PCI-AER library interface specification

Adrian M. Whatley and Elisabetta Chicca
20 March 2007
Interface revision 1.18

1. Introduction

The PCI-AER library is a set of low/intermediate level functions useful for accessing and controlling the PCI-AER board. They will be used in the spike-train generation code, in data-logging code and in other programs that need to access the PCI-AER board.

2. Description of the PCI-AER library functions

The driver divides the functionality of the PCI-AER board into separate minor devices for the monitor, sequencer and mapper. The library, and hence this document, follows this organisation. However, some functions are applicable to more than one of these sub-devices, and are documented first to avoid duplication of information.

Function prototypes and structure typedefs are to be found in a header file called pciaerlib.h. Constants are declared in pciaer.h. The library itself is called libpciaer.a.

This document is *not* final. It describes the library interface as of revision 1.21.2.2 of pciaerlib.h.

Unless otherwise described, all functions return 0 for success or an errno value otherwise.

2.1 Common functions applicable to more than one sub-device

int PciaerGetVersionInfo(int handle, pciaer_version_info_t *pvi);

Given a handle to an open sub-device, this function fills the supplied `pciaer_version_info_t` structure pointed to by its argument with the version number of the driver, the contents of the release registers of the two FPGAs and the contents of the S5920's revision identification register (RID). The FPGA release and driver version information is divided within the respective unsigned short into a high byte containing a major version number and low byte containing a minor revision number. This function can be called on any of the sub-devices and requires that the device was opened for read access.

```
typedef struct {
    unsigned short driver_version;
    unsigned short fpga1_release;
    unsigned short fpga2_release;
    unsigned char s5920_revision_id;
} pciaer_version_info_t;
```

int PciaerGetPciInfo(int handle, pciaer_pci_info_t *ppi);

This function fills the supplied `pciaer_pci_info_t` structure pointed to by its argument with information about the PCI bus, slot and device corresponding to the open sub-device indicated by the given handle. Amongst other things, this makes it possible in a multi-PCI-AER card system to determine which handles, and therefore board indices, relate to which physical card. This function can be called on any of the sub-devices and requires that the device was opened for read access.

```
typedef struct {
    unsigned int bus;
    char *slot_name;
    char *device_name;
    unsigned short vendor;
    unsigned short device;
    unsigned short subsys_vendor;
    unsigned short subsys_device;
    unsigned char base_class;
    unsigned char sub_class;
    unsigned char prog_if;
    unsigned char irq;
    unsigned char slot;
    unsigned char func;
} pciaer_pci_info_t;
```

int PciaerSetCounterPeriod(int handle, int period_us);
int PciaerGetCounterPeriod(int handle, int *pPeriod_us);

These two functions deal with the AER Clock Period. The ...Set... function accepts an argument specifying the period in microseconds between counter updates. The only valid values are 1, 10, 50 and 100. The ...Get... function fills the `int` pointed to by its second argument with one of these values according to the current status. These functions can be called using a handle to either the monitor or sequencer sub-device. The ...Set... function requires that the handle was opened for write access; the ...Get... function requires only read access.

int PciaerResetCounter(int handle);
int PciaerResetCounterGetTime(int handle, struct timeval *pResetTime);
int PciaerGetLastCounterResetTime(int handle, struct timeval *pResetTime);

The first two of these three functions reset the counter to 0. The first uses no argument other than a handle opened on either the monitor or sequencer. The second and third functions return in the `struct timeval` pointed to by their second argument the time at which the counter was reset in the same format used by the `gettimeofday` system call. The first two functions require that the handle was opened for write access. The second and third functions require that the handle was opened for read access.

int PciaerGetCounterValue(int handle, unsigned int *pValue);

This function fills the `unsigned int` pointed to by its second argument with a best estimate of the current value of the counter. This function can be called using a handle to either the monitor or sequencer sub-device and requires that the handle was opened for read access.

int PciaerResetFifo(int handle);

This function resets the FIFO belonging to the sub-device to which the handle refers, clearing any events that may be queued. The handle must have been opened for read access on the monitor sub-device or write access on either the sequencer or mapper sub-device.

int PciaerSetArbConfig(int handle, int arbconf);
int PciaerGetArbConfig(int handle, int *pArbconf);

