

Pico Computing Inc.



Pico Driver (Pico.sys) 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

1 Overview

This manual describes the lowest level software component used to access Pico Cards. The driver provides the hardware level interface all other software such as [Pico.dll](#), and [PicoUtil.exe](#). Under the Windows Driver Model (WDM) the Pico Driver is called Pico.sys. Under Linux 2.6 the Pico Driver is called Pico.ko. The Pico driver is intimately concerned with the operation of the firmware and the [hardware](#) of the Pico 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 .

1.1 Version Information

Versions are numbered with four decimal digits:

major.minor.release.counter

for example: **3.10.0.6**

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: V3.10.0.06. Jul 21 2005 08:32:59.  
Pico.dll: V3.10.0.06. Jul 21 2005 08:32:59.  
Pico.sys: V3.10.0.06 Debug: Jul 20 2005 16:36:41.  
Firmware: (FPGA) V3.10.0.6.  
PPC: Pico Monitor V3.10.0.6.
```

The fields of this message have the following meanings:

- **PicoUtil.exe:** V3.10.0.06. Jul 21 2006 08:32:59
This is the version and compilation date of PicoUtil.exe.
- **Pico.dll:** V3.10.0.06. Jul 21 2006 08:32:59.
This is the version number of Pico.dll.
- **Pico2K.sys:** V3.10.0.06 Debug: Jul 20 2006 16:36:41.
This is the version and compile date of the driver (Pico.sys).
- **Firmware:** (FPGA) V3.10.7.0.
This is the version of the current FPGA bit image
- **PPC:** Pico Monitor V3.10.1.0.
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:

3.10.0.6

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

3.10.0.6

- Versions with the same major version numbers but different minor version numbers (eg. 3.9 to 3.10) should be compatible at the source code level. They may require the code to be recompiled.

3.10.0.6

- 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 3.10.0.x PicoUtil.exe should run with a version 3.10.1.x version of Pico.dll. Naturally, improvements in a later version number will not be present in the earlier code.

3.10.0.6

- 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.

2 Architecture

Overview

The Pico Driver provides the hardware level interface for higher level software such as [Pico.dll](#), and [PicoUtil.exe](#). Under the Windows Driver Model (WDM) the Pico Driver is called Pico.sys. Under Linux 2.6 the Pico Driver is called Pico.ko. The Pico driver is intimately concerned with the operation of the firmware and the [hardware](#) of the Pico Cards. This manual describes the operation of Pico Driver.

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

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

3 Device IO calls

Device IO calls

The Pico Card is a plug-and-play device. When the Pico card is inserted into the PCMCIA slot the driver is loaded by the operating system. From the perspective of an application program the Pico

Card is treated as a generic device.

A handle to the device must be obtained by opening the Pico.sys driver as follows:

```
drvHnd = CreateFile(
    "\\.\Pico1",
    GENERIC_READ | GENERIC_WRITE,    //access mode
    0,                                //shared mode: 0 =
    exclusive                          //SHARED_READ |
    NULL,                              //SHARED_WRITE will also work
    OPEN_EXISTING,
    0, NULL
);
```

Second and subsequent Pico Cards can be opened with the name `Pico2`, `Pico3`,... `Pico32`.

In Linux this is

```
drvHnd = open("/dev/pico");
```

Subsequent accesses to the device use `DeviceIoControl` under Windows or `IoControl` under Linux 2.6. This is a generic 'out of band' interface:

```
DeviceIoControl(
    drvHnd,                                //handle to driver from open
    IO_CALL,                              //defined in the following
    sections
    inBuf, sizeof(inBuf),                 //input buffer and size
    outBuf, sizeof(outBuf),              //output buffer and size
    &oCount,                              //number of bytes written
    NULL                                  //overlapped I/O (not used)
);
```

Many calls use only the input or only the output buffer and the pointers can be replaced with `NULL, 0`.

3.1 Version, Configuration, and Statistics

PICO_VERSION

This call to `DeviceIoControl` returns the version information in ascii. The following is typical:

```
DeviceIoControl(
    drvHnd,                                //handle to driver from open
    PICO_VERSION,                          //this operation
    NULL, 0,                               //input buffer and size
    outBuf, sizeof(outBuf),               //output buffer and size
    &oCount,                              //number of bytes written
    NULL                                  //overlapped I/O (not used)
);
```

After this call `outbuf` might contain `"V3.4.0.02 Debug: Oct 25 2005 18:16:48."`

PICO_GET_CONFIG

This call to DeviceIoControl returns version and capabilities of the firmware. The following is typical

