Pico Computing
API Documentation
December 2011

Contents

1. Overview

Pico Computing Application Programmer Interfaces (APIs) provide the link between your application software running on a Windows or Linux computer and the hardware algorithm (or firmware) implemented in the FPGA.
The hardware-software interfaces provided by Pico include a number of software library functions which depend upon firmware components in the FPGA. The goal of these interfaces is to abstract away the details of communication to provide a platform-portable method of programming Pico Computing products.

Pico Computing FPGA platforms can interface with host computers using PCI Express or USB. Depending on the Pico platform you are using, the bandwidth and latencies of data transferred between host computer and FPGA may vary substantially. These variations depend on the electrical characteristics of the interface, for example the number of PCI Express lanes being used, but also on the nature of the application and the size of individual data elements being transferred. As an application programmer, you can have a significant impact on the final performance of these interfaces by learning and using appropriate APIs and methods for your target platform and application.

This manual describes the *PicoBus* and *PicoStream* interfaces. These are firmware concepts designed to move data to and from Pico Cards using the best available mechanisms - typically using Bus Mastering technology. The PicoBus interface provides a light weight interface between the PCIe infrastructure and the Pico cards. The PicoStream interface adds additional data buffering, making it better suited high data throughput. The software construct, Pico Channel, dovetails with these two firmware constructs.

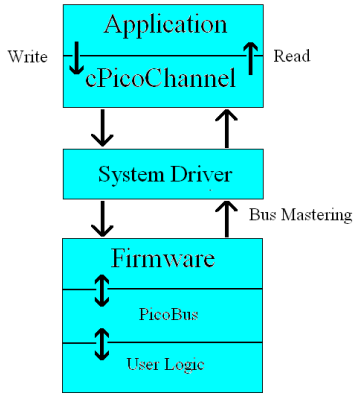
2. Architecture

The Pico Channel software API is modeled on file reads and writes. The firmware is modeled on a standard bus device (Picobus) or a streaming interface (PicoStream). The software end of the channel is a C++ class (cPicoChannel). The firmware end of a channel can be thought of as a peripheral device attached to the PicoBus or PicoStream, defined in the FPGA logic of the Pico Card. Pico channels provide the software machinery for Bus Mastering between the Pico Cards and a PC-host.

Pico streams are similar to Pico channels, with the addition of a FIFO buffer between the host PC and the firmware.
Both channels and streams use bus mastering to maximize throughput. Bus mastering is a process whereby the firmware takes temporary control of the PC bus. Bus mastering is attractive because it takes much of the burden off the processor and is significantly faster than single word transfers. Bus mastering is frequently referred to as DMA. Although this is technically incorrect, both strategies have a similar purpose - speed and independence from the processor. The terms bus mastering and DMA are used interchangeably throughout the Pico documentation.
The class cPicoChannel makes calls to the system driver, Pico32.sys, Pico64.sys, or CyUSB.sys (used only by the Pico E101 card). PicoXX.sys operates in its own address space and at the innermost ring of the operating system. PicoXX.sys is responsible for very low level functions such as buffer management, interrupt handling, and the scheduling of I/O calls to support the upper level classes.

PicoXX.sys delivers this data to a PCIe/USB endpoint. On the E16 and E101, these are on a dedicated, 3rd party chip. On the E17, M501 and M503, the PCIe endpoint is instantiated in firmware logic. Output from this core is sent into a DMA engine in firmware logic called the PicoBus; this is sometimes further translated into a PicoStream. Data in this format is made available to the *User Logic*, which is what you, the firmware developer, is to develop.

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

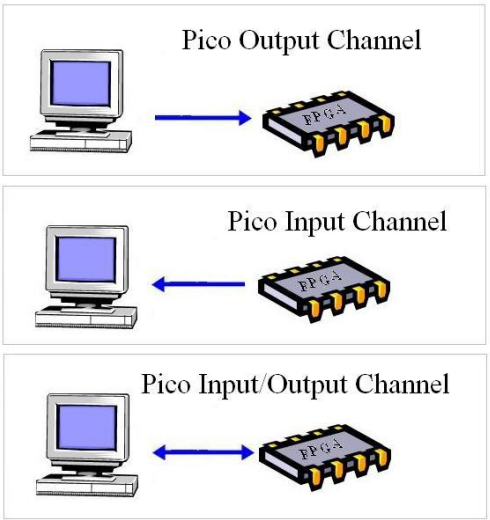


Channel Data flow

3. Channel Basic Concepts

The cPicoChannel is a C++ class that provides access to a Pico Computing card. cPicoChannel allows data to be transferred to and from the FPGA and can also be used for control purposes. For example you can load FPGA bit images using member functions of cPicoChannel. A channel can be used move data to & from memories on the card, and to directly access specific registers for device status.

The channel interface to all Pico cards is modeled on file reads and writes:
channel.Read(void *bufP, int size) is used to move data from the Pico card to the PC
channel.Write(void *bufP, int size) is used to move data from the PC to the Pico card.



The firmware on the Pico Card may be 32bit or 128bit based:

E16, E17, and E101 cards are 32bit based
The M501 and M503 cards are 128 bit based

The firmware on the Pico Card interfaces to the PicoBus using the following signals. These signals are referred to as the *PicoBus*:

picoAddr[31:0], picoDataIn[31:0], PicoDataOut[31:0], PicoRd, PicoWr, and PicoClk
or

picoAddr[31:0], picoDataIn[127:0], PicoDataOut[127:0], PicoRd, PicoWr, and PicoClk

A channel occupies a part of the memory mapped address space of the PC host. Data is moved to and from the PC-host using Bus Mastering memory transfers or single word transfers. The firmware implemented on the Pico Card side of the interface is 'memory mapped' - a range of addresses are assigned that are unique to the particular channel. The processing of data as it arrives or leaves the Pico Card is entirely the province of the firmware implemented on the Pico Card. The firmware may:

- Process the data as it is written from the PC or read to the PC.
- Store the data in a FIFO.
- Store the data in a RAM on the Pico Card for subsequent processing.

The particular firmware that handles data sent over the DMA interface is user defined, and referred to as a *user logic device,* as it exists in user logic. As data is transferred to and from the Pico card two independent addresses are being manipulated. The first is the physical address in the PC-host memory. This is entirely handled by PicoXX.sys and does not concern the application developer in the least. The second address is the address within the address space of the channel. This address is of fundamental concern to the developer. The address is referred to as the *Pico Address*. Necessarily, the Pico Address will be within the memory mapped range of the channel. As each 32-bit or 128-bit word is written to or read from the FPGA, both the physical address in the PC host memory and the Pico Address are incremented (by exactly four or sixteen). Other than this auto-increment property, as an application developer you have full control over the Pico Address and can use it for a large number of purposes. The Pico Address may be used to signal specific functions in the peripheral device - a simple and zero bandwidth way of communicating with the firmware.