These two functions deal with how many devices are multiplexed into the arbiter. The ...Set... function takes an argument which should be one of the values `PCIAER_IOC_ARB_0_16`, `PCIAER_IOC_ARB_1_15` or `PCIAER_IOC_ARB_2_14`. The names of these constants reflect the way the 16 available bits are split between channel number and actual AE bits. The ...Get... function fills the `int` pointed to by its second argument with one of these values according to the current status. These functions can be called using a handle to either the monitor or mapper sub-device provided that the handle was opened for write access for the ...Set... function, and for read access for the ...Get... function.

int PciaerSetSeqArbChannel(int handle, int ch);
int PciaerGetSeqArbChannel(int handle, int *pCh);

These two functions deal with which channel of the arbiter the sequencer output is connected to. The ...Set... function uses its `ch` argument to specify the number of the arbiter channel (0, 1, 2 or 3) to which the sequencer is connected. The ...Get... function fills the integer pointed to `pCh` with one of these values according to the current status. These functions can be called using a handle to either the sequencer or mapper sub-device provided that the given handle was opened for write access for the ...Set... function and for read access for the ...Get... function.

int PciaerGetFifoDepth(int handle, int *pDepth);

This function fills the integer pointed to by `pDepth` with the depth of the FIFO belonging to the sub-device to which the handle refers, i.e. the number of words (not events or bytes) it can hold. Note that for the monitor FIFO, when time labels are enabled, each event requires three FIFO words, so a monitor FIFO with a depth of 64K could hold a maximum of 21845 complete time-stamped events; and for the sequencer FIFO each event *typically* requires two words, a delay and an address, so a 64K deep sequencer FIFO might hold only 32K words but might hold slightly more if not every event requires a delay. This function requires that the given handle was opened for read access.

int PciaerGetStatistics(int handle, pciaer_stats_t *pStats)
int PciaerResetStatistics(int handle, pciaer_stats_t *pStats

These two functions fill the supplied `pciaer_stats_t` structure pointed to by their second argument with statistics relating to the performance of the sub-device on which they are called. In the case of the ...Reset... function, the statistics are reset after being read, and the pointer argument can be NULL if the values of the statistics before reset are not needed. These functions require that the supplied handle was opened for read access to return the current statistics. Write access is required for the reset function.

```
typedef struct {
    size_t size;
    unsigned long words_transferred;
    unsigned long user_transfers;
    unsigned long total_interrupts;
    unsigned long overflows_underflows;
    unsigned long timeouts;
    unsigned long memory_usage;
} pciaer_stats_t;
```

2.2 Monitor sub-device

int PciaerMonOpen(unsigned int iBboard, int flags, int *pHandle);

Opens the monitor sub-device on the given PCI-AER board, indexed from 0. The flags are the same as those that may be supplied to the system call `open`; the only relevant possibilities here are `O_RDONLY`, `O_WRONLY`, `O_RDWR`, and `O_NONBLOCK`.

Each board's monitor sub-device may only be opened by a single process to ensure that multiple processes cannot read from it simultaneously thus erroneously splitting the incoming data into multiple streams. If the device is already open, PciaerMonOpen will return `EBUSY`. Otherwise no special action is taken on opening the device, since the monitor is always enabled to prevent sending devices (chips) from hanging waiting for a request to be acknowledged. Note that when the monitor device is opened, the monitor FIFO may contain stale, possibly very stale data. Applications must call PciaerResetFifo if they want to be sure of reading recent data. Opening the device will never block.

If and only if the open is successful (i.e. when the return value is 0) a handle to the monitor sub-device is returned in the integer pointed to by the third parameter.

int PciaerMonClose(int handle);

Closes the monitor sub-device referred to by its argument. The device is marked as no longer busy, and can then be re-opened. Otherwise no special action is taken on closing the device, since the monitor is always enabled to prevent sending devices (chips) from hanging waiting for a request to be acknowledged. The monitor FIFO may continue to be filled by incoming address-events.

int PciaerMonReadRaw(int handle, int *p, unsigned int nToRead, unsigned int *pnRead);

PciaerMonReadRaw attempts to read up to `nToRead` 32-bit data words from the monitor sub-device referenced by `handle` into the buffer starting at `p`. If and only if the return value indicates success (0), the number of words actually read is placed in the unsigned int pointed to by `pnRead`.

