHOLE_BUF.

```

```

private: int PicoXface::BuildAndSendCmd (int cmd, const char *titleP, int
count,...);

```

This command generates a packet and sends it to the Pico Card.

Parameters:

cmd specifies the type of packet,

```

enum {KHC_LOOPBACK=0, //0 used for testing
      KHC_MISC, //1 used to set misc register
      KHC_VERSION, //2 retrieve version
      KHC_TIMEOUT, //3 set timeout of keyhole
      KHC_LOAD_ELF, //4 load program. u32=starting sector
      KHC_RAM_TEST, //5 ram test
      KHC_READ_MEM, //6 read memory at address specified in
param[0]
      KHC_FMTD, //7 start of formatted I/O
      KHC_ERR, //8 error, followed by error # & 32bit value.
      KHC_TEST, //9 miscellaneous test
      KHC_EXITTEST, //10 exit from test program
      KHC_DEBUG, //11 set debug from param[0] != 0
      KHC_WRITE_MEM, //12 param[0] = address, write upto 255 values
into memory (RAM)
      KHC_CALL_MEM, //13 call address at param[0]
      KHC_VERIFY_MEM, //14 verify data against memory
      KHC_CLEAR_MEM, //15
      KHC_DUMP_MEM, //16
      KHC_LOST, //17 data was lost between Pico card and
driver
      KHC_SINGLEBYTE, //18 single byte (carried in KH_END element)
      KHC_LASTCMD //19
};
titleP specifies a title for debugging output on the host
count specifies the number of parameters following the count parameter. If count is
negative the following parameter is an array of abs(count) elements.

```

## 4.20 Miscellaneous Functions

```

public: FLASHROM_ADDR (void);
PicoXface::GetCurrentImageAddr
public: bool PicoXface::CheckReadOnly (void);
public: int (int maybeThis=-1);
PicoXface::GetFirstFileSector
public: cGString (void);
PicoXface::GetMonitorVersion
public: FLASHROM_ADDR (bool=false);
PicoXface::GetPrimaryImageAddress
public: FLASHROM_ADDR (int second=0);
PicoXface::GetBackupImageAddress
public: int PicoXface::GetSectorNum (const char* fname);
public: int PicoXface::HowmanyFiles (FILE_TYPE type=SECTOR_FREE);
public: int (const char *fileNameP, uint8_t *bufP, bool
reverseBits);
PicoXface::IsValidBitFileForThisPart
public: bool (FLASHROM_ADDR);
PicoXface::LastFileSectorFromAddr
public: int (int adr, void *bufP, int sizeofBuf);
PicoXface::ReadAttributeMemory
public: int PicoXface::ReadRam (uint32_t *adrP, uint32_t *blockP, int
len);
public: int PicoXface::SetPicobase (cGString);
public: bool (void);
PicoXface::IsSimulationFile
public: bool (void);
PicoXface::ValidBackupImage
public: bool (void);
PicoXface::ValidPrimaryImage
public: int PicoXface::WriteRam (uint32_t adr, uint32_t *blockP, int len);

```

```

void          DeletePicoDrv          (PicoDrv *drvP);
void          DeletePicoDrvFile      (PicoDrv *drvP);
private: SECTOR_MAP*
PicoXface::Addr2SectorMap
private: int
PicoXface::AllocateFlashRomSpace    (FLASHROM_ADDR fileSize, int **sectorPP,
uint32_t options=0);
private: int PicoXface::CfiProbe     (void);
private: int PicoXface::CheckAllocation(int fileSize, const char *fullNameP,
uint32_t options);

private: int PicoXface::ClearSectorMap (void);
private: FLASHROM_ADDR
PicoXface::FindFile
private: cGString
PicoXface::SubstituteParams
private: int
PicoXface::WriteAttributeMemory     (int adr, void *bufP, int sizeofBuf);
private: PicoDrv
PicoXface::CreatePicoDrvFile
private: PicoDrv
PicoXface::CreatePicoDrv             (int dvcNo=0, bool readOnly=false);

```

## 4.21 Multiple Pico Cards

```

uint32_t      AvailablePicoCards    (void);
uint32_t      CurrentPicoCard       (void);

```

these functions return a bit array specifying which Pico Cards are available to the host. Only one card can be accessed at a time. The Pico Cards are known by the names \\.\Pico<i> to Windows where ii is a number in the range 1-32. The value returned by these functions is a bit mask of the current or available Pico Cards. bit 1 << (i-1) will be set if the corresponding card is visible to the system.

## 4.22 Read Flash File

The functions in the group read data from the flash ROM on the Pico Card.