```
PICO_CONFIG cfg;
DeviceIoControl(
    drvHnd,                //handle to driver from open
    PICO_GET_CONFIG,      //this operation
    NULL,0,               //input buffer and size
    cfg,sizeof(cfg),      //output buffer and size
    &oCount,               //number of bytes written
    NULL                   //overlapped I/O (not used)
);
```

After this call cfg will be filled in with the proper values. The structure of PICO_CONFIG is

```
struct PICO_CONFIG
{
    pico_version_t  firmwareVer;    //+0 firmware version
    uint32_t        capabilities;   //+4 capabilities, OR-ed together
    (see following table).
    uint16_t        model;          //+8 == 0xE12 or 0xE14 or E16
    uint16_t        lxCard,         //12 == 'LX' or 'FX'
    uint32_t        magicNum;       //14 == 0x5397
    uint32_t        portAdr,        //16 == port address assigned to
Pico Card
    address         memAdr,         //20 == virtual memory base
    address         openCount,      //24 == number of programs
    attached to driver
    shareAccess;    //28 == open mode of programs
    using pico.sys
    uint64_t        physicalAdr;    //32 == physical address of
memory space
    uint32_t        interrupt;      //38 == interrupt number assigned
to Pico Card
    int             lastError;      //40 == last error reported by
Pico.sys
    uint32_t        startupError;    //44 == NTSTATUS code from
startup of pico.sys
    char            description[128]; //48+ = description of driver
};
```

Capabilities Table

Name	Bit	Meaning
PICO_CAP_FLASH	0x0001	Facilities to Access Flash from PC is implemented in firmware. This allows the host PC to read flash ROM directly and to fabricate the appropriate control sequences to write flash ROM.
PICO_CAP_TURBO LOADER	0x0002	Turbo loader control & status ports are implemented in firmware. This allows the host PC to interrogate the last address loaded into the flash ROM (peekaboo register).
PICO_CAP_KEYHO LE	0x0004	PPC keyhole debugging register is implemented in firmware. This allows the application program to transfer data between the PC host program and the Pico Card program.
PICO_CAP_FPGAL OADED	0x0008	FPGA Restart port is implemented in firmware. This allows the PC host to reset the peekaboo register and to provoke a reload of firmware from the flash ROM (reboot).
PICO_CAP_BUSMA STERING	0x0010	Image has bus mastering
PICO_CAP_ADC	0x0020	Image has A/D and D/A
PICO_CAP_PIC	0x0040	Image has programmable Interrupt Controller
PICO_CAP_JTAG_S	0x0080	Image has JTAG spy (it always has JTAG support)

```

PY
PICO_CAP_ETHER 0x0100 Image has Ethernet
NET
PICO_CAP_UARTLI 0x0200 Image has UART lite
TE
PICO_CAP_GPIO 0x0400 Image has GPIO capabilities
PICO_CAP_MPORT 0x0800 Image support multiport RAM
_RAM
PICO_CAP_HOBBL 0x2000 E-12 is using configuration memory to set AER (hobbled mode)
ED
PICO_CAP_KEYHO 0x4000 raw keyhole hardware ignoring presence of monitor.elf
LE_RAW
PICO_CAP_MONIT 0x8000 monitor is running (always zero from driver. Pico.dll makes this
OR determination)

```

PICO_GET_STATISTICS

This call to DeviceIoControl returns various statistics about the Pico Card. After the call most of the statistics are set to zero. The following is typical:

```

uint32_t stats[32];
DeviceIoControl(
    drvHnd, //handle to driver from open
    PICO_VERSION, //this operation
    NULL, 0, //input buffer and size
    stats, sizeof(stats), //output buffer and size
    &oCount, //number of bytes written
    NULL //overlapped I/O (not used)
);

```

After the call stats[] will have the following values:

Stats[]	Value	Cleared
stats[0]	dma address	N
stats[1]	last pico address	N
stats[2]	last dma command	N
stats[3]	count of DMA requests executed manually (ie using straight read/write memory)	Y
stats[4]	number of bytes transferred manually	Y
stats[5]	number of DMA transactions	Y
stats[6]	number of bytes transferred over DMA	Y
stats[7]	number of times DMA did not start (jammed)	Y
stats[8]	number of calls to Service Interrupt	Y
stats[9]	number of times interrupts could not be serviced because another task was accessing que structures	Y
stats[10]	value of debug flags	N
stats[11]	number of times an I/O transaction was started under the timer interrupt	Y
stats[12]	physical address of memory block assigned to Pico.sys	N
stats[13]	size of memory block assigned to Pico.sys	N
stats[14]	physical address of memory block mapped to PCI / PLX registers	N
stats[15]	size of memory block mapped to PCI / PLX registers	N
stats[16]	nu	N
stats[17]	nu	N
stats[18]	last error (refer to Pico_errors.xml)	Y
stats[19]	error that occurred at startup of Pico.sys	N

stats[20] physical address of DMA register

N

3.2 Debugging Flags

PICO_SET_FLAGS

This call to DeviceIoControl sets global debugging flags. Some of these pertain to Pico.sys. The following call will get/put the flags:

```
uint32_t newFlags, oldFlags;
newFlags = BUG_DRIVER_SUMMARY | BUG_DRIVER_DETAIL;
DeviceIoControl(
    drvHnd,                                //handle to driver from open
    PICO_SET_FLAGS,                        //this operation
    &newFlags, sizeof(newFlags),           //input buffer and size
    &oldFlags, sizeof(oldFlags),          //output buffer and size
    &oCount,                                //number of bytes written
    NULL                                    //overlapped I/O (not used)
);
```

This call will set the flags from newFlags and return the previous settings in oldFlags. The flags are bit bit encoded as follows:

Flags Table.

Name	Bits	Meaning
BUG_ELF_LOAD	0x00000001	Display segments loaded in an ELF file (implemented in monitor.elf)
BUG_ELF_PRO	0x00000002	Display segments loaded in an ELF file (implemented in monitor.elf)
GRAM		
BUG_ELF_RFU	0x0000FFF0	Reserved for user. These bits are not used, but may be commandeered by the user. They can be set in an application program.
BUG_DRIVER_SUMMARY	0x00200000	Debugging summary mode
BUG_DRIVER_INTERRUPT	0x00400000	Debug INterrupt events in Pico.sys
BUG_DRIVER_POWER	0x00800000	Debug power event in Pico.sys
BUG_DRIVER_DETAIL	0x01000000	Debugging detail mode
BUG_DRIVER_JTAG	0x02000000	Debug Jtag events in Pico.sys
BUG_DRIVER_KEYHOLE	0x04000000	Debug Keyhole events in Pico.sys
BUG_DRIVER_BUS_MASTERING	0x08000000	Debug Bus Mastering events in Pico.sys

PICO_GET_FLAGS

This call to DeviceIoControl gets global debugging flags. Some of these pertain to Pico.sys. The following call will get the flags:

```

uint32_t flags;
DeviceIoControl(
    drvHnd,                //handle to driver from open
    PICO_GET_FLAGS,       //this operation
    NULL,0,                //input buffer and size
    &flags,sizeof(fags),   //output buffer and size
    &oCount,                //number of bytes written
    NULL                   //overlapped I/O (not used)
);

```

Debug switches will not generate output in the release version of Pico.sys. You must have the debug version `picoD.sys` loaded in place of `pico.sys`.

To capture the debug activity:

- Copy `c:\pico\bin\picoD.sys` over `c:\windows\system32\drivers\Pico.sys`
- Run a debugging tool such as `DbgView.exe` from www.sysinternals.com to view the output
- Eject/insert the Pico Card to reload the driver (ie `picoD.sys` masquerading as `Pico.sys`).

You may set the debugging switches from the `Picoutil.exe`, `picocommand.exe`, your own program or using the command line utility `PicoBug.exe`.

`PicoBug -h` will display the following screen:

```

----- Picobug: Program to set or reset debug flags
-----

PicoBug # - specifies card number
PicoBug B - set Bus Mastering debugging, ~B - reset Bus Mastering
debugging
PicoBug D - set detail debugging, ~D - reset detail debugging
PicoBug I - set debug interrupt events, ~I - reset debug interrupt
events
PicoBug J - set JTAG debugging, ~J - reset JTAG debugging
PicoBug K - set keyhole debugging, ~K - reset keyhole
debugging
PicoBug P - set debug power events, ~P - reset debug power
events
PicoBug S - set summary debugging, ~S - reset summary
debugging
PicoBug E - set debug program load, ~E - reset debug program
load
PicoBug U - set ELF user switch, ~U - reset ELF user switch
PicoBug [+ | -] hexValue will set / reset a specific bit in the debug
flags.
PicoBug ~ or PicoBug - will reset all debugging flags.

```