The number of bytes written to or read from a channel is controlled by either the software or the firmware:
When a channel write or read is performed the number of bytes transferred is specified in the read / write call.
When picoXX.sys accesses the device the pacing registers (provided by the user logic devices) specify how many words can be transferred. Pacing registers are not strictly required in any particular user logic device, but without them the user must provide some alternate means of data flow control, as well as use separate API calls.

There are also a number of other properties of a channel, such as granularity and variable length timeouts, which grant the designer highly nuanced control over the operation of a channel.

3.1 Address Space of a Pico Channel

The address space [0x1000,0000 - 0x7FFF,FFFF] is dedicated to all channels on any one Pico card. You can create a user logic device that uses this entire address space. However, to support multiple devices simultaneously the following two limitations should be observed:

1. The address space used by the device should be limited to 0x10,0000 (1048576 bytes or one megabyte).
2. The beginning address of the address space should be 0x1010,0000 + n*0x10,0000 (n=1 thru 1791).
3. pacing registers must be provided at these addresses:
   0x1000,0000 + n * 0x10 (n=1 thru 1791) e.g. 0x10000010 is the read pacing register for channel 1

Pacing registers are used by PicoXX.sys to control the pace at which data is written to or read from the device.

Channels #1 thru #9 (0x1010,0000 - 0x10AF,FFFF) are used by various devices generated by Pico Computing. The remaining channel numbers are available to users.

The Pico Address (picoAddr) is the address supplied to the user logic. The value of picoAddr will always be within the memory mapped address space of the user logic device. If a DMA operation is specified that will cause the picoAddr to stray beyond the bounds of the channel, the operation will be split into two (or more) sub-operations and the picoAddr will be reset before the second sub-operation.

On each read or write operation the picoAddr is incremented. picoAddr is incremented by 4 (32bit architectures) or 16 (128bit architectures). The picoAddr value wraps around when it reaches the end of the range. For example:

```
cPicoChannel channel(12);
channel.Read(buf32P, 16);
```

On 32-bit architectures, command would retrieve:
word 0x10C00000, 0x10C00004, 0x10C00008, 0x10C0000C from the Pico Card
and store the results at
buf32P[0], buf32P[1], buf32P[2], buf32P[3]

On 128-bit architectures, command would retrieve:
word 0x10C00000
and store the results at
buf128P[0]

The next call of the same command would read 16 bytes (four words) would transfer word 0x10C00010 thru 0x10C0001F. Near the end of the address range a Read would transfer 0x10CFFFF8, 0x10CFFFFC, and then wraparound to 0x10C0000.

The steady progression of addresses passed to the Pico Card through DMA can be changed as follows:

```
channel.SetPicoAddr(0);
```

This would set the picoAddr back to the beginning of the range (ie 0x10C00000 for channel 12). Note that the value of picoAddr carries twenty bits of precision. In other words the picoAddr is relative to the memory mapped address space determined by the channel number. The system will automatically add the channel number into the uppermost bits of the actual address.

3.2 Channel Number Arithmetic

The channel number is used to specify which part of the address space the device occupies. Each channel occupies 0x100000 (1,048,576 bytes) bytes of addressable space within the available address range [0x1010,0000 - 0x7FFF,FFFF]. 16 bytes of space in the 0x100XXXXX space is also associated with the larger space. These are referred to as the *pacing registers*. The default channel space and pacing register space are defined as follows:

Channel	Low PicoAddr	High PicoAddr	Pacing Registers
1	0x10100000	0x101FFFFF	0x10000010 - 0x10000001F
2	0x10200000	0x102FFFFF	0x10000020 - 0x1000002F
n	0x10000000 + n * 0x100000	0x100FFFFF + n* 0x100000	0x10000000 + n*0x10
1791	0x7FF00000	0x7FFFFFFF	0x100007F0 - 0x100007FF

The following macros can translate channel number to raw channel address or raw pacing register address, or vice versa:

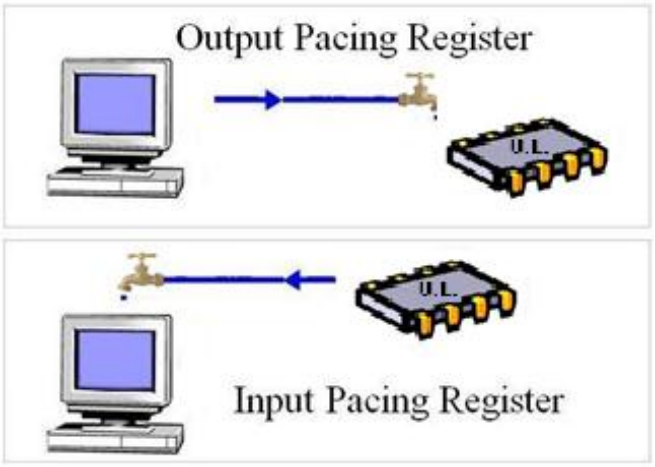
BM_CHANNEL_FROM_ADDR(addr),
BM_CHANNEL_FROM_STATUS(statusAddr),
BM_ADDR_FROM_CHANNEL(channel),
BM_STATUS_FROM_CHANNEL(channel)

The cPicoChannel class does the addressing arithmetic for you; hence, there is no need for an address parameter in channel.Read() or channel.Write()

The first two pacing registers are the *read pacing register* and the *write pacing register*, and are the only registers that Pico uses in their own user logic devices. Only the read side of these registers is used. The write side of the write pacing register is used to control the end-of-frame signal for Pico Streams.

3.3 Pacing Registers

Pico highly recommends all user logic devices implemented on Pico Cards implement two pacing registers which will be used by PicoXX.sys to limit how much data can be transferred to/from the device. If this is not done, software developers are obliged to use one of the other two entry points into the API.



The pacing registers are assigned in groups of 16 addresses (4 * 32bit registers). The pacing registers are assigned beginning at address: 0x10000,0010 user address space

Relative Address	Name	Direction	Width	Purpose
0	Read pacing register	R	32 (bits)	Control data flow for channel read operations
0	Not used	W	32 (bits)	n/a
4	Write pacing register	R	32 (bits)	Control data flow for channel write operations
4	Not used	W	32 (bits)	n/a
8	Not used	R/W	32 (bits)	n/a
12	Not used	R/W	32 (bits)	n/a

Read and write pacing registers have the following structure:

```
typedef struct
  {UINT32 wordsAvailable:20, //bits 0x000FFFFF
          nu             :6,  //bits 0x03F00000 == 0
          signature      :6;  //bits 0xFC000000 == 0x26 for read
                              //                 == 0x22 for write
  } BM_CHANNEL_STATUS;
```

In other words, the wordsAvailable field is the lower 20 bits, and the upper 6 bits is a signature, with 6 bits in the middle not used.