```

int PicoXface::ReadFlash
(FLASHROM_ADDR addr,
void *bufP,
int size
);

int PicoXface::ReadFile
(FLASHROM_ADDR sector,
const char *bitFileNameP,
char **notesP,
uint8_t **dataPP);

int PicoXface::ReadFlash
(FLASHROM_ADDR address,
void* blockP,
int len);

private: int PicoXface::VerifyFlashAgainstFile
(uint32_t start_sector,
const char *fileNameP,
int flashSize,
bool quiet=false);

```

Read data from the card's flash ROM.

Parameters:

addr	starting address in the flash ROM
bufP	destination buffer
size	number of bytes to read

Returns:

number of bytes read, or a negative error code

## 4.23 Serial Number

The functions in this group read or write the serial number and other information to the permanent (SecSci) sector of the flash ROM.

```
int PicoXface::ReadSerialNumber
    (uint64_t    *serialP,
     cGString    &fmttedSerialCs,
     cGString    &partNoCs,
     bool        *lockedP
    );
```

Read the card's serial number and FPGA part number.

```
int PicoXface::WriteSerialNumber
    (cGString    fmttedSerialCs,
     cGString    partNoCs);
```

Write the card's serial number and FPGA part number (used during manufacturing only).

Parameters:

serialP	pointer to an 8-byte buffer (or u64) where the serial number will be stored.
fmttedSerialCs	reference to a string that will receive a formatted representation of the serial number
partNoCs	reference to a string that will receive the part number of the FPGA.
lockedP	pointer to a bool field which will be set to one if the secure sector on the flash ROM is locked.

Returns:

1	= serial number is present in secure sector (as it should be)
2	= serial number derived from JTAG interface,
< 0	= error code.

## 4.24 Reboot

Functions in this group will restart the FPGA with the specified FPGA file (on flash ROM) and may also initiate any ELF file linked to the bit file.

```
int PicoXface::Reboot
    (FLASHROM_ADDR    fileAddr,
     bool              runLinked = true,
     KEYHOLE_MODE     enable=KEYHOLE_UNCHANGED
    );
```

```
int PicoXface::Reboot
    (const char        *fileNameP,
     bool              runLinked = true,
```

```

        KEYHOLE_MODE          enable=KEYHOLE_UNCHANGED
    ) ;

```

Boots the FPGA program with the specified address in flash ROM.

Parameters:

`addr` is address of file in the FLASHROM file system. Some special cases for particular effects:

- `addr = BOOT_POWER_CYCLE (0x4002)` function does a hard reboot (power-down, power-up).
- `addr = BOOT_PRIMARY_FAST (0x4001)` tries an ordinary Primary boot, if this fails a hard reboot.

`fileNameP` name of the FPGA program to boot (see note)

`runLinked` if true, run the ELF file linked to the boot image on the card: see [ChangeFileProperties](#).

Returns:

flash ROM address of newly-loaded FPGA program, or a negative error code.

Notes:

1. On Pico cards which have flash ROM (E-12, E-14, and E-15) the filename or fileAddr refers to a file within the flash ROM file system. Upon successful completion the value returned is the flash ROM address of the file
2. For Pico cards which do not have flash ROM the file name refers to a file on the host system. The call which specifies a file address is not valid. Upon successful completion the value returned is zero.

## 4.25 RegisterCallbacks

The functions in this group provide a mechanism for the calling program to closely monitor the operations performed by Pico.dll.

```

void PicoXface::RegisterEventCallback
    (int(*eventP)(int erCode, uint32_t param, const char *msgP));

```

The event callback is called at various times with the specified error code, supporting parameter, and ascii message.

```

void PicoXface::RegisterPrintf
    (int (*printfP)(uint32_t color, const char *fmtP,...), COLOR_SCHEME
 *colorSchemeP=NULL);

```

The printfP callback specifies a routine which will output text in the callers display space. The first parameter of the callback specifies the color, the second and subsequent parameters are the format and parameters to the format in the style of printf. NULL parameters reset to default values, ie, No output, and color=black.

```

void PicoXface::RegisterProgressCallBack
    (int (*callbackP)(int oddometer, bool setLimit, const char *));

```

Set the callback function used for signalling an event. The function parameter to RegisterEventCallback has itself three parameters: code, subcode, and msgP. code is the basic (error) code. subCode is the parameter to the basic code. msgP is a string containing descriptive information. It may be NULL.

```

private: void PicoXface::RegisterAudit
    (void(*auditP)(const char *msgP));

```

This function is used for debugging only. It provides a mechanism for logging ascii debug information.

## 4.26 Run Program

Functions in this group load programs into the RAM on the Pico Card and start executing them.

```
int PicoXface::RunFromFlash (const char *pcFileNameP, bool
loadLinkedFile=true);
    Run the specified ELF program from the flash ROM on the Pico Card. If
loadLinkedFile is true and the specified file has a bit file linked to it,
that file will be loaded before the ELF file is loaded.

int PicoXface::RunFromPC (const char *pcFileNameP, uint32_t flags=0);
    Run the specified file from the host disk system.
```

## 4.27 Save and Restore whole flash ROM

Functions in this section read or write the entire flash ROM to a file on the host disk system.