```

Example: PicoBug 3 sd      Sets summary and debug flags for Pico Card
3.
          PicoBug ~bids   Resets Bus Mastering, Interrupt, detail,
and
                                summary flags on first Pico Card.
          PicoBug +0x10   Sets one of the 12 user flags for a PPC
program.

```

NOTE: Debug flags apply across the entire Pico software stack.

: Switches `b d i j k p`, and `s` require the debug version of `Pico.sys`.

: Switches `e` and `u` pertain to programs running on the PPC in the Pico Card.

For example:

picobug bids will enable the bus mastering, and interrupt debug switches and display both summary and detail messages.

picobug ~bids will disable the same switches

3.3 Read Operations

PICO_READ_SNIPPETS

This call to DeviceIoControl reads a few bytes from the beginning of each sector. It is used by BuildDirectory (in Pico.DLL) to quickly scan the flash ROM and locate sectors which contain file headers. The following is typical:

```
DeviceIoControl(
    drvHnd,                                //handle to driver from open
    PICO_READ_SNIPPETS,                    //this operation
    inBuf32, sizeof(inBuf32),              //input buffer and size
    outBuf, sizeof(outBuf),                //output buffer and size
    &oCount,                                //number of bytes written
    NULL                                    //overlapped I/O (not used)
)
inBuf32[0] = beginning address,
inBuf32[1] = zero,
inBuf32[2] = logical sector size,
inBuf32[3] = number of bytes to read from each sector.
```

After this call outBuf will contain as many snippets (of length inBuf32[3]) for each sector as can be fit in the size specified for the outBuf.

PICO_READ_DEVICE

This call to DeviceIoControl reads a from the memory mapped area of the Pico Card. This is the area in which you would implement devices in firmware. The following is typical:

```
DeviceIoControl(
    drvHnd,                                //handle to driver from open
    PICO_READ_DEVICE,                      //this operation
    inBuf32, sizeof(inBuf32),              //input buffer and size
    outBuf, sizeof(outBuf),                //output buffer and size
    &oCount,                                //number of bytes written
    NULL                                    //overlapped I/O (not used)
)
```

The input buffer should contain the 64bit address at which the read should start. The sizeof the output parameters defines how many 32bit words are read.

PICO_READ_FLASH

This call to DeviceIoControl reads a from the flash memory of the Pico Card (if present). The following is typical:

```
DeviceIoControl(
    drvHnd,                                //handle to driver from open
    PICO_READ_FLASH,                      //this operation
    inBuf32, sizeof(inBuf32),              //input buffer and size
    outBuf, sizeof(outBuf),                //output buffer and size
    &oCount,                                //number of bytes written
```

```

        NULL //overlapped I/O (not used)
    )

```

The input buffer should contain the 64bit address at which the read should start. The sizeof the output parameters defines howmany 32bit words are read.

PICO_READ_CIS

This call to DeviceIoControl reads a from the CIS memory of the Pico Card. The word CIS in this context should be liberally interpreted. On the E-12 it is the Card Information Structure. On the E14-15 this call reads the cardbus function registers. On the E-16 this call reads the PCI Express function registers. The following is typical:

```

DeviceIoControl(
    drvHnd, //handle to driver from open
    PICO_READ_CIS, //this operation
    inBuf32, sizeof(inBuf32), //input buffer and size
    outBuf, sizeof(outBuf), //output buffer and size
    &oCount, //number of bytes written
    NULL //overlapped I/O (not used)
)

```

The input buffer should contain the 64bit address at which the read should start. The sizeof the output parameters defines howmany 32bit words are read.

PICO_READ_LCS

This call to DeviceIoControl reads a from the Pico E-16 LCS registers. The following is typical:

```

DeviceIoControl(
    drvHnd, //handle to driver from open
    PICO_READ_LCS, //this operation
    inBuf32, sizeof(inBuf32), //input buffer and size
    outBuf, sizeof(outBuf), //output buffer and size
    &oCount, //number of bytes written
    NULL //overlapped I/O (not used)
)

```

The input buffer should contain the 64bit address at which the read should start. The sizeof the output parameters defines howmany 32bit words are read.

PICO_READ_PLX_8111

This call to DeviceIoControl reads a from the Pico E-16 PLX-8111 registers. The following is typical:

```

DeviceIoControl(
    drvHnd, //handle to driver from open
    PICO_READ_PLX_8111, //this operation
    inBuf32, sizeof(inBuf32), //input buffer and size
    outBuf, sizeof(outBuf), //output buffer and size
    &oCount, //number of bytes written
    NULL //overlapped I/O (not used)
)

```

The input buffer should contain the 64bit address at which the read should start. The sizeof the output parameters defines howmany 32bit words are read.

PICO_READ_PORT

The standard Pico firmware (usually loaded from PrimaryBoot.bit) provides several ports to control features implemented in that firmware. This call to DeviceIoControl will read the specified I/O port(s). The following is typical

```

uint32 outBuf[2];
outBuf[0] = port_address;
outBuf[1] = port_count;
DeviceIoControl(
    drvHnd,                                //handle to driver from open
    PICO_READ_PORT,                        //this operation
    NULL,0,                                //input buffer and size
    outBuf,sizeof(outBuf),                //output buffer and size
    &oCount,                               //number of bytes written
    NULL                                   //overlapped I/O (not used)
)

```

After this call outBuf[0,1,2,...] will contain the values of the specified ports.

3.4 Write Operations

PICO_WRITE_DEVICE

This call to DeviceIoControl writes to the memory mapped area of the Pico Card. This is the area in which you would implement devices in firmware. The following is typical:

```

DeviceIoControl(
    drvHnd,                                //handle to driver from open
    PICO_READ_DEVICE,                      //this operation
    inBuf32, sizeof(inBuf32),              //input buffer and size
    outBuf,sizeof(outBuf),                //output buffer and size
    &oCount,                               //number of bytes written
    NULL                                   //overlapped I/O (not used)
)

```

The input buffer should contain the 64bit address at which the read should start. The sizeof the output parameters defines how many 32bit words are read.

PICO_WRITE_CIS

This call to DeviceIoControl writes to the CIS memory of the Pico Card. The word CIS in this context should be liberally interpreted. On the E-12 it is the Card Information Structure. On the E14-15 this call reads the cardbus function registers. On the E-16 this call reads the PCI Express function registers. The following is typical:

```

DeviceIoControl(
    drvHnd,                                //handle to driver from open
    PICO_READ_CIS,                          //this operation
    inBuf32, sizeof(inBuf32),              //input buffer and size
    outBuf,sizeof(outBuf),                //output buffer and size
    &oCount,                               //number of bytes written
    NULL                                   //overlapped I/O (not used)
)

```

The input buffer should contain the 64bit address at which the read should start. The sizeof the output parameters defines how many 32bit words are read.

PICO_WRITE_LCS

This call to DeviceIoControl writes to the LCS registers on the Pico E-16 card (only). The following is typical:

```

DeviceIoControl(
    drvHnd,                                //handle to driver from open

```

```

        PICO_READ_LCS,                //this operation
        inBuf32, sizeof(inBuf32),    //input buffer and size
        outBuf, sizeof(outBuf),      //output buffer and size
        &oCount,                      //number of bytes written
        NULL                          //overlapped I/O (not used)
    )

```

The input buffer should contain the 64bit address at which the read should start. The sizeof the output parameters defines howmany 32bit words are read.

PICO_WRITE_PLX_8111

This call to DeviceIoControl writes to the LCS registers on the Pico E-16 card (only). The following is typical:

```

    DeviceIoControl(
        drvHnd,                        //handle to driver from open
        PICO_READ_PLX_8111,            //this operation
        inBuf32, sizeof(inBuf32),    //input buffer and size
        outBuf, sizeof(outBuf),      //output buffer and size
        &oCount,                      //number of bytes written
        NULL                          //overlapped I/O (not used)
    )

```

The input buffer should contain the 64bit address at which the read should start. The sizeof the output parameters defines howmany 32bit words are read.

PICO_WRITE_FLASH_CMD

This call to DeviceIoControl writes a series of commands to the flash ROM. The data structure passed in the input buffer has the following structure:

```

typedef struct
{
    uint32_t addr;                    //address
    uint16_t data;                   //data
    nu;                               //to ensure uniform packing
}
FLASH_CMD;

FLASH_CMD
EnterCFIprobeMode[] = {{0x55, 0x98}},
ExitCFIprobeMode[] = {{0x00, 0xF0}};

```

The following is typical:

```

    DeviceIoControl(
        drvHnd,                        //handle to driver from open
        PICO_WRITE_FLASH_CMD,         //this operation
        EnterCFIprobeMode,
        sizeof(EnterCFIprobeMode),    //input buffer and size
        NULL, 0,                      //output buffer and size
        &oCount,                      //number of bytes written
        NULL                          //overlapped I/O (not used)
    );

```

This will put the flash ROM in CFI probe mode. With different FLASH_CMD data the flash ROM a variety of tasks can be performed including erasing and writing the flash ROM. Refer to the flash ROM manufacturers documentation for information on CFI mode.

PICO_WRITE_PORT

This call to DeviceIoControl will write to the specified I.O ports. The following is typical:

```
uint32 outBuf[2];
outBuf[0] = port_address;
outBuf[1] = port_count;
DeviceIoControl(
    drvHnd,
    PICO_WRITE_PORT,
    NULL,0,
    outBuf,sizeof(outBuf),
    &oCount,
    NULL
)
//handle to driver from open
//this operation
//input buffer and size
//output buffer and size
//number of bytes written
//overlapped I/O (not used)
```

3.5 Keyhole Access

Keyhole Access

The keyhole port is a set of firmware and software which provides a full duplex communication channel between the PCMCIA bus (host PC) and the PPC (Pico Card). The channel is thirty six bits wide which provides for a four bit 'command field' and a 32 bit data field. The command field dictates what the sender expects the receiver to do with the 32bit data field. Before transfer of data from the keyhole a call to DeviceIoControl with operation code=PICO_KEYHOLE_ENABLE should be made:

PICO_KEYHOLE_ENABLE

This call to DeviceIoControl enables or disables polling of the keyhole port by the driver. The following is typical:

```
bool enable;
enable = true;
DeviceIoControl(
    drvHnd,
    PICO_WRITE_REG,
    &enable,sizeof(enable),
    NULL,0,
    &oCount,
    NULL
)
//enable to keyhole receive logic
//handle to driver from open
//this operation
//input buffer and size
//output buffer and size
//number of bytes written
//overlapped I/O (not used)
```

Thereafter transfers to and from these FIFO's are managed using the following structures:

```
typedef struct
{
    union {uint32_t u32;
        struct {uint32_t asci3:8, //0.000000FF bits
                asci2:8, //0.0000FF00 bits
                asci1:8, //0.00FF0000 bits
                asci0:8;}; //0.FF000000 bits
        struct {uint32_t byteCount:10, //0.000003FF bits number
                packetType:6, //0.0000FC00 bits
                portNo:16; //0.FFFF0000 bits port.
        };
    };
};
```

```

    } KEYHOLE_DATA;

typedef struct
{
    uint16_t status :6,
             nu      :2,
             signature:8;
} KEYHOLE_STATUS;

//Structure of data returned from driver
typedef struct
{
    KEYHOLE_DATA d;
    UINT8        cmnd; //command associated with data
} KEYHOLE_BUF;

```

The port number is a value between 0 and 65535. Port 0 will read or write the next message regardless of the port specification used in the message. Port 1 is used by the Kprintf and Kerror functions, the remaining ports are available to the user for any purpose whatsoever.

PICO_PPC_TO_PC_KEYHOLE

This call to DeviceIoControl reads the next message written by the PPC on the Pico Card on the specified port. The following is typical:

```

KEYHOLE_BUF keyholeBuf[258];
keyholeBuf[0].portNo = desired port;
DeviceIoControl(
    drvHnd, //handle to driver from open
    PICO_PPC_TO_PC_KEYHOLE, //this operation
    NULL, 0, //input buffer and size
    keyholeBuf, sizeof(keyholeBuf), //output buffer and size
    &oCount, //number of bytes written
    NULL //overlapped I/O (not used)
)

```

After this call the keyholeBuf might contain the message from the PPC:

```

keyholeBuf[0].d.portNo      = desired port;
keyholeBuf[0].d.packetType  = //see packetType table
keyholeBuf[0].d.cmnd       = KH_BEGIN; // == 0
keyholeBuf[0].d.byteCount  = n; //number of bytes being transferred in
keyholeBuf[1,2,3,...]
//messages payload is carried in entries 1,2,3,... n/4
keyholeBuf[1].d.cmnd       = KH_DATA; // == 1
keyholeBuf[1].d.u32       = desired value;
keyholeBuf[2,3,4,...n/4].d.u32 = .....
//message is terminated with KH_END
keyholeBuf[n+1].cmnd      = KH_END; // == 15
keyholeBuf[n+1].portNo   = 0; //not used
keyholeBuf[n+1].portNo   = 0; //not used
keyholeBuf[n+1].portNo   = 0; //not used

```

The keyhole could also be used to receive an eight bit message:

```

keyholeBuf[0].d.portNo      = desired port;
keyholeBuf[0].d.packetType  = //see packetType table
keyholeBuf[0].d.cmnd       = KH_END; // == 15
keyholeBuf[0].d.byteCount  = value; //

```

In other words, a plain KH_END is a single byte transfer; A message beginning with KH_BEGIN, containing zero or more 32 bit values, and ending with KH_END is a block transfer.

PICO_PC_TO_PPC_KEYHOLE

This call to DeviceIoControl writes a message from the PC to the specified port. The PPC can subsequently receive this message. The following is typical:

```

KEYHOLE_BUF keyholeBuf[258];
keyholeBuf[0].d.portNo      = desired port;
keyholeBuf[0].d.packetType  = //see packetType table
keyholeBuf[0].d.cmnd        = KH_BEGIN; // == 0
keyholeBuf[0].d.byteCount   = n; //number of bytes being transferred in
keyholeBuf[1,2,3,...]
//messages payload is carried in entries 1,2,3,... n/4
keyholeBuf[1].d.cmnd        = KH_DATA; // == 1
keyholeBuf[1].d.u32         = desired value;
keyholeBuf[2,3,4,...n/4].d.u32 = .....
//message is terminated with KH_END
keyholeBuf[n+1].cmnd        = KH_END
keyholeBuf[n+1].portNo      = 0; //not used
keyholeBuf[n+1].portNo      = 0; //not used
keyholeBuf[n+1].portNo      = 0; //not used
DeviceIoControl(
    drvHnd,                                //handle to driver from open
    PICO_PC_TO_PPC_KEYHOLE,                //this operation
    keyholeBuf,                             //input buffer
    (n+2)*sizeof(keyholeBuf[0]),          //input size
    NULL, 0,                                //output buffer and size
    &oCount,                                //number of bytes written
    NULL                                    //overlapped I/O (not used)
);

```

After this call the PPC will be able to access the message.
The keyhole can also be used to send an eight bit message:

```

keyholeBuf[0].d.portNo      = desired port;
keyholeBuf[0].d.packetType  = //see packetType table
keyholeBuf[0].d.cmnd        = KH_8BIT;    // == 7
keyholeBuf[0].d.byteCount   = value;      //
DeviceIoControl(
    drvHnd,                                //handle to driver from open
    PICO_PC_TO_PPC_KEYHOLE,                //this operation
    keyholeBuf,                             //input buffer
    sizeof(keyholeBuf[0]),                 //input size
    NULL, 0,                                //output buffer and size
    &oCount,                                //number of bytes written
    NULL                                    //overlapped I/O (not used)
);

```

In other words, a plain KH_END is a single byte transfer; A message beginning with KH_BEGIN, containing zero or more 32 bit values, and ending with KH_END is a block transfer.

3.6 Load FPGA operations

PICO_LOAD_FPGA

This call to DeviceIoControl will specify the load file name that is currently loaded into the FPGA. This call **does not load the FPGA**.

The following is typical:

```
char *fileNameP="abc.bit";
```

```

DeviceIoControl(
    drvHnd,                                //handle to driver from open
    PICO_WRITE_PORT,                       //this operation
    fileNameP, strlen(fileNameP),         //input buffer and size
    NULL, 0,                               //output buffer and size
    &oCount,                               //number of bytes written
    NULL                                   //overlapped I/O (not used)
)

```

If the length of the fileNameP is zero the file name will be cleared in Pico.sys.

In the current implementation of pico.sys this call does nothing useful. Loading the Pico E-16 is performed by the driver by writing to channel zero. This call is in support of that activity.

3.7 JTAG Operations

The Pico Cards typically implement a JTAG port which is accessible over the PCMCIA / Cardbus / PICO Express bus. This JTAG port may be used for anything that a JTAG port can be used for except loading the FPGA (for the obvious reason that as soon as the FPGA start to load it is reset and cannot support a JTAG interface). This DeviceIoControl call will send data to the Pico Card over the JTAG interface and retrieve the output from the JTAG interface.

A structure with the following format is passed to the driver:

```

word 0. 32 bit value = options.
word 1. 32 bit value = insert this many bits after data (dvcsBfor)
word 2. 32 bit value = skip this number of bits at the beginning of
TDO data (dvcsAft)
word 3. 32 bit value = 1 for tms data, = 0 for tdi data.
word 4+ n bytes      = tms/tdi data
word 4+n = asciz debugging string.

```

The following call is typical:

```

uint32_t iBuf32[100], oBuf32[100], count32;
buf32[0] = countOfBits;
memmove(&buf32[4], tmsOrTdiData, (bitCount+31)/32);
....
count32 = (bitCount+31)/32;
DeviceIoControl(
    drvHnd,                                //handle to driver from open
    PICO_SETGET_JTAG,                       //this operation
    iBuf32, count32,                         //input buffer and size
    oBuf32, sizeof(oBuf32),                 //output buffer and size
    &oCount,                               //number of bytes written
    NULL                                   //overlapped I/O (not used)
);

```

For each bit in the input stream the driver:

- Outputs 0x10 + TMS*4 + TDI on JTAG port with TCK=0.
- Strobes TCK=1 (high).
- Reads status port and stores 0x10 bit in tdoP.

Data is written back to oBuf32 and oCount returns the number of bytes returned.

The values at buf32[1] (dvcsBfor - devices before the target device), and buf32[2] (dvcsAfter - devices after the target device) are required by the JTAG protocol.

The options field is bit encoded as follows:

Name	Value	Meaning
TMS_BIT	0x04	Data should be sent over TMS port of JTAG
TDI_BIT	0x01	Data should be sent over TDI port of JTAG
JTAG_XIT_BI T	0x80	Signal transition form exit-ir or exit-dr requires an extra bit on TDI line
JTAG_PPC_B0x40 IT	0x40	PPC jtag implementation requires an additional TMS and TDI bit when writing to a register

The JTAG implementations are by no means standard. The kludge represented by the 0x40 and 0x80 bits are manifest.

3.8 Bus Mastering Support calls

Transfers to/from devices or RAM which take advantage of the Bus Mastering capability of the Pico Cards use normal file reads and writes. The calls in this section are in support of this activity.

PICO_SET_PICOADR

This call to DeviceIoControl will set the Pico address for the file device associated with a specific channel. The Pico Address is the address presented to the Pico Card when a Bus Mastering transaction occurs. The following is typical:

```
uint32_t picoAdr=0x124;
DeviceIoControl(
    drvHnd,                                //handle to driver from open
    PICO_SET_PICOADR,                       //this operation
    &picoAdr, sizeof(picoAdr),              //input buffer and size
    NULL,0,                                  //output buffer and size
    &oCount,                                 //number of bytes written
    NULL                                     //overlapped I/O (not used)
)
```

PICO_SETGET_DEVICE_PARAM

This call to DeviceIoControl will set a property of a Pico Channel. The following is typical:

```
uint32_t properties[2];
properties[0] = cPicoChannel::SET_PICOADDR; //the code for the
property
properties[0] = 0x124;                       //the value for the
property
DeviceIoControl(
    drvHnd,                                //handle to driver from open
    PICO_SETGET_DEVICE_PARAM,              //this operation
    &picoAdr, sizeof(picoAdr),              //input buffer and size
    NULL,0,                                  //output buffer and size
    &oCount,                                 //number of bytes written
    NULL                                     //overlapped I/O (not used)
)
```

The acceptable codes for this call are:

Property code	Property	size
cPicoChannel::SET_ACCEPTABLE_RECORDSIZE	uint32_t	4

<code>cPicoChannel::SET_BASEADDR</code>	<code>uint32_t</code>	4
<code>cPicoChannel::SET_DEBUG_FLAGS</code>	<code>uint32_t</code>	4
<code>cPicoChannel::SET_DEVICESIZE</code>	<code>uint32_t</code>	4
<code>cPicoChannel::SET_PICOADDR</code>	<code>uint32_t</code>	4
<code>cPicoChannel::SET_GRANULARITY</code>	<code>uint32_t</code>	4
<code>cPicoChannel::SET_READ_TIMEOUT</code>	<code>PICO_TIMEOUT</code>	8
<code>cPicoChannel::SET_WRITE_TIMEOUT</code>	<code>PICO_TIMEOUT</code>	8