Each user logic device should set the wordsAvailable to the number of 32-bit words that may be transferred to/from the device. A value of zero means 'nothing available'.

3.4 Granularity

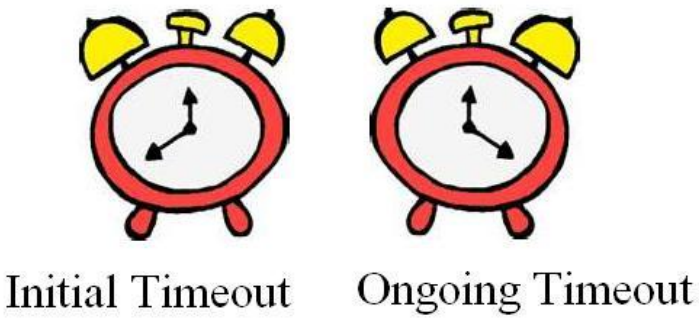
Granularity is used to avoid initiating a large number of very small I/O transactions. Granularity is a property of the channel that prevents an I/O transaction being initiated when the device does not have enough data to fill the minimum specified by granularity. A developer might want to control granularity to increase bandwidth, possibly at the expense of some increased data latency.

For example, consider a firmware application that must process data in full records of a given size, 100 bytes. The process of data in such an application will not commence until 100 bytes of data are advertised in the associated pacing register. In this case, a useful optimization might be to set the granularity to 100, so that no I/O will be initiated unless at least 100 bytes are available. This simple optimization can greatly increase performance by reducing the total number of bus transactions.

Note: if the number of bytes requested in the channel.Read / Write is less than granularity, the driver will ignore the granularity restriction.

3.5 Timeouts

Timeouts are used to limit how long a Read or Write operation will remain blocked. If a timeout other than zero is specified, the read/write will time-out after the specified time whether the I/O operation has completed or not. In these cases the number of bytes returned by the Read/Write function may be less than the number requested. A value of zero returned by Read/Write indicates that no data was transferred. The error condition created by the timeout is reported by the GetErrors() function.



The timeout periods can be specified by setting the appropriate values in the structure **PICO_TIMEOUT**. Which is passed to **channel.SetGetOptions()**. There are two timeout fields:

PICO_TIMEOUT.initialTimeout. This value specifies the timeout for the first block transferred. Any device may take some time before it can respond to a request. TimeOut1 will be set according to the behaviour of the firmware device.
PICO_TIMEOUT.ongoingTimeout. This value specifies the timeout for the second and subsequent blocks transferred. Once the first byte is received data will often arrive in a burst. Timeout2 is usually significantly smaller than TimeOut1.

Either initialTimeout or ongoingTimeout may be zero, which will be interpreted as no timeout. In this case, PicoXX.sys will wait indefinitely for the channel to report ready. Timeout values are specified in units of 1 millisecond. The default values for the timeouts are initialTimeout=100 (ie 100 milliSeconds), and ongoingTimeout=50 (ie 50 milliSeconds).

When a transfer to/from the device is requested, using **channel.Read** or **channel.Write**, the pacing registers are read to determine whether the device is able to accept / provide the requested number of words. The number of bytesAvailable (from the pacing registers * 4) limits the size of data transfer according to the following set of rules:

bytes transferred = minimum of
bytesAvailable (from pacing register * 4),
bytes requested by user (from Read or Write request),
bytes to end of channel address space (derived from picoAddr and channel size),
other limitations based upon the hardware.

If the minimum value from the above rules is larger than the granularity specified for the channel, then the I/O is initiated. Otherwise the channel will wait and possibly timeout.

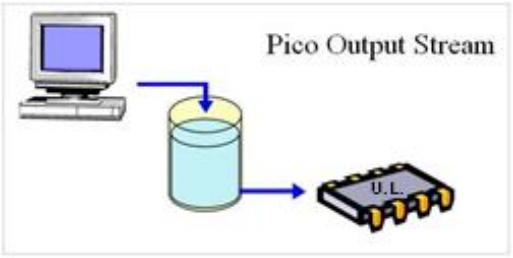
The function **channel.BytesAvailable()** may be called to report the pacing register value to the software.

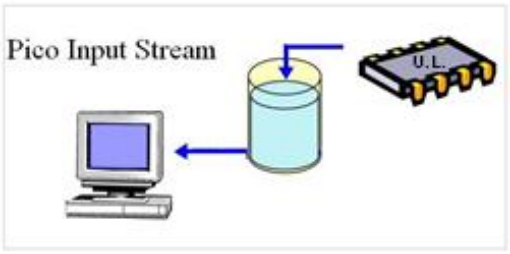
Multiple read or multiple writes should not be outstanding against a channel. This will transfer data in an unpredictable fashion. However, reads and writes can be overlapped. The flow of data to/from the device is then completely dictated by the status returned by the device status registers.

3.6 Pico Streams

Pico Streams are identical to Pico Channels except streams have a FIFO buffer between the PCIe/USB endpoint (and the entire host system) and the user logic. This FIFO isolates the stream from fluctuations in the rate of delivery of data to/from the host PC. On the other hand, the Pico Stream eliminates any concept of the PicoAddress, instead creating a single port to which data is written, or from which it is read.

Conceptually a Pico Stream is built as follows:





4. Software

Pico_channel.h contains the abstract class cPicoChannel and multiple derived classes. Each derived class is specific to a particular Pico cards. In general there is a particular class for each Pico card (except for the legacy E-12, E14, and E-15 cards which are handled by the cPicoChannel_Exx class). The class structure is as follows:

Class	Purpose	Implementation File
cPicoChannel	fundamental user access class	source\pico_channel.cpp
cPicoAbstractChannel	abstract class for the following sub-classes	none
cPicoChannel_E16	derived class supporting E-16 card	source\pico_channel_E16.cpp
cPico_Channel_E17	derived class supporting E-17 card	source\pico_channel_E17.cpp
cPicoChannel_E101	derived class supporting E-101 card	source\pico_channel_E101.cpp
cPicoChannel_M501	derived class supporting M501 and M503 cards	source\pico_channel_E501.cpp
cPico_channel_IP	derived class supporting IP access	source\pico_channel_IP.cpp