Both blocking and non-blocking read will be supported. If the monitor FIFO is empty, in the blocking case the function will block, whereas in the non-blocking case the function will return `EAGAIN`. Otherwise, if the Monitor FIFO is not empty, as many 32-bit words as possible will be read from the FIFO and placed into the user's buffer (i.e. until the FIFO becomes empty, or the end of the user's buffer is reached).

The 32–bit data words read from the monitor are formatted as follows:

Value of (*data_word* & MONITOR_DWORD_TYPE_MASK)	**Meaning of (*data_word* & MONITOR_DWORD_DATA_MASK)**
MONITOR_DWORD_AER_ADDR	AE address value
MONITOR_DWORD_TIME_HI	High order word of time at which AE occurred
MONITOR_DWORD_TIME_LO	Low order word of time at which AE occurred
MONITOR_DWORD_ERROR	Error code

Whether or not time values are present in the stream of words read depends on whether time labels are enabled – see the PciaerMonSetTimeLabelFlag and PciaerMonGetTimeLabelFlag functions. If the time values are present, they are in units of the AER clock update period – see the PciaerSetCounterPeriod and PciaerGetCounterPeriod functions.

long PciaerMonRead(int handle, pciaer_monitor_read_ae_t *p, unsigned int nToRead, unsigned int *pnRead);

In contrast to PciaerMonReadRaw, the PciaerMonRead function attempts to read whole events rather than words from the monitor. PciaerMonRead attempts to read up to `nToRead` events from the monitor sub-device referenced by `handle` into the buffer starting at `p`. The number of events actually read is placed in the unsigned int pointed to by `pnRead`. The events read are represented by structures of type `pciaer_monitor_read_ae_t`:

```
typedef struct {
    unsigned int ae;
    unsigned int time_us;
} pciaer_monitor_read_ae_t;
```

In these structures, the time_us field is not valid if time labels are not enabled (see the PciaerMonSetTimeLabelFlag and PciaerMonGetTimeLabelFlag functions) but if time labels are enabled, the times are always in milliseconds, irrespective of the current AER clock update period.

PciaerMonRead exhibits the same blocking / non-blocking behaviour as PciaerMonReadRaw.

Three different kinds of error conditions may be encoded in the long return value. If the return value is 0L there is no error. If the upper half of the the long value is zero and the lower half non-zero, then the lower half represents an errno value resulting from an error being reported from the kernel/driver level. If the lower half of the return value is zero and the upper half non-zero, then the upper half represents a hardware generated error code from a MONITOR_DWORD_ERROR word in the stream of words read from the device. Finally, if both halves of the long are non-zero, the whole long is a negative number representing a protocol error detected by the library. The following such numbers are currently defined:

```
PCOLERR_UNEXPECTED_TIME_HI    -2L
PCOLERR_UNEXPECTED_TIME_LO    -3L
PCOLERR_MISSING_TIME_LBLS     -4L
PCOLERR_MISSING_TIME_LO       -5L
```

long CookWithTimeLabels(const int *pRaw, unsigned int nRaw, pciaer_monitor_read_ae_t *pCooked, unsigned int nToCook, unsigned int *pnUsedRaw, unsigned int *pnCooked);
long CookWithoutTimeLabels(const int *pRaw, unsigned int nRaw, pciaer_monitor_read_ae_t *pCooked, unsigned int nToCook, unsigned int *pnUsedRaw, unsigned int *pnCooked);

These two functions are used internally by PciaerMonRead to translate buffers containing raw streams of words read from the device into buffers containing events, and may also be called by the user to perform the same task. Which function to use depends on whether time labels are present in the raw data. The parameters to the functions are as follows:

`pRaw`	Pointer to source buffer containing raw data
`nRaw`	Number of words available at `pRaw` for translation
`pCooked`	Pointer to destination buffer of event structures
`nToCook`	Maximum number of events to be written to `pCooked`
`pnUsedRaw`	Pointer to an integer receiving the number of raw source words processed
`pnCooked`	Pointer to an integer receiving the number of events placed into the destination buffer

Both functions return a long which encodes error conditions in the same format as described under PciaerMonRead above. If an error occurs, it does not necessarily mean that no words were consumed and no events were produced; the integers pointed to by pnUsedRaw and pnCooked will still be valid.