```
int PicoXface::RestoreBySector (const char *fileNameP);
int PicoXface::SaveBySector (const char *fileNameP);
```

## 4.28 SelfTest

```
int PicoXface::SelfTest
(COLOR_SCHEME colorsP,
 uint32_t testMask = 0xFFFFFFFF
);
```

This function initiates the internal self test of the Pico Card. The tests are specified by the following enum:

```
enum {SELFTEST_AER                SELFTEST_FLASH_STATUS    SELFTEST_CONFIG
      SELFTEST_FLASH_ERASE        SELFTEST_REBOOT        SELFTEST_CHECK_JTAG
      SELFTEST_JTAG_IDCODE        SELFTEST_IGNORE_ERRS    SELFTEST_STOPON_ERR
};
```

```
protected: void PicoXface::CheckReadable (void);
protected: void PicoXface::CheckWriteable (void);
```

These functions perform a single read between the host and the Pico Cad. They are used by the installation program.

## 4.29 Service Functions

```
void PicoXface::PrintBuf (const char *bufP);

private: int PicoXface::AnalyzeCommand (cGString cmd, KEYHOLE_BUF
**pCodePP, int64_t *valueP);
private: int PicoXface::EvaluateExpression (const char *icmdP, int len,
int64_t *valueP, cGString &resultStringCs);

private: int PicoXface::GetFileContent (const char *fileNameP,
uint8_t **dataPP);
```

```
private: cGString PicoXface:FileNameOnly          (const char
*fullFileNameP); //ie without directory etc
```

## 4.30 Transferring data between th PC and the Pico Card

Functions in this section transfer data to or from the host PC to/from a Pico Card.

```
int          (FLASHROM_ADDR addr, void *bufP, int      all cards
PicoXface::ReadDev byteCount);
ice
int          (FLASHROM_ADDR addr, const void *bufP,   all cards
PicoXface::WriteDe int byteCount);
vice
int          (int portAdr addr, const void *bufP, int E12/14/15
PicoXface::ReadPor byteCount);
ts
int          (int portAdr addr, const void *bufP, int E12/14/15
PicoXface::WritePo byteCount);
rts
int          (int *addrP, void *bufP, int byteCount); E14/15, E12
PicoXface::ReadRam          partial
int          (int addr, const void *bufP, int      E14/15, E12
PicoXface::WriteRa byteCount);          partial
m
```

Parameters:

addr	starting address in the memory mapped address space on the Pico Card.
portAdr	port address
bufP	bufP source / destination buffer
byteCount	number of bytes to read / write

Returns: number of bytes written, or a negative error code.

When the host PC wishes to transfer information to a Pico Card there are four different mechanisms:

1. Directly to the firmware using a memory mapped address.
2. Directly to the firmware using an I/O port.
3. Directly to the SDRAM or DDR2 RAM on the Pico Card.
4. Using Bus Mastering transactions. (see [Pico Chanel](#)s).

### Read/Write Device

The functions presume the firmware loaded on the FPGA is configured to respond to the particular address. The address is on address from the perspective of the Pico Card. All the translation between host-PC absolute addresses is performed by pico.sys. Direct access to the FPGA is by far the fastest method by which data can be transferred between the Pico Card and the host-pc.

### Read/Write Ports

These functions provide access to the I/O ports fabricated in the Pico Card FPGA. These ports are typically used by Pico Computing only so that the maintenance functions will not detract from the user address space. The port address should be a value between 0 and 256 and must be x4 hex or xC hex. If byteCount is one the operation will be performed as a byte I/O. If byteCount is a multiple of 2 or if the card is a Pico E-12 the operation will be performed as a 16bit word I/O. If byteCount is a multiple of 4 the operation will be performed as a 32bit I/O.

### Read/Write RAM

These functions enable data to be transferred directly to and from the RAM on the Pico Cards. On the E14/15 the DDR2 RAM is supported with a multi port interface. This allows direct access from the

host-PC. On the E-12 this function is performed by the PPC running on the Pico Card.

## 4.31 Version

Functions in this groups manipulate version information.

```
int PicoXface::PicoUnmakeVersion
    (int ver=-1,
     int *majorP=NULL,
     int *minorP=NULL,
     int *revisionP=NULL,
     int *counterP=NULL);
```

This function Unpack version information.

Nominally this is packed in each byte of ver. If ver == -1, this function returns the current version of the dll. Any of the int\* parameters may be null, which is the default. For example: ver = PicoUnmakeVersion() <class\_pico\_xface.html#a35>; returns current version PicoUnMakeVersion(ver, &major, &minor); returns major and minor version information PicoUnMakeVersion(ver, &major, &minor, &revision, &counter); returns all version information.