4.1 The Class cPicoChannel

#include <pico_channel.h>

class cPicoChannel	
{public:	
int AvailablePicoCards	**(uint8_t *availableP, int availableSize);**
cPicoChannel	**(int channelNo=10, const char *paramsP, cPicoChannel *parentP);**
virtual ~cPicoChannel	**();**
int GetBytesAvailable	**(int regNum=-1);**
int GetError	**(char *bufP=NULL, int bufSize=0);**
int GetPicoConfig	**(PICO_CONFIG *configP);**
int GetStatistics	**(PICO_STATISTICS *statsP);**
HANDLE GetHandle	**(void);**
int Read	**(void *bufP, int byteCount, int picoRelAddr=-1);**
int Write	**(void *bufP, int byteCount, int picoRelAddr=-1);**
int SetPicoAddr	**(uint32_t picoAddr);**
int ReadDevice	**(int relAdr, void *bufP, int byteCount=4);**
int WriteDevice	**(int relAdr, void *bufP, int byteCount=4);**
int ReadDeviceAbsolute	**(int absAdr, void *bufP, int byteCount=4);**
int WriteDeviceAbsolute	**(int absAdr, void *bufP, int byteCount=4);**
int ReadMemory	**(int addr, void *bufP, int byteCount=4, uint32_t typeOfMemory);**
int WriteMemory	**(int addr, void *bufP, int byteCount=4, uint32_t typeOfMemory);**
int ReadStatus	**(int offset, void *bufP);**
int LoadFPGA	**(const char *bitFileNameP, uint8_t *dataP=NULL, int dataSize=0);**
int SetGetOptions	**(uint32_t optionCode, void *vP, int byteCount=4);**
int WriteControl	**(int offset, uint32_t ctrl);**
};	

NOTE: for clarity we are not describing cPicoAbstractChannel which is as an abstract class supporting individual classes for each Pico card.

4.1.1 Constructor and Destructor

The class cPicoChannel can be used to create a channel on the Pico Card. The member functions Read and Write can then be used to transfer data to and from the card. Refer to the **include\pico_channel.h** and **source\pico_channel.cpp.** Functions returns a non negative number if the operation is successful, otherwise they return a negative error code. The negative error code refers to codes in the file Pico_error.cpp and can be accessed by calling the functions InterpretError(), or FullError().

cPicoChannel(int channelNo, const char *parametersP=NULL, cPicoChannel *parentP=NULL)

This function creates a channel to the Pico card:

- Channel number is the channel used for communication between the host and the card. The value must be a number between 1 and 1791. Note that channels 1 through 10 are devices included by Pico Computing. The default value of 10 is the first channel that is not already used by Pico Computing.

- **parametersP** points to a string containing a series of keyword, or keyword=value, pairs providing additional requirements for the channel. Each possible field in this string is documented in the following sections. The parametersP string may contain $ variables:

Keyword	Meaning	Example
$(model)	Model name of pico Card	E101
$(picobase)	Value of environmental variable	c:\pico

  NOTE: The $model variable always resolves to the model of Pico card #1. If a bit file is specified it must be consistent with the model.

- **parentP** refers to an existing cPicoChannel. The new channel will have the same underlying Pico Card.

~cPicoChannel();

The function closes the channel and releases the underlying file object used to service the channel.

channelSize=

channelSize=value

This field of parametersP specifies the size of the pico channels. The default is 0x100000 bytes and is rarely changed.

File=

file=bitFileName
or
file=flash:bitfileName (E17 only)

This field of parametersP specifies the bit file to load into the FPGA as the channel is opened. The bitFileName refers to a file on the host system, or a file on the flashROM file system on an E17. The M501 it is possible to loaded from either source, but all other cards will load from either flashROM or the host system (see table below).

In absence of some other field in parametersP the bit file will be used to locate an appropriate model. For example if the bit file contains a header indicating that it is an LX240 FPGA image, then the first card with an LX240 FPGA will be used.

NOTE: The bit file will not be written to the flash automatically. If the Pico Card requires flashROM loading then the flash must be written before the channel is created, using the PicoFlash class, or PicoCommand (see the Getting Started document.

Model	FLASH	Considerations
E16	N	bitFileName always refers to a file on the host system. The card will be reported as not loaded until a bit file has been loaded
M503	Y	The flashROM is platform flash and is not accessible to the user. bitFileName always refers to a file on the host system. However, the card will be reported as loaded at power-on because the FPGA must have been loaded from platform flash and implement the PCIe interface in order to be visible to the operating system.
E17	Y	The flashROM is normal flash and is accessible to the user. A valid PrimaryBoot.bit file must be present at power-on in order for the operating system to recognize the PCIe end point. The syntax flash:bitFileName is equivalent to bitFileName and refers to a flashROM file.
M501	Y	The flashROM is normal flash and is accessible to the user. A valid PrimaryBoot.bit file must be present at power-on in order for the operating system to recognize a PCIe end point. The syntax flash:bitFileName is will boot from a flashROM file or the syntax bitFileName will refer to a host file.

ignorePacing
ignorePacing

This field of parametersP allows the driver to ignore the signature on the pacing registers. This option might be used when the FPGA image does not support the Pico Channel architecture, or when the FPGA is possibly not loaded and you are first establishing a connection to the card.

ipAdr=
ipAdr=hostname

This field of parametersP specifies the hostname or ip address of another machine which physically contains the Pico Card. The raget machine must be runnig PicoChannelServer and may be running PicoDaemon. See Access a Pico Channel Server.

!loaded
!loaded

This field of parametersP causes the FPGA to be loaded only if the same image is not already loaded.

Model=
model=<picomodel>

This field of parametersP specifies a basic or specific model of the Pico Card. Possible values for model are

Basic mode	Specific model
E16	E16LX50
E17	E17FX70T, E17SX50T
E101	E101LX45
M501	M501LX240
M503	M503LX240
For example:	
model=E17	specifies either E17FX or E17LX as acceptable
model=E17SX50T	specifies E17SX50T only as acceptable

picoCardNum=
picoCardNum=#

This field of parametersP specifies a particular FPGA within a cluster. The card number is assigned by the operating system according to an algorithm of its own. The absolute value cannot therefore be predicted. However, the number is consistent and stable during each boot. The number is reported in cPicoChannel::AvailableCards() and is displayed by picocommand -y, ie,

\\.\pico1
\\.\pico2
\\.\pico3
etc

The number refers the PCIe end points and not the physical Pico Card. For example, a Pico M501, E16, or E17, has a single end point for each card. The M503 occupies two physical slots on the EX-500 backplane and may implement either one or two end points. The M503 may appear as one or two picoCardnums's (with one being the most common).

If lieu of other fields specified in the parametersP string the cPicoChannel constructor will select the first unused picoCardNum.

readOnly or WriteOnly

readOnly

This field of the **parametersP** variable limits the channel to read only mode. In this mode the write pacing register is not required or verified

writeOnly

This field of the **parametersP** variable limits the channel to write only mode. The write pacing register is not required or verified.

Absent either of these fields, the channel is opened in read / write capable. As such the both pacing registers are required and will be verified (unless ignorePacing is specified).

serialNum=

serialNum=value

This field of parametersP specifies and exact serial number of the Pico Card. This should be unique.

A demonstrative example

```
cPicoChannel(10, "readOnly, picoCardNum=2, file=E16LX50-PrimaryBoot.bit);
```

4.1.2 Blocking Functions

The blocking functions are controlled by the pacing registers. They are blocking in the sense that the firmware may hold off the completion of the read / write functions as needed.