int PciaerMonSetChannelSel(int handle, int Ch);
int PciaerMonGetChannelSel(int handle, int *pCh);

These two functions deal with which channels are actually monitored. The ...Set... function uses `Ch` to specify a 4-bit mask containing a 1 for each channel which should be monitored, and a 0 for each channel which should not be monitored. The ...Get... function fills the interger pointed to by `pCh` with such a bit mask according to the current status. The ...Set... function requires that the given handle was opened for write access; the ...Get... function requires only read access.

int PciaerMonSetTimeLabelFlag(int handle, int lblflag);
int PciaerMonGetTimeLabelFlag(int handle, int *pLblflag);

These two functions deal with the monitor's Time Label flag. If the ...Set... function is called with the `lblflag` argument zero, time labels will be disabled, MONITOR_DWORD_TIME_HI and MONITOR_DWORD_TIME_LO words will not appear in the raw data stream read by PciaerMonReadRaw, and the times in the event structures delivered by PciaerMonRead will be meaningless. Otherwise, if the `lblflag` argument is not zero, time labels will be enabled,

MONITOR_DWORD_TIME_HI and MONITOR_DWORD_TIME_LO words will appear in the raw data stream read by PciaerMonReadRaw, and the times in the event structures delivered by PciaerMonRead will be meaningful. The ...Get... function sets the `int` pointed to by its `pLblflag` argument to 0 or 1 according to the current status of the flag. The ...Set... function requires that the given handle was opened for write access; the ...Get... function requires only read access.

int PciaerMonGetFifoFlags(int handle, int *pFlags);

This function fills the integer pointed to by `pFlags` with a status word containing a combination of the monitor FIFO flag bits `PCIAER_IOC_MON_EMPTY`, `PCIAER_IOC_MON_HALF_FULL` and `PCIAER_IOC_MON_FULL`, together with some other reserved bits. This call can be used to determine whether a subsequent read call might be able to read a half or a full FIFO's worth of data. Just how much that is depends on the depth of the FIFO. This function requires that the given handle was opened for read access.

2.3 Sequencer sub-device

int PciaerSeqOpen(unsigned int iBoard, int flags, int *pHandle);

Opens the sequencer sub-device on the given PCI-AER board, indexed from 0. The flags are the same as those that may be supplied to the system call `open`; the only relevant possibilities here are `O_RDONLY`, `O_WRONLY`, `O_RDWR`, `O_NONBLOCK` and `O_SYNC`.

Each board's sequencer sub-device may only be opened by a single process to ensure that multiple processes cannot write to it simultaneously thus erroneously interleaving data. If the device is already open, PciaerSeqOpen will return `EBUSY`.

If and only if the open is successful (i.e. when the return value is 0) a handle to the sequencer sub-device is returned in the integer pointed to by the third parameter.

int PciaerSeqClose(int handle);

Closes the sequencer sub-device referred to by its argument. This will block unless and until the sequencer FIFO is empty, or a signal is received. (If this behaviour is not desired, the user should call PciaerResetFifo on the handle before calling PciaerSeqClose.) After this condition is met, the Sequencer hardware (including the sequencer FIFO) is disabled and the device is marked as no longer busy so that it can subsequently be re-opened.

int PciaerSeqWriteRaw(int handle, const unsigned int *p, unsigned int nToWrite, unsigned int *pnWritten);

PciaerSeqWriteRaw attempts to write up to `nToWrite` 32-bit data words to the sequencer sub-device referenced by `handle` from the buffer starting at `p`. If and only if the return value indicates success (0), the number of words actually written is placed in the unsigned int pointed to by `pnWritten`.