```
int PicoXface::VersionCompatibility
    (pico_version_t firmwareVersion,
     pico_version_t softwareVersion,
     int erC,
     int =0);
```

This function determines whether the firmwareVersion and softwareVersion are compatible. It is presumed that the software will be all of the same version. However, the firmware might be updated on a different schedule that the software or have different backwards compatibility requirements.

```
private: cGString PicoXface::FormatVersion (void);
```

This function returns a formatted ascii string containing the version information for Pico.dll.

Versions are numbered with four decimal digits:

**major.minor.release.counter**

for example: **3.3.0.6**

All Pico software and firmware uses the version number to ensure compatibility and in some cases to allow nominally incompatible versions to interoperate. The version number can be obtained in several ways:

- From Windows Explorer: right click the file and select Properties/Version.
- From the Help/About menu item in PicoUtil.
- From Pico.dll using the function GetProgramVersion().

When PicoUtil starts it will access the Pico Card and display version information. The following message is typical:

```
PicoUtil.exe: V3.3.0.06. Jul 21 2005 08:32:59.
Pico.dll: V3.3.0.06. Jul 21 2005 08:32:59.
Pico2K.sys: V3.3.0.06 Debug: Jul 20 2005 16:36:41.
Firmware: (FPGA) V3.3.0.6.
PPC: Pico Monitor V3.3.0.6.
```

The fields of this message have the following meanings:

- **PicoUtil.exe:** V3.3.0.06. Jul 21 2005 08:32:59  
This is the version and compilation date of PicoUtil.exe.
- **Pico.dll:** V3.3.0.06. Jul 21 2005 08:32:59.  
This is the version number of the DLL.
- **Pico2K.sys:** V3.3.0.06 Debug: Jul 20 2005 16:36:41.  
This is the version and compile date of the driver.
- **Booted with V3.1.7.bit**  
This is the name of the bit image file currently loaded into the FPGA.
- **Firmware:** (FPGA) V3.1.7.0.  
This is the version of the current FPGA bit image
- **PPC:** Pico Monitor V3.1.7.0.  
This is the version of the program running on the PPCb on the Pico Card. This version information may not always be present if the Pico Card is a LO version (logic only), or if the monitor program is not running.

The software will verify versions at startup and not operate if the versions are not compatible.

Version numbers are managed according to the following rules:

### 3.3.0.6

- Versions with a different major version number can be expected to be highly incompatible and will require changes to user application code.

### 3.3.0.6

- Versions with the same major version numbers but different minor version numbers (eg. 3.2 to 3.3) should be compatible at the source code level. They may require the code to be recompiled.

### 3.3.0.6

- Versions with the same major and minor version numbers but which differ in the release number will be compatible at the object code level - ie a version 3.3.0.x PicoUtil.exe should run with a version 3.3.8.x version of Pico.dll. Naturally, improvements in a later version number will not be present in the earlier code.

### 3.3.0.6

- The counter is a number that is used to ensure that no two instances of different software or firmware are released with the same numbers. Code with different counters should be compatible across the board.

## 4.32 Write Flash File

The functions in this section write files to the flash ROM. The functions take some care not to erase a sector if it is not necessary, and they will randomize the location of files to spread out the usage of the flash ROM. However, it is usually necessary to erase the sector before writing and the erase operation is quite slow.