`int Read(void *bufP, ULONG byteCount, int picoRelAddr=-1);`

The Read function will transfer data from the device on the Pico card to the host PC. byteCount must be a multiple of four, however, it may be as large as the channel size. Read will return the number of bytes transferred which may be less than byteCount if the device timed out. Read will return a negative error code if an error (other than a timeout) occurs.

`int Write(void *bufP, ULONG byteCount, int picoRelAddr=-1);`

The Write function will transfer data from the host PC to the device fabricated on the Pico card. byteCount must be a multiple of four, however, it may be as large as the channel size. Write will return the number of bytes transferred which may be less than byteCount if the device timed out. Write will return a negative error code if an error (other than a timeout) occurs.

4.1.3 Non-Blocking Functions

The non blocking functions will access the hardware and return immediately.

`int GetBytesAvailable(int regNum);`

This function returns the value of the read pacing register (regnum =PICO_READ_PACING), the write pacing register (regnum=PICO_WRITE_PACING=4) or either of the other two pacing block registers, regnum=PICO_INTR_STATUS or PICO_INTR_CONTROL). Reads to the pacing read/write report the status of the pacing regisger minus any read / write requests that are outstanding on this user logic device. For example, if the firmware advertises that it can accept 100 bytes, but there is a Read pending that will consume 80 bytes, BytesAvailable() will return 20. If more bytes are committed than are advertised BytesAvailable will return zero. To obtain the current value of the firmware pacing registers use ReadStatus(). However, be aware that this value may not be the number of bytes that can be transferred without waiting. Note that the metric of the pacing registers is words but this function returns bytes.

`int ReadDevice(int relAdr, void *bufP, int byteCount);`

The ReadDevice function will transfer data from the device to the host PC. byteCount must be a multiple of four, however, it may be as large as memory available on the PC host. ReadDevice differs from Read in two

important respects: ~~first~~ it does not wait for the pacing register, and it does not use picoAddr (but rather the relative address passed as the first parameter).

`int WriteDevice(int relAdr, void *bufP, int byteCount);`

The WriteDevice function will transfer data from the device to the host PC. byteCount must be a multiple of four, however, it may be as large as memory available on the PC host. WriteDevice differs from Write in two important respects: it does not wait for the pacing register, and it does not use picoAddr (but rather the relative address passed as the first parameter). ReadDevice's address parameter is interpreted as an address relative to the beginning of the channel (anywhere from 0x10A00000 through 0x101FFFFF for channel 10), and will be a 20-bit number.

`int ReadDeviceAbsolute(int absAdr, void *bufP, int byteCount=4);`

The ReadDeviceAbsolute function will transfer data from the device to the host PC. byteCount must be a multiple of four, however, it may be as large as memory available on the PC host. ReadDeviceAbsolute differs from Read in two important respects: it does not wait for the pacing register and it does not use picoAddr (but rather the offset passed as the first parameter). ReadDeviceAbsolute's address parameter is interpreted as an absolute address (anywhere from 0x10100000 through 0x7FFFFFFF), and will be a 31-bit number.

`int WriteDeviceAbsolute(int absAdr, void *bufP, int byteCount=4);`

The ReadDeviceAbsolute function will transfer data from the device to the host PC. byteCount must be a multiple of four, however, it may be as large as memory available on the PC host. ReadDeviceAbsolute differs from Read in two important respects: it does not wait for the pacing register and it does not use picoAddr (but rather the offset passed as the first parameter). ReadDeviceAbsolute's address parameter is interpreted as an absolute address (anywhere from 0x10100000 through 0x7FFFFFFF), and will be a 31-bit number.

4.1.4 Memory Access Functions

Memory on Pico Cards may be of a variety of forms. All FPGAs have RAM elements in logic, and these can be assembled into a block (called Block RAMS). Most Pico Cards also have some external RAM. Pico Clusters may have additional neighborhood RAM, shared between all cards. Generally, the speed of access diminishes with distance from the FPGA and the capacity increases with distance.

`int ReadMemory(int addr, void *bufP, int byteCount=4, TYPEOF MEMORY typeOfMemory=0);`

The ReadMemory function will transfer data memory from the memory accessible to the Pico Card to memory to the provided memory block (bufP). byteCount must be a multiple of four, and is limitted to 0x10000 bytes.

`int WriteMemory(int addr, void *bufP, int byteCount=4, TYPEOF MEMORY typeOfMemory=0);`

The WriteMemory function will transfer data from host PC to memory on memory accessible to the Pico Card. byteCount must be a multiple of the firmwareWidth (see GetPicoConfig in Other Service Functions), and is limitted to 0x100000 bytes. The typeOfMemory parameter (if provided) must be one of the previously mentioned constants:

The TYPEOF_MEMORY variable noted in the previous two functions distinguishes the various classes of memory available on each FPGA. Some of these are real and others are memory mapped I/O. **TYPEOF_MEMORY** is defined in the following typedef.

Definition	Platform	Notes
typedef enum		
{BLOCKRAM_MEMORY=1,	Any	Equivalent to Read/WriteAbsoluteDevice

PORT_MEMORY,	E16, E17	Alternative address space access through the I/O address space. On the E-16 this Memory is the address space of the DEVICE_AMEMORY bank beginning at address 0xFFE00000.
STATIC_MEMORY,	E16, M503	This is a small amount of static memory packaged on the Pico Card. Static memory is often very fast and easy to access.
DDRx_MEMORY,	E17,M501, M503	DDR2 or DDR3 memories are usually large and require significant firmware to access.
ATTRIBUTE_MEMORY,	All	See configuration memory.
FLASH_MEMORY,	E17, M501	Read only access to flashROM on the Pico Card.
DEVICE_AMEMORY=16,	any	Primary device memory. Typically this is 0x100000 bytes long.
DEVICE_BMEMORY=17,	any	Secondary device memory. Typically this is 0x100000 bytes long and is rarely implemented in the FPGA.
DEVICE_CMEMORY=18,	any	Configuration memory. Typically 4096 bytes long and contains PCIe configuration Information along with other vendor specific or Pico Card specific registers.
DEVICE_PMEMORY=14,	any	Parent address space. The E-16 has a PCIe bridge device built onto the Pico Card. Certain functions (such as loading the E-16) require access to this address space. When a Pico Card is mounted on an M500 backplane the parent memory will refer to a bridge device on the M500 board. This is generally a pass through port and performs no useful function.
DEVICE_GMEMORY=15,	any	GrandParent address space. When a Pico Card is mounted in a M500 backplane, the PEX-8664 on the M500 is accessible through this memory address. There are many registers accessible through this memory space that control the card, for example loading the FPGA, or clearing errors on the PCIe.
} TYPEOF_MEMORY;		

Most of the special purpose memory (PMEMORY, GMEMORY, and PORT_MEMORY) is handled internally by the cPicoChannel class or by the underlying driver. These address spaces are exposed for debugging and special purpose functions.