Data words to be written to the Sequencer FIFO should be formatted as follows:

Value of (*data_word* & `SEQUENCER_DWORD_TYPE_MASK`)	**Meaning of (*data_word* & `SEQUENCER_DWORD_DATA_MASK`)**
`SEQUENCER_DWORD_END_SEQUENCE`	none
`SEQUENCER_DWORD_AER_ADDR`	AE address value
`SEQUENCER_DWORD_DELAY`	Relative delay in units of current AER Clock Period
`SEQUENCER_DWORD_WAIT_TIME`	Absolute value of counter to wait for

Both blocking and non-blocking write will be supported. If the sequencer FIFO is full, in the blocking case the function will block, whereas in the non-blocking case the function will return `EAGAIN`. If the sequencer FIFO is not full, as many words as possible will be written to the FIFO from the user's buffer (i.e. until the FIFO becomes full or the end of the user's buffer is reached). If the `O_SYNC` file flag is set and `O_NONBLOCK` is not set, the call will not return until all of the words in the user's buffer have been written to the FIFO.

int PciaerSeqWrite(int handle, const pciaer_sequencer_write_ae_t *p, unsigned int nToWrite, unsigned int *pnWritten);

In contrast to PciaerSeqWriteRaw, the PciaerSeqWrite function attempts to write whole events rather than words to the sequencer. PciaerSeqWrite attempts to write up to `nToWrite` events to the sequencer device referenced by `handle` from the buffer starting at `p`. If and only if the return value indicates success (0), the number of events actually written is placed in the unsigned int pointed to by `pnWritten`. The events read are represented by structures of type `pciaer_sequencer_write_ae_t`:

```
typedef struct{
    unsigned int isi_us;
    unsigned int ae;
} pciaer_sequencer_write_ae_t;
```

PciaerSeqWrite exhibits the same blocking / non-blocking behaviour as PciaerSeqWriteRaw.

int PrepareRawWriteBuffer(const pciaer_sequencer_write_ae_t *pEvents, unsigned int nEvents, unsigned int *pRawSeqWordsBuffer, unsigned int nRawSeqBufferWords, unsigned int *pnEventsConverted, unsigned int *pnRawSeqBufferWordsUsed);

This function is used internally by PciaerSeqWrite to translate buffers containing events formatted as described under PciaerSeqWrite above into buffers containing raw sequencer words as required by PciaerSeqWriteRaw and may also be called by the user to perform the same task. The parameters to the function are as follows:

`pEvents`	Pointer to source buffer containing `pciaer_sequencer_write_ae_t` structures
`nEvents`	Number of `pciaer_sequencer_write_ae_t` structures available for translation at `pEvents`.
`pRawSeqWordsBuffer`	Pointer to destination buffer to receive raw sequencer words
`nRawSeqBufferWords`	Maximum number of words to be written to `pRawSeqWordsBuffer`
`pnEventsConverted`	Pointer to an integer receiving the number of complete source event structures for which the function has written a representation into the destination buffer at `pRawSeqWordsBuffer`
`pnRawSeqBufferWordsUsed`	Pointer to an integer receiving the number of words the function has written into the destination buffer at `pRawSeqWordsBuffer`

int PciaerSeqFlush(int handle);

If the sequencer FIFO is not empty, PciaerSeqFlush will block until it is empty.

int PciaerSeqGetFifoFlags(int handle, int *pFlags);

This function fills the integer pointed to by `pFlags` with a status word containing a combination of the sequencer FIFO flag bits `PCIAER_IOC_SEQ_EMPTY`, `PCIAER_IOC_SEQ_HALF_FULL` and `PCIAER_IOC_SEQ_FULL`, together with some other reserved bits. This function requires that the given handle was opened for read access.

2.4. Mapper sub-device

int PciaerMapOpen(unsigned int board, int flags, int *pHandle);

Opens the mapper sub-device on the given PCI-AER board, indexed from 0. The flags are the same as those that may be supplied to the system call `open`; the only relevant possibilities here are `O_RDONLY`, `O_WRONLY` and `O_RDWR`.

Each board's mapper sub-device may only be opened by a single process. If the device is already open, PciaerMapOpen will return `EBUSY`. Otherwise no special action is taken on opening the device, since the mapper hardware is always enabled by default. Opening the device will never block.

If and only if the open is successful (i.e. when the return value is 0) a handle to the mapper sub-device is returned in the integer pointed to by the third parameter.

const volatile pciaer_mapper_sram_t * PciaerMapGetMapperMemory(int handle);

The primary means of access to the mapping tables is intended to be via the various functions of this library whose names end in 'Mapping'. The mapper SRAM can also be read directly by obtaining a pointer to it using this function, but reading from this memory will only produce valid results while mapper output is disabled and the mapper is not busy (see the functions PciaerMapOutputEnable etc.). This direct access is intended primarily for use in debugging, and direct write access is not permitted in order to guarantee the integrity of the mapping tables. For information about the structure of the mapping tables in the mapper SRAM, refer to the hardware documentation.