```

int PicoXface::WriteFile
    (const uint8_t    *dataInP,
     int              dataSize,
     const char      *pcFileNameP,
     const char      *flashNameP,
     const char      *noteP = NULL,
     const char      *linkedFileNameP = NULL,
     uint32_t        writeOptions,
     int              *preallocatedSpaceP = NULL
    );

```

This function creates a new file.

Parameters:

dataInP	a pointer to the data to be written to the new file. This field can be NULL in which case the data will be read from the pcFileNameP
dataSize	size of the data in dataInP.
pcFileNameP	name of the file on the host. This field as stored in the header of the flash file unless it is NULL.
flashNameP	the name of the file in the flash ROM. This name must be unique, although there are options which allow the flashNameP to be automatically renamed to preserve this requirement. The flashNameP may include \ or / characters to signify a directory structure. In the present implementation this is accepted has no meaning other than to make the flashNameP unique. The length of this field is limited by the size of the header (nominally 256 bytes minus the other information stored in the header).
noteP	ascii information stored in the header to describe the file. The length of this field is limited by the size of the header (nominally 256 bytes minus the other information stored in the header).
linkedFileNameP	this field links the file to another file on the flash ROM. If the flashNameP is a bit file the linkedFileNameP will usually be an elf file. If the flashNameP is an elf file the linkedFileNameP will always be a bit file. The length of this field is limited by the size of the header (nominally 256 bytes minus the other information stored in the header).
writeOptions	This field contains a number of bits selected from the following enum:
enum	{WRITE_BACKUP, WRITE_PRIMARY, WRITE_EARLIEST, WRITE_ALLOWDUPS, WRITE_CONTIGUOUS, WRITE_OVER, WRITE_BOOT2, WRITE_BOOT1, WRITE_IGNORE_PART};
WRITE_BACKUP	This bit is set if the PrimaryBoot.bit is being written and the user wishes the system to automatically write a BackupBoot.bit file immediately following the PrimaryBoot.bit file.
WRITE_PRIMARY	This bit indicates that the file being written is the primary boot file. In this case the flashNameP will always be PrimaryBoot.bit and the file will be allocated beginning at flash ROM address 0x4000.
WRITE_EARLIEST	This bit causes the file to be allocated on the earliest available sector. If this bit is reset the file will be written at a random location in the flash ROM.
WRITE_ALLOWDUPS	This bit allows the flashNameP to conflict with another file name in the flash ROM system. However, Pico.dll will automatically add digits to the end of the file to make the flashNameP unique.
WRITE_CONTIGUOUS	This bit will cause the file to be written on contiguous sectors. This is set automatically by Pico.dll based upon the type of file being written. Bit files are always written contiguously, all other files are written in linked logical sectors.
WRITE_OVER	This bit allows Pico.dll to write a file with the

same name by erasing the old file. This operation is not equivalent to the update operation since the file in question will be reallocated when it is written.

WRITE\_BOOT1 This bit will cause Pico.dll to boot the FPGA file being written after the write is successful.

WRITE\_BOOT2 This bit will cause Pico.dll to reboot the Pico Card with the primary boot before writing the file. This options is useful when user FGPA files are being written to the file which may not always be capable of accessing the flash ROM. The reboot operation will attempt to reboot the Pico Card with PrimaryBoot.bit. If this fails Pico.dll will perform a hard reboot (ie power-down and power-up the Pico Card)

WRITE\_IGNORE\_PART This bit defeats the normal part number check which is performed when an FPGA file is written to a Pico Card. The part number check prevent a FPGA file being written to the flash ROM which cannot be loaded into the FPGA on the Pico Card. This option might be useful if the part number for which the FPGA image was prepared is for a slower speed grade than the part on the Pico Card.

sectorsP not a public parameter. Please use the default value of NULL

```
int PicoXface::WriteFlashDeluxe
(FLASHROM_ADDR addr,
 const void *dataP,
 int dataSize);

int PicoXface::UpdateFile
(FLASHROM_ADDR addr,
 const uint8_t *fileDataP=NULL,
 int fileSize=0x55555555,
 int romSize=0xAAAAAAAA);

private: int PicoXface::UpdateFile
(const char *fileNameP,
 const uint8_t *fileDataP=NULL,
 int fileSize=0x55555555,
 int romSize=0xAAAAAAAA,
 FLASHROM_ADDR addr,
 const char *noteP,
 const char *flashNameP,
 const char *pcFileNameP,
 bool allowSmaller
);
```

The first function is the primary update function. Update writes new data over the top of existing data. In other words it uses the same sector allocation as the original file. This function is used in circumstances in which the flash ROM has been carefully laid out and the update operation is simply updating a the file. However, if an attempt to write a file that is larger than the original file (strictly speaking requires more sectors) the update function will reallocate the file to a new area of flash ROM (ie defeat the very purpose of the update operation). If you must retain more precise control of the Update operation, use the second form of the Update function. This function returns an error if the file is not exactly the same sector size as the original.

#### Parameters:

dataInP file data, or NULL to read data from pcFileNameP  
fileSize size of dataInP, in bytes  
romSize size of data on the flash ROM  
pcFileNameP name of file to copy to card, or NULL if data is given in dataInP  
flashNameP name of the file on the card, or NULL to use the filename from pcFileNameP  
noteP optional note for the file. May be NULL.  
linkedFileNameP name of "linked" file

## 5 Class cCIS

This class provides access to the CIS (Card Information Structure) on the Pico Card. The CIS is a binary structure used to describe the host resources required by the Pico Card. Normally, the CIS is provided by Pico Computing and does not need to be changed. The CIS is generated by the core firmware of the Pico Card and is made available to the operating system when the card is first powered on.

source file:

```
c:\pico\software\source\cis.cpp
```

header file:

```
c:\pico\software\include\cis.h
```

### 5.1 CIS functions

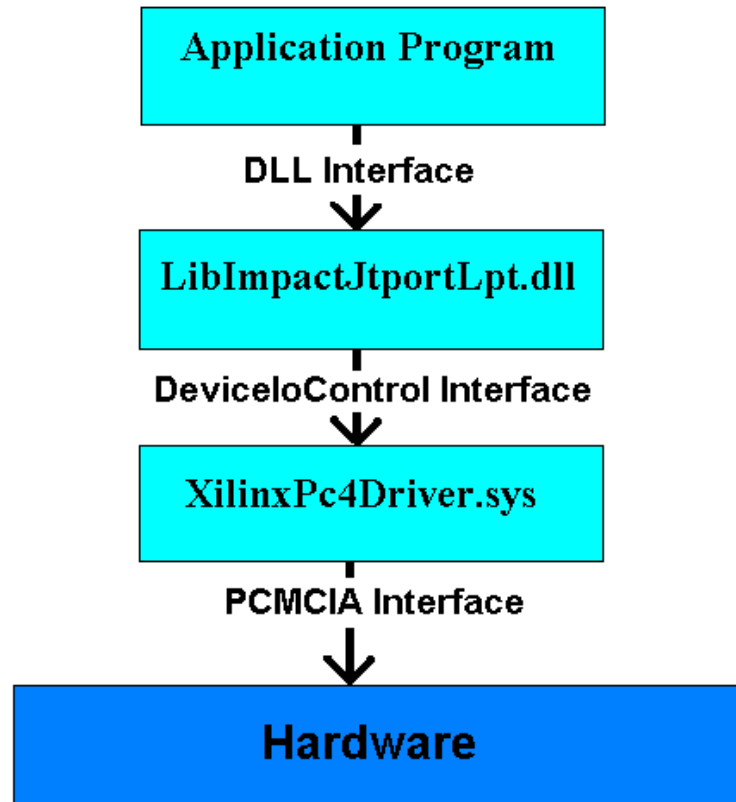
This class provides access to the CIS (Card Information Structure) on the Pico Card. The CIS is a binary structure used to describe the host resources required by the Pico Card. Normally, the CIS is provided by Pico Computing and does not need to be changed. The CIS is generated by the core firmware of the Pico Card and is made available to the operating system when the card is first powered on.

```
cCIS:cCIS(PicoXface*);
int cCIS::CheckCIS
    (const char    *fileNameP,
     const char    *outFileNameP,
     uint8_t       **dataPP,
     cGString      &interP,
     int           *nonFatalErrorP);
int cCIS::EncodeCIS
    (void          *dataP,
     int           dataSize,
     const char    *fileNameP,
     int           *nonFatalErrorP=NULL);
int cCIS::InterpretCIS
    (void          *u8P,
     int           size,
     cGString      &outputCs);
int cCIS::ReadAndInterpretCIS
    (uint8_t **bufPP,
     cGString &tupleStr,
     bool fromImage);
cGString cCIS::NameofTuple (uint8_t type);
int cCIS::ReadOriginalCIS (void *bufP, int bufSize);
```

## 6 cJTAG class members

The cJTAG class is designed to interface with the JTAG controller built into the Pico Cards. JTAG is an acronym for 'Joint Test Action Group' the technical subcommittee that create IEEE 1149.1. This is a bit serial interface allows test equipment to interface directly to modern hardware chips for testing and initialization. On the Pico Card the JTAG port is fabricated in PrimaryBoot.bit and can be accessed directly as a parallel port driver over the PCMCIA / Cardbus interface. The normal operating system drivers (parport.sys under Windows) interfaces this parallel port so that it appears as a standard LPT driver. Third party software such as Xilinx EDK, or Xilinx Impact access this parallel port on the PC host. Typically a special purpose driver is insinuated between the application software and the parallel

port drivers since there is no direct way under Windows to control a parallel port. The following software stack is typical:



**Typical JTAG software Stack**

The essence of the JTAG interface is a mechanism to read or write specific registers on a particular hardware device. A register called an **Instruction Register (IR)** is first programmed to indicate which **Data Register (DR)** is to be read or written. Some information can be gleaned by examining the responses to setting the Instruction Registers only (see [DetectDevices](#)). Other information is obtained from Data Register specified by the Instruction Register. Both the Instruction register and the Data Register are variable in length. Access to the JTAG controller is mediated by a finite state machine embedded on the chip.

Software which accesses a JTAG equipped device will normally read a .bsd or .bsd1 file which contains a detailed description of the Instruction Registers on a particular device.

Software using the PicoXface class can access the JTAG port directly.

The cJTAG class uses the enum **TAP\_STATE** for to enumerate the states through which the finite state machine on the JTAG controller passes.

The cJTAG class uses the enum **JTAG\_SCANLOC** to signal which device is being accessed. On a Pico Card the first device in the chain is always the FPGA, the second device is the CPLD, and the third is the Ethernet Controller. These are enumerated in **JTAG\_SCANLOC**.

The enum **JTAG\_REG** defines various registers within all devices. This information would normally be obtained from a bsd file, however, on the Pico Card it is more predictable.

## 6.1 DetectDevices

```
int DetectDevices(uint32_t *idP, int idCount);
```

This function returns the number of devices on the JTAG chain and fills the uint32\_t array idP[] with the device ID's detected. On all Pico Cards there are three devices:

- The FPGA
- A CPLD or PIC
- An Ethernet chip

The DetectDevices function will determine the number of devices without any knowledge of the internal commands accepted by the devices.

### Example:

```
uint32_t ids[3];
if ((erC=DetectDevices(ids, 3)) < 0)
    printf("Error %u\n", -erC)
else {printf("Number of devices=%u\n", erC);
      for (ii=0; ii < erC; ii++)printf("Device %u: %08X\n", ii, ids[ii]);
```

### Theory of Operation:

The DetectDevices function resets the JTAG controller on each device and then reads the IDcodes from through the DR channel of the JTAG controller.

## 6.2 Capture

```
int Capture(JTAG_SCANLOC scanLoc, JTAG_REG jtagReg, BYTE* tdiP, ULONG bitCount,
            BYTE *tdoP);
```

The Capture function programs the specified register on the specified device to the value specified in **tdiP** and returns the data read from the device in **tdoP**.

- **bitCount** is the number of bits in the JTAG register.
- **tdiP** may be NULL in which case the value written to the register is zero.
- **tdoP** may be NULL in which case the output data will not be returned.
- **Capture** returns zero or a negative number if an error occurs.

### Example:

```
if ((erC=Capture(
    JTAG_FPGA_SCANLOC, // device location == 1 (FPGA)
    JTAG_REG_IDCODE, // register within FPGA == 0x3C9
    NULL, // tdi = zeroes (ie a read operation)
    32, // number of bits in FPGA IDCODE register
    &idCode32)) < 0) // data variable to receive reply
    printf("Error %u\n", -erC);
```

### Theory of operation:

The JTAG device is driver to the captureDR state and the specified number of bits is output to the **Data Register**. Contemporaneous with this date from the JTAG device is harvested on the TDO line. and stored in tdoP. The JTAG device is returned to the idle state.

## 6.3 GotoTapState

```
int GotoTapState(TAP_STATE targetState);
```

This function drives the TMS (finite state machine) on the JTAG controller into the specified state. This function is used internally and always takes the shortest path to the specified state, except when the target state is XTAP\_STATE\_RESET. In the case of reset the function always outputs 6 ones on the TMS line.

Example:

```
if ((erC=GotoTapState(XTAP_STATE_IDLE)) < 0) printf("Error trnsitioning to
idle state: %u\n", -erC);
```

### Theory of Operation:

The necessary state transitions are shipped to the JTAG controller over the TMS line.

## 6.4 SetInstructionRegister

```
int SetInstructionReg(JTAG_SCANLOC scanLoc, ULONG cmd);
```

This function is an private function of the cJTAG class which loads the Instruction Register on the JTAG controller.

### Theory of Operation:

The JTAG device is driver to the shiftIR state and the specified **cmd** is loaded into the instruction register.

## 6.5 WriteReadJtag

```
int WriteReadJtag(bool tms, BYTE *tdiP, BYTE *tdoP, int bitCount, int dvcsBfor, int
dvcsAft, const char *descriptionP);
```

This function is a private function of the cJTAG class. It is the fundamental link to the entry in the device driver Pico.sys which marshalls the bitwise access to the JTAG port.

- **tms** True = dataP is to be output on the TMS line, false = dataP is to be output on the TDI line.
- **dataP** Data to be sent to the JTAG device TMS or TDI line.
- **tdoP** Data arena into which output from the JTAG device will be received.
- **bitCount** The number of bits in tdiP or tdoP.
- **dvcsBfor** The number of devices on the JTAG chain before the current device.
- **dvcsAft** The number of devices on the JTAG chain after the curretn device.
- **descriptionP** Description of the operation being performed and will be used by deugging output.

The driver Pico.sys automatically compensates for the position of the current device on the JTAG chain as defined by dvcsBfor and dvcsAft. dvcsAft zero's will be transmitted before the data bits. dvcsBfor zero's will be transmitted after the data bits. Consider a chain of six devices, three devices before the current device and two after.

```
totalDevices = 3+1+2 = 6. The bit stream will be:
 1 2 3 44444444 5 6: three bits for devices beforet, bitCount bits for current
device,
  ||||| ||||| ||| two bits for for devices after.
```

```

||||| ||||||| +++--- dvcsAft = 2 to ignore capturing bits 5 & 6
||||| ++++++--- bitCount for current device
+++++---dvcsBFor = 3 to append three extra bits after tdi data. */

```

The detailed operation of the JTAG port may be examined by setting the BUG\_DRIVER\_JTAG bit in Pico.sys. This can be done by calling:

```
PicoXface::SetDebugFlags(BUG_DRIVER_JTAG)
```

and using a tool such as DbgView.exe from [www.sysinternals.com](http://www.sysinternals.com).

## 6.6 JtagSpyStart / JtagSpyStop

```

int JtagSpyStart(void(*consumerP)(int device, int irBits, uint8_t *regP, uint8_t
*oldP, uint8_t *newP, int drBits),
                void(*rawPortsP)(bool dataB, UINT8 port) = NULL,
                void(*jbugOutP) (const char *fmtP, uint32_t ctrl, uint8_t
*u8P)=NULL,
                bool debug=false)

```