4.1.5 Channel Options

Options are fixed values that alter the behavior of the channel over its entire life. Some options can be set when the channel is created (see Constructor & Destructor) and are fixed for the life of the channel. Other options can be changed dynamically.

Options are set or retrieved using a generic function:

int SetGetOptions(uint32_t paramCode, void *vP);

Unless otherwise noted the parameter vP points to a uint32_t which contains the new parameter. When the function returns *vP will be updated with the old value of the parameter.

Parameter Code		Default	Meaning
SET_DEVICESIZE	channelSize	0x100000	channelSize is the size of the memory space used by the device. The value should be between 0x100000 and

			0x100000 * 1791. PicoXX.sys will wrap the Pico Address so that it is always in the range [0, channelSize-1].
SET_BASEADDR	baseAddr		Changes the address of the memory range and in effect changes the channel number
SET_GRANULARITY	granularity	4	granularity is the number of words that must be present before aDMA transfer will be initiated. Default = 1.
SET_READ_TIMEOUT		100,50	Specify the timeouts associated with a read. Refer to Using Timeouts. vP should point to a PICO_TIMEOUT structure
SET_WRITE_TIMEOUT		100,50	Specify the timeouts associated with a write. Refer to Using Timeouts. vP should point to a PICO_TIMEOUT structure
SET_PICOADDR	picoAddr		Same as function SetPicoAddr()
SET_PICOADDR_STRATEGY	strategy		PS_ZERO_EACH_ACCESS cause picoAddress to be set to zero at the beginning of each read / write function. As data is read / written picoAddress will be incremented by 4. PS_IGNORE causes picoAddress to increment as each read / write function is performed.
SET_DEBUG_FLAGS	debugFlags		Specify debug flags to PicoXX.sys

4.1.6 Loading the FPGA

The FPGA can be loaded when the cPicoChannel class is created. However, there are occasions for re-loading or rebooting the FPGA after the class is created. This can be achieved using the LoadFpga function.

`int LoadFPGA(const char *bitFileNameP);`
This function loads the FPGA from a flash ROM file (E-17) or from a PC-Host file (E-16, M501, M503).

`bitFileNameP` points to an ascii string. The file name may have the prefix flash: which indicates that the file resides in the flash memory system of the E-17. Otherwise the file will be presumed to be on the host PC. See file= in the cPicoChannel constructor.

Examples:
LoadFPGA("c:\\Pico\\bin\E17FX70T-PicobusCounter.bit")
LoadFPGA("flash:$(picobase)\\bin\\M503-PicobusCounter.bit")
LoadFPGA("$(picobase)\\bin\\$(model)-PicobusCounter.bit")

In the second and third cases we have taken advantage of $(variables) to enhance the portability of the code.

4.1.7 Other Service Functions

int GetError();
This function returns the last error generated by cPicoChannel. Extended error information is available through the InterpretDriverError function.

HANDLE GetHandle();
This function returns the handle of the underlying file class used to implement the cPicoChannel. The most common use for this call is to obtain a handle which is then passed to DeviceIoControl. DeviceIoControl is used to perform special functions on the file class.

int SetPicoAddr(uint32_t picoAddr);
This function sets the address made available when the DMA transfers data to the device fabricated on the Pico card. The picoAddr may only be as large as the channelSize parameter (specified using SetGetOptions(SET_DEVICESIZE,..). SetPicoAddr() returns a negative value to indicate an error.

int ReadStatus(int offset, uint32_t *u32P);
This function reads a word from the pacing register of the user logic device. Offset == 0 is the read pacing register, offset == 4 is the write pacing register, offset == 8 and offset == 12 are user defined registers.

typedef struct
{uint32_t model; uint16_t openCount, cardNum;}
AVAILABLE_CARD;

int AvailablePicoCards (AVAILABLE_CARD *availableP=NULL, int availableSize=0);
Returns number of Pico cards visible to this application and fills availableP with references to available cards. The AVAILABLE_CARD has the following fields:

typedef struct

{uint32_t model;	//Model = 0x0E174658. Readers of the ascii persuasion will recognize 0x46 and 0x58 and the //code for 'F' and 'X'. Thus 0xE174658 is an E-17 FX card and 0x0E175358 is an E-17 SX card.
uint16_t cardNum;	//As in \\.\Pico3
uint8_t openCount;	//Number of users
uint8_t dead :1,	//Card is dead. The card has failed some significant internal test (such as PCIe failure) and cannot be used. This condition can normally be corrected by restarting the computer
loaded :1,	FPGA is loaded. Either the FPGA is loaded from flashROM at power-on or the FPGA has been explicitly loaded using the cPicoChannel constructor(...file=...) or cPicoChannel::LoadFPGA() function
remote :1,	Card is on remote machine. The Pico Card is physically located on another machine.
nu :5;	
uint32_t nu1[2];	
} AVAILABLE_CARD;	

4.2 Access to Pico Channel Server

The specification **ipAdr=hostname** in the constructor of a class will allow the Pico Channel to access Pico Card(s) on a network attached machine. For example:

channelP = new cPicoChannel(10, "writeOnly,ipAdr=1.2.3.4");

will open channel 10 on the machine at IP address 1.2.3.4.

Client IP services are managed by the source module **pico_channel_IP.cpp.** The server must be running either **PicoChannelServer** or **PicoDaemon.** The latter is a service (under Windows) which is a small program designed to start PicoChannelServer when it receives the appropriate packet from pico_channel_IP.

The purpose of PicoDaemon is to provide an 'every-present' target to which message can be sent to manage PicoChannelServer. There are other mechanisms under Windows to achieve this, but the pair of programs PicoDaemon and PicoChannelServer are compatible and interoperable with a Linux server and client.

All subsequent traffic uses the TCP/IP connection established directly with the channel server.

4.2.1 PicoDaemon

PicoDaemon.exe must be registered as a service under Windows and before it is started.

PicoDaemon /regserver	register PicoDaemon as a service
PicoDaemon /unregserver	remove PicoDaemon as a service
net start PicoDaemon	start PicoDaemon

net stop PicoDaemon stop PicoDaemon

After it has been registered you may also designate PicoDaemon to be started automatically when Windows is rebooted. Goto

Control panel /Administrative tools / Component Services & select Services (local)
and specify **automatic startup**.

PicoDaemon accepts a UDP packet from the class cPicoChannel_IP which signals it to start **PicoChannelServer**

PicoDaemon accepts the following command line arguments:

/debug	enables debug mode
/port <adr>	specify port address, default=5001.
/silent	do not display message boxes for regserver or unregserver
/regserver	registers PicoDaemon as a service.
/unregserver	unregisters PicoDaemon as a service.

Once the port address is specified it will be used on all subsequent invocations of PicoDaemon.