If the function fails, a null pointer is returned.

int PciaerMapClose(int handle);

Closes the mapper sub-device referred to by its argument. The device is marked as no longer busy, and can then be re-opened. If mapper output has been disabled using PciaerMapOutputDisable, mapper output is re-enabled. Otherwise no special action is taken on closing the device

int PciaerMapSetMapping(int handle, unsigned short source, unsigned short count, const unsigned short *pDestList);

This function establishes a new mapping or replaces the existing mapping from the given `source` AE to the list of `count` destination AEs at `pDestList`. This requires that `handle` was opened for write access. Mapping output is suspended for the duration of the call. If there is insufficient free contiguous mapper memory available on the device, the function will fail returning `ENOSPC`. In this case, calling PciaerMapCompact may help – see below.

int PciaerMapClearMapping(int handle, unsigned short source);

This function deletes all mappings for the given `source` AE. This requires that `handle` was opened for write access. Mapping output is suspended for the duration of the call.

int PciaerMapGetMappingCount(int handle, unsigned short source, unsigned short *pCount);

If and only if this function returns success (0), it places the count of the number of destination AEs for the given `source` AE into the unsigned short pointed to by `pCount`. This requires that `handle` was opened for read access. Mapping output is suspended for the duration of the call.

int PciaerMapGetMapping(int handle, unsigned short source, unsigned short bufsize, unsigned short *pBuffer, unsigned short *pCount);

If and only if this function returns success (0), it places a list of the destination AEs for the given `source` AE into the buffer pointed to by `pBuffer`. The size of the buffer must be specified in terms of sizeof(unsigned short) in `bufsize`. If the buffer is too small to contain the complete list of destination addresses for the given source address, the function returns `EINVAL`. On success, the actual number of valid destination addesses is placed in the unsigned short pointed to by `pCount`. This

function requires that handle was opened for read access. Mapping output is suspended for the duration of the call.

int PciaerMapAddToMapping(int handle, unsigned short source, unsigned short count, const unsigned short *pDestList);
int PciaerMapDeleteFromMapping(int handle, unsigned short source, unsigned short count, const unsigned short *pDestList);

These functions respectively add or delete the list of count destination addresses pointed to by pDestList to or from the current set of destination addresses for the given source AE. They both require that handle was opened for write access. Mapping output is suspended for the duration of the calls. If there is insufficient free contiguous mapper memory available on the device, PciaerMapAddToMapping will fail returning ENOSPC. In this case, calling PciaerMapCompact may help – see below.

int PciaerMapFindNextMapping(int handle, unsigned short *pSource);

Replaces the integer pointed to by pSource with the next source address-event for which a mapping exists after the one specified. This function causes mapping output to be suspended for the duration of the call and requires that handle was opened for read access.

int PciaerMapGetMappingsBitVector(int handle, void *p);

Fills the 8K of memory pointed to by p with a bit vector in which a 0 represents a source address-event for which a mapping does not exist and a 1 represents one for which a mapping does exist. This function causes mapping output to be suspended for the duration of the call and requires that handle was opened for read access.

int PciaerMapClearAllMappings(int handle);

Clears the mapper's memory to a state in which no address-events are mapped. This function causes mapping output to be suspended for the duration of the call and requires that handle was opened for write access.

int PciaerMapCompact(int handle);

Forces maximal compaction of the on-device mapping tables. This function causes mapping output to be suspended for the duration of the call, which may be a considerable period of time. It requires that handle was opened for write access.

int PciaerMapGetFreeSpace(int handle, unsigned int *pWords);

Fills the unsigned int pointed to by pWords with the number of free words in the mapper SRAM. This gives an indication of how many more destination addresses could be stored, but the free space may be fragmented and need compaction before it can be used. This function requires that the handle was opened for read access.

int PciaerMapOutputEnable(int handle);
int PciaerMapOutputDisable(int handle, int wait);
int PciaerMapGetOutputState(int handle, int *pState);

These three functions deal with whether output from the mapper is or is not enabled. By default, mapper output is enabled, but reading or writing the mapper's hardware SRAM contents via the above function calls or via the memory mapped into user space using PciaerMapGetMapperMemory will only produce valid results while mapper output is disabled and the mapper is not busy. When using the library functions to read or write mapping table entries, the driver guarantees that this condition is satisfied by suspending mapping output for the duration of the call. However, when reading the memory mapped into user space using PciaerMapGetMapperMemory, the user program must ensure using these three functions that mapper output is temporarily disabled and idle before it inspects the RAM and must enable it again afterwards.

