PAPI Software Specification

This software specification describes the PAPI 3.0 Release, and is current as of March 08, 2004. It consists of the following sections:

- Introduction to PAPI Constants
- Standardized Event Definitions
- Return Codes
- Low Level API
- High Level API

Introduction to PAPI Constants

The **PAPI** constants are defined in the header files:

```
papiStdEventDefs.h
papi.h
```

The header file **papiStdEventDefs.h** contains platform specific constants. These constants are presented in Table 1: Standardized Event Definitions below. The user should read the documentation that accompanies this table for an explanation of these constants.

The remaining header file **papi.h** contains the **PAPI** Return Codes , among other internal definitions.

Standardized Event Definitions

The following is a table of hardware events deemed relevant and useful in tuning application performance. These events have identical assignments in the header files on different platforms *however they may differ in their actual semantics. In addition, all of these events are not guaranteed to be present on all platforms. Please check your platform's documentation carefully.* Note: these values should not be changed by the user.

Value	Symbol	Description
0x80000000	PAPI_L1_DCM	Level 1 data cache misses
0x80000001	PAPI_L1_ICM	Level 1 instruction cache misses
0x80000002	PAPI_L2_DCM	Level 2 data cache misses
0x80000003	PAPI_L2_ICM	Level 2 instruction cache misses
0x80000004	PAPI_L3_DCM	Level 3 data cache misses
0x80000005	PAPI_L3_ICM	Level 3 instruction cache misses
0x80000006	PAPI_L1_TCM	Level 1 total cache misses
0x80000007	PAPI_L2_TCM	Level 2 total cache misses
0x80000008	PAPI_L3_TCM	Level 3 total cache misses
0x80000009	PAPI_CA_SNP	Snoops
0x8000000A	PAPI_CA_SHR	Request for access to shared cache line (SMP)
0x8000000B	PAPI_CA_CLN	Request for access to clean cache line (SMP)
0x8000000C	PAPI_CA_INV	Cache Line Invalidation (SMP)
0x8000000D	PAPI_CA_ITV	Cache Line Intervention (SMP)
0x8000000E	PAPI_L3_LDM	Level 3 load misses
0x8000000F	PAPI_L3_STM	Level 3 store misses
0x80000010	PAPI_BRU_IDL	Cycles branch units are idle
0x80000011	PAPI_FXU_IDL	Cycles integer units are idle
0x80000012	PAPI_FPU_IDL	Cycles floating point units are idle
0x80000013	PAPI_LSU_IDL	Cycles load/store units are idle
0x80000014	PAPI_TLB_DM	Data translation lookaside buffer misses
0x80000015	PAPI_TLB_IM	Instruction translation lookaside buffer misses
0x80000016	PAPI_TLB_TL	Total translation lookaside buffer misses
0x80000017	PAPI_L1_LDM	Level 1 load misses
0x80000018	PAPI_L1_STM	Level 1 store misses
0x80000019	PAPI_L2_LDM	Level 2 load misses

0x8000001A	PAPI_L2_STM	Level 2 store misses
0x8000001B	PAPI_BTAC_M	BTAC miss
0x8000001C	PAPI_PRF_DM	Prefetch data instruction caused a miss
0x8000001D	PAPI_L3_DCH	Level 3 Data Cache Hit
0x8000001E	PAPI_TLB_SD	Translation lookaside buffer shootdowns (SMP)
0x8000001F	PAPI_CSR_FAL	Failed store conditional instructions
0x80000020	PAPI_CSR_SUC	Successful store conditional instructions
0x80000021	PAPI_CSR_TOT	Total store conditional instructions
0x80000022	PAPI_MEM_SCY	Cycles Stalled Waiting for Memory Access
0x80000023	PAPI_MEM_RCY	Cycles Stalled Waiting for Memory Read
0x80000024	PAPI_MEM_WCY	Cycles Stalled Waiting for Memory Write
0x80000025	PAPI_STL_ICY	Cycles with No Instruction Issue
0x80000026	PAPI_FUL_ICY	Cycles with Maximum Instruction Issue
0x80000027	PAPI_STL_CCY	Cycles with No Instruction Completion
0x80000028	PAPI_FUL_CCY	Cycles with Maximum Instruction Completion
0x80000029	PAPI_HW_INT	Hardware interrupts
0x8000002A	PAPI_BR_UCN	Unconditional branch instructions executed
0x8000002B	PAPI_BR_CN	Conditional branch instructions executed
0x8000002C	PAPI_BR_TKN	Conditional branch instructions taken
0x8000002D	PAPI_BR_NTK	Conditional branch instructions not taken
0x8000002E	PAPI_BR_MSP	Conditional branch instructions mispredicted
0x8000002F	PAPI_BR_PRC	Conditional branch instructions correctly predicted
0x80000030	PAPI_FMA_INS	FMA instructions completed
0x80000031	PAPI_TOT_IIS	Total instructions issued
0x80000032	PAPI_TOT_INS	Total instructions executed
0x80000033	PAPI_INT_INS	Integer instructions executed
0x80000034	PAPI_FP_INS	Floating point instructions executed
0x80000035	PAPI_LD_INS	Load instructions executed
0x80000036	PAPI_SR_INS	Store instructions executed
0x80000037	PAPI_BR_INS	Total branch instructions executed
0x80000038	PAPI_VEC_INS	Vector/SIMD instructions executed
0x80000039	PAPI_FLOPS	Floating Point Instructions executed per second
0x8000003A	PAPI_RES_STL	Cycles processor is stalled on resource
0x8000003B	PAPI_FP_STAL	Cycles any FP units are stalled
0x8000003C	PAPI_TOT_CYC	Total cycles

0x8000003D	PAPI_IPS	Instructions executed per second
0x8000003E	PAPI_LST_INS	Total load/store instructions executed
0x8000003F	PAPI_SYC_INS	Sync. instructions executed
0x80000040	PAPI_L1_DCH	L1 data cache hit
0x80000041	PAPI_L2_DCH	L2 data cache hit
0x80000042	PAPI_L1_DCA	L1 data cache access
0x80000043	PAPI_L2_DCA	L2 data cache access
0x80000044	PAPI_L3_DCA	L3 data cache access
0x80000045	PAPI_L1_DCR	L1 data cache read
0x80000046	PAPI_L2_DCR	L2 data cache read
0x80000047	PAPI_L3_DCR	L3 data cache read
0x80000048	PAPI_L1_DCW	L1 data cache write
0x80000049	PAPI_L2_DCW	L2 data cache write
0x8000004A	PAPI_L3_DCW	L3 data cache write
0x8000004B	PAPI_L1_ICH	L1 instruction cache hits
0x8000004C	PAPI_L2_ICH	L2 instruction cache hits
0x8000004D	PAPI_L3_ICH	L3 instruction cache hits
0x8000004E	PAPI_L1_ICA	L1 instruction cache accesses
0x8000004F	PAPI_L2_ICA	L2 instruction cache accesses
0x80000050	PAPI_L3_ICA	L3 instruction cache accesses
0x80000051	PAPI_L1_ICR	L1 instruction cache reads
0x80000052	PAPI_L2_ICR	L2 instruction cache reads
0x80000053	PAPI_L3_ICR	L3 instruction cache reads
0x80000054	PAPI_L1_ICW	L1 instruction cache writes
0x80000055	PAPI_L2_ICW	L2 instruction cache writes