4.2.2 PicoChannelServer

PicoChannelServer does the heavy lifting. PicoChannelServer listens for a TCP/IP connection from a client. When is receives a TCP/IP packets it unravels the packets and accesses the local Pico Card. PicoChannelServer then packages up and returned data and sends it back to the client.

PicoChannelServer accepts the following command line arguments:

$	enable 'debug mode'. Each message received is logged to the display
/port	specify port address on which to post a listen; default = 5001.

When PicoChannelServer is running the $ key can be used to enable or disable debug mode.

PicoChannelServer can be invoked from a command line or from **PicoDaemon.** If the daemon is not running but the channel server is running the UDP packet is ignored.

4.2.3 Pico Remote Server Example

Example:

```
cPicoChannel *channelP;

//Get a summary of Pico Cards at the channel server 1.2.3.4
ii = AvailablePicoCards(avail, sizeof(available), "1.2.3.4");

channelP = new cPicoChannel(10, "ipAdr=1.2.3.4, readOnly,
file=$(picobase)\\bin\\mybitfile.bit");

//Check for errors
if ((erC=channel.GetError()) < 0) {print("Error %u creating picochannel\r\n", -
erC); return erC;}

//Perform a write
channel.Write(buf, sizeof(buf));

//Perform a write
ii = channel.Read(buf, sizeof(buf));
```

The text specifying the channel server ("ipAdr=1.2.3.4" in this case) will be removed when it is passed to the class constructor at the channel server. At the channel server a channel is created on behalf of the remote user, ie,

```
      channelP = new cPicoChannel(10, "readOnly,
file=$(picobase)\\bin\\mybitfile.bit");
```

NOTE: Special care should be exercised with the **file=fileName** parameter to the class constructor. The file will be resolved at the channel server**.**

If you wish to load from your local machine you must specify a file name that will resolve to your local machine - such as

\\myMachine\c:\%picobase%\bin\mybitfile.bit

NOTE: The same caveat applies to $(picobase) or any other environment variables. They will be resolved in the context of the channel server.

4.2.4 User Defined Server Functions

The user may supply a library which provides additional function to the program **picoChannelServer.** Under Windows such a file usually has the suffix **.dll**. The library file allows the user to provide functionality at the server rather than use multiple channel functions which must be passed individually across the TCP/IP line. For example the user supplied library file could perform a read/modify/write function which cannot be interrupted by a second user accessing the same card.
The class cPicoChannel_IP provide the marshalling of data to and from the server.

Server Side

The library file resides on the PicoChannelServer side of the remote connection. The library file must declare three export functions:

```
extern "C" __declspec(dllexport) int Initialize(uint32_t version);

extern "C" __declspec(dllexport) int Shutdown(void);

extern "C" __declspec(dllexport) int Process(uint32_t cmd, cPicoChannel *channelP,
                                             void *bP, int ilen, void *obP,
int *olenP);
```

The functions **Initialize** is called when the library file is loaded. The user should verify that the version is acceptable and return zero if it is otherwise a negative error code.

The function **Shutdown** is called before the library file is removed from memory.

The function **Process:**

- processes an input buffer and stores the results in the output buffer.
- updates the value at ***olenP**.
- returns a negative error code if there is an error.
- return a positive value to indicate various shades of success.

The parameter **cmd** must be in the range:

```
[PICO_TU_USER_COMMAND_LO, PICO_TU_USER_COMMAND_HI]
```

Refer to **sample\userChannelServer** for an example of this function.

User Side

When Pico Channel is instantiated on the client side the parameter to the constructor should include the phrase:

userlib=filename

for example:

```
     channelP = new cPicoChannel(10, "ipAdr=1.2.3.4,
userLib=c:\\Pico\\bin\\myUserLibl.dll");
```

The filename in this case is resolved at the channel server and therefore refers to a file on the server.

From the client side the user defined function is called as follows:

```
       inBuf[0] = 0x5a5a5a5a;
       erCode   = channel.UserCommand(cmd, &channel, inbuf,sizeof(inbuf),
outbuf,&(size=sizeof(outbuf)));
       if (erCode < 0) {printf("error %u\r\n", -erCode);}
       else            {printf("sizeof results=%u: 0x%08X\r\n", size, outBuf[0]);
```

Bigendian Note: The header used to transport the user buf to the channel server is automatically aligned to hi-lo byte order. However the use data buffers (inbuf and outbuf) will be in the format of the originating system - ie lo-hi byte order for Pentium based systems.

4.3 DeviceIOControl

DeviceIoControl is an out of band file control mechanism that can be used to perform functions not included in cPicoChannel.

The function GetHandle() is used to obtain the operating system handle which is required by DeviceIoControl.

```
DeviceIoControl(channel.GetHandle(), PICO_SET_FLAGS,
                              &inputBuf,sizeof(inpuBuf),       //input parameters
                              &outputBuf,sizeof(outputBuf),    //output results
(optional)
                              &oCount, NULL);
```

For example, the following call to DeviceIoControl will set and return the debugging flags:

```
u32 = 0x9F66000;
DeviceIoControl(chhannel.GetHandle(), PICO_SET_FLAGS, &u32,sizeof(u32),
&u32,sizeof(u32), &oCount, NULL);
```

(Of course, this could also be done using SetGetOptions function)

DeviceIoControl is not consistent across operating systems. However, cPicoChannel has a protected cross platform implementation IoControl. You can use this function by deriving your own class as follows:

```
class cMyPicoChannel : public cPicoChannel
   {public:
    int IoControl(uint32_t fcn, void *ibP, uint32_t ilen, void *obP, uint32_t
olen, pico_size_t nu)
        {return m_channelP->IoControl(fcn, ibP, ilen, obP, olen, nu);}
   };
```

4.4 Defines Associated with Channels

The following defines are associated with DMA access and the associated status registers.

#define BM_CHANNEL_BASE	0x10100000U
#define BM_CHANNEL_SIZE	0x100000U
#define BM_MAX_CHANNELS	1791U

#define BM_STATUS_BASE	0x10000010U
#define BM_STATUS_SIZE	0x10U
BM_CHANNEL_BASE:	Bus Mastering devices use the address space above 0x10100000. Channel 1 is 0x10100000.
BM_CHANNEL_SIZE:	Each Bus Mastering channel is 0x10000 (1,048,576) bytes wide.
BM_CHANNELS_MAX:	There are a maximum of 1791 channels == (0x7FFFFFFF - 0x10100000) / 0x100000.
BM_CHANNEL_STATUS_BASE:	Status registers use [0x10000000 - 0x100FFFFF]. Channel 1 is 0x10000010.
BM_CHANNEL_STATUS_SIZE:	Each bank of status registers is 16 bytes wide (4 * 32bit registers).