The ...Enable function uses no argument, other than the handle to the mapper.

The ...Disable function uses its wait argument as follows. If the argument is 0, the call returns immediately after disabling the mapper output, and it is the user's responsibility to determine when the mapper becomes idle. If the argument is non-zero, the call will not return until the mapper has become idle and the results of reading from the RAM will be meaningful.

The ...GetOutputState function fills the integer pointed to by pState with a status word containing a combination of the bits `PCIAER_IOC_MAP_OUT_ENABLED` and `PCIAER_IOC_MAP_OUT_BUSY`. Both bits must be zero before accesses to the mapper RAM are meaningful.

The ...Enable and ...Disable functions both require that the handle was opened for write access; the ...GetOutputState function requires only read access.

int PciaerMapSetChannelSel(int handle, int ch);
int PciaerMapGetChannelSel(int handle, int *pCh);

These two functions deal with which channels' input is actually processed by the mapper. The ...Set... function uses `ch` to specify a 4-bit mask containing a `1` for each input channel which should be processed by the mapper, and a `0` for each channel which should be ignored by the mapper. The ...Get... function fills the integer pointed to by `pCh` with such a bit mask according to the current status. The ...Set... function requires that `handle` was opened for write access to the device; the ...Get... function requires only read access.

int PciaerMapGetFifoFlags(int handle, int *pFlags);

This function fills the integer pointed to by `pFlags` with a status word containing a combination of the mapper FIFO flag bits `PCIAER_IOC_MAP_EMPTY` and `PCIAER_IOC_MAP_FULL`, together with some other reserved bits. This function requires that `handle` was opened for read access.

int PciaerMapGetFifoFillCount(int handle, int *pCount);

This function fills the integer pointed to by `pCount` with the number of times the mapper FIFO has filled since the last reset of the mapper statistics. This function requires that `handle` was opened for read access.

int PciaerMapSetOutputConfig(int handle, int conf);
int PciaerMapGetOutputConfig(int handle, int *pConf);

These two functions deal with the configuration of the mapper output. The ...Set... function uses `conf` to specify one of the values `PCIAER_IOC_MAP_OUT_PASS_THRU`, `PCIAER_IOC_MAP_OUT_1_TO_1` or `PCIAER_IOC_MAP_OUT_1_TO_MANY` for the current mapper output configuration. The ...Get... function fills the integer pointed to by `pConf` with one of these values according to the current configuration. The ...Set... function requires that `handle` was opened for write access; the ...Get... function requires only read access.

int PciaerMapSetDemuxConfig(int handle, int conf);
int PciaerMapGetDemuxConfig(int handle, int *pConf);

These two functions deal with the configuration of the AER demultiplexer on the mapper output. The ...Set... function uses `conf` to specify one of the values `PCIAER_IOC_MAP_DEMUX_0_16`, `PCIAER_IOC_MAP_DEMUX_1_15` or `PCIAER_IOC_MAP_DEMUX_2_14`. The names of these constants reflect the way the 16 address bits are split between channel number and actual AE bits. The ...Get... function fills the integer pointed to by its argument with one of these values according to the current configuration. The ...Set... function requires that `handle` was opened for write access; the ...Get... function requires only read access.

int PciaerMapSetProtocol(int handle, int protocol);
int PciaerMapGetProtocol(int handle, int *pProtocol);

These two functions deal with the type of AER protocol used on the mapper output. The ...Set... function uses `protocol` to specify one of the values `PCIAER_IOC_MAP_AER_P2P` or `PCIAER_IOC_MAP_AER_SCX` indicating the classic point to point, four phase handshake protocol or the shared bus, data only driven on acknowledge protocol respectively. The ...Get... function fills the

integer pointed to by pProtocol with one of these values according to the current status. The ...Set... function requires that `handle` was opened for write access; the ...Get... function requires only read access.

~ o O o ~