```
int JtagSpyStop(void);
```

`JtagSpyStart` initiates the capture of JTAG traffic. This operation depends:

- JTAG operating across the internal JTAG port of the Pico Card, and
- The card having the JTAG spy logic enabled (Pico E-14 only).

`JtagSpyStop` terminates the capture of JTAG traffic.

As noted in the [the introduction to the cJTAG class](#) third party software that depends upon a parallel port to access the JTAG controller may do using a simulated parallel port implement over the Cardbus / PCMCIA interface. `JtagSpyStart` enables the capture of this traffic by recording the accesses in an internal fifo. `JtagSpyStart` also initiates a separate thread to poll and analyse recorded data from the 2040 element fifo which capturing the data and status traffic to/from the parallel port. `JtagSpyStart` calls the consumer and rawPort functions as it analyze this fifo traffic. The two functions supplied to `JtagSpyStart` are:

- `void Consumer(int device, int irBits, uint8_t *regP, uint8_t *oldP, uint8_t *newP, int drBits);`  
This function is called when `JtagSpyStart()` uncovers an access to the JTAG port which reads or writes a data register.
- `void RawPorts(bool dataB, UINT8 port);`  
This function is called for every byte that was written to the data port (`dataB=true`) or every byte that was read from the status port (`dataB=false`).
- `void(*jbugOutP) (const char *fmtP, uint32_t ctrl, uint8_t *u8P);`  
This function is called when there is debugging information to display.

`JtagSpyStart` follows the state transitions on the JTAG device (`tmsState`) and captures `tms`, `tdi`, and `tdo`. This coupled with the information about the number of bits in each Instruction Register enables the software to determine values read from and written to each register. The consumer function is called when an Update-DR state is exited. At this point the:

- IR is defined (in `regP`) and is `irBits` long,
- the current **device** is determined by examining the `tms` bits and selecting the device which is not in bypass mode.
- `tdiP` will contain the new value for the register
- `tdoP` will contain the old value of the register
- `drBits` is the number of bits in the `tdi` and `tdo` data streams.

**NOTE:** This function is only available if the coresponding firmware has been instantiated on the FPGA

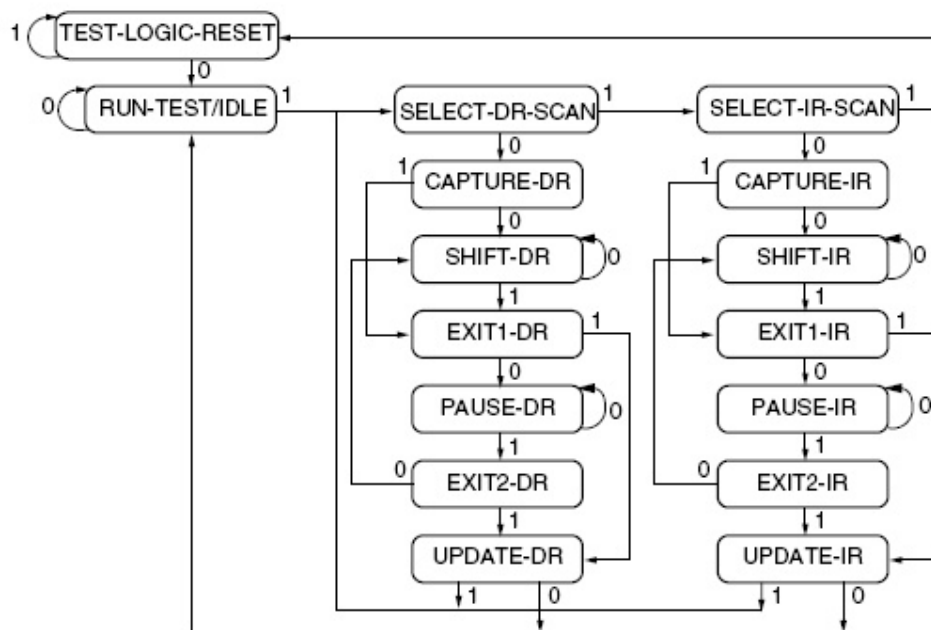
file and only on the E-14.

## 6.7 JTAG Theory of Operation

The parallel interface has three ports: data, status, and control. The outbound (data) port and the inbound (status) port are the only ones used by the JTAG interface. The four lines to the JTAG controller are:

- **TDI\_BIT** 0x01 bit of the output data port
- **TCK\_BIT** 0x02 bit of the output data port
- **TMS\_BIT** 0x04 bit of the output data port
- **TDO\_BIT** 0x10 bit of the input status port

The TMS\_BIT controls the finite state machine as follows:



NOTE: The value shown adjacent to each state transition in this figure represents the signal present at TMS at the time of a rising edge at TCK.

Shifting a pattern of all ones into the IR register and then updating the IR register will put each device in bypass mode. In this mode the device only outputs a single one bit when the capture-dr path through the finite state machine is entered.