#define BM_CHANNEL_FROM_ADDR(addr)	(1+(addr - BM_CHANNEL_BASE) / BM_CHANNEL_SIZE)
#define BM_CHANNEL_FROM_STATUS(addr)	(1+(addr - BM_STATUS_BASE) / BM_STATUS_SIZE)
#define BM_ADDR_FROM_CHANNEL(channel)	(BM_CHANNEL_BASE + (channel-1) * BM_CHANNEL_SIZE)
#define BM_STATUS_FROM_CHANNEL(channel)	(BM_STATUS_BASE + (channel-1) * BM_STATUS_SIZE)

These macros map channels to addresses and addresses to channels.

```
typedef struct
  {UINT32 wordsAvailable:20, //bits 0x000FFFFF
          nu             :6,  //bits 0x03F00000 == 0
          signature      :6;  //bits 0xFC000000 == PCMCIA_BM_READ_STATUS_SIGNATURE
(0x26) or
                              //               == PCMCIA_BM_WRITE_STATUS_SIGNATURE
(0x22)
  } BM_DEVICE_STATUS;

 Status register returned from firmware device to control transfer between host and
Pico Card.

typedef struct
   {uint32_t  timeout1;  //timeout for first call
    uint32_t  timeout2;  //timeout for second and subsequent calls
   } PICO_TIMEOUT;
```

These structures are used to set timeout.

4.5 Creating a Sample Program

The following code snippet illustrates the basic techniques of accessing a Pico Card(s). The snippets are taken from the Picobus_counter example (%picobase%\samples\picobus_counter\software)

```
1. #define CHANNEL_NO 1

2. int main(int argc, char* argv[])
3. {int               erC, ii;
4.    uint32_t          buf[1000];
5.    const char       *paramsP="!loaded, file=$(model)_Picobus_counter_ISE.bit";
6.    cPicoChannel      channel(CHANNEL_NO, paramsP);
7.    //Check that channel was created correctly
8.    if ((erC=channel.GetError((char*)buf, sizeof(buf))) < 0)
9.       {printf(stderr, "Error %u creating channel\nContext: %s\n", -erC, (char
*)buf);
10.        exit(1);
11.      }
```

paramsP contains parameters in the general format keyword=value,....
buf will receive any error message generated by cPicoChannel.

The integer **erC** will receive the error number: non-negative if successful, negative if an error.

Line 6 creates the cPicoChannel class, with channel # = 1
Lines 8-11 check that the channel was created correctly, and report an error message if it did not

5. Firmware

The Picobus firmware in all Pico bit images provides functionality that handles DMA transfers. The same interface also allows your application program to make single word accesses (i.e. non DMA accesses) to the Pico card without additional logic. The PicoBus interfaces to the user logic peripheral with the following signals:

picoAddr[31:0]	address of operation	bits [19:0] all map to this user logic device and [31:20] select which device
picoDataIn[31:0]	data from the PC	
PicoDataOut[31:0]	data to the PC	
PicoRd	asserted when data is going to the PC	
PicoWr	asserted when data is coming from the PC	
PicoClk	clock for above signals	

The interface is exactly the same whether the device is accessed (from software) using a single word access protocol (Read Device or Write Device see the class cPicoChannel) or DMA access (Read or Write The class cPicoChannel).

5.1 Implementing a Channel in Firmware

All of the following Verilog code snippets could be found in a typical Verilog file defining a user logic device. The Verilog declaration of a device should be:

```
`include "PicoDefines.v"

module BMmodule
  (input             PicoRst,
   output [31:0]     PicoDataOut,       //data returned
   input  [31:0]     PicoAddr,          //address from PCIe/USB bus
   input  [31:0]     PicoDataIn,        //data    from PCIe/USB
   input             PicoRd,            //IO Read from PCIe/USB bus
   input             PicoWr,            //IO Write to  BM/PCMCI bus
   input             PicoClk            //clock from   PCIe/USB
  );
```

For data being transferred from the PC host to the Pico Card:
picoAddr and PicoDataIn will be valid on the rising edge of PicoClk when PicoRd is asserted.

For data beings transferred from the Pico Card to the PC host
picoAddr is valid on the rising edge of PicoClk when PicoWr is asserted.
PicoDataOut should be valid before the next rising edge of PicoClk.

PicoAddr is specified when the application program accesses the Pico card. PicoAddr may be any address in the range [0x1000,0000 0x7FFF,FFFF]. User logic devices that adhere to the channel model, and have the default size of 0x100000 bytes, only respond when the top 12 bits of PicoAddr are the same as their channel #, plus 0x10000000. This allows any channelNumber between 1 and 1791. The bottom 20 bits may be used for devices within the user logic device, or ignored, at the user's discretion. The following logic develops the signals myRead and myWrite, which would be used as more specific versions of PicoRd and PicoWr:

```
`define MY_DEVICE_ADR 32h'10200000 //This user logic device is on Channel 2

wire   meSelected, myRead, myWrite;
```

```
assign meSelected = {PicoAddr[31:20], 20'b0} == `MY_DEVICE_ADR;
assign myRead     = meSelected & PicoRd;              //PCIe/USB  read  data
assign myWrite    = meSelected & PicoWr;              //PCIe/USB write data
```

The logic to read or write data might be something like:

```
reg [31:0] deviceData;
always @(posedge PicoClk)
    begin
    if (myRead) deviceData   <= deviceData + 1;   else      //perform action required
when data is read
    if (myWrite) deviceData  <= deviceData + PicoDataIn; //perform actions required
when data is written
    end

assign PicoDataOut = (myRead ? deviceData : 0);   //OR the device data onto the Pico
output Bus
```

The pacing register requires the following addition logic:

```
`define BM_READ_STATUS_ADR  32'h01000200  //read pacing register for channel 2
`define BM_WRITE_STATUS_ADR 32'h01000204  //write pacing register for channel 2
wire   myStatRead, myStatSelectR, myStatWrite, myStatSelectW;
assign myStatSelectR  = ({picoAddr[31:2], 2'b0} == `BM_READ_STATUS_ADR);
assign myStatSelectW  = ({picoAddr[31:2], 2'b0} == `BM_WRITE_STATUS_ADR);
assign myStatRead     = (myStatSelectR & PicoRd);  //device read status
assign myStatWrite    = (myStatSelectW & PicoRd);  //device write status
reg    [11:0]readWordsAvailable;  //number of 32bit words available to read from
device
reg    [11:0]writeWordsAvailable; //number of 32bit words that may be written to the
device
```

The output to the Pico bus would then be:

```
assign PicoDataOut = (myRead      ? deviceData
: 0) |
                     myStatRead  ? {`PCMCIA_BM_STATUS_SIGNATURE, 6'h0,
readWordsAvailable}  : 0) |
                     myStatWrite ? {`PCMCIA_BM_STATUS_SIGNATURE, 6'h0,
writeWordsAvailable} : 0);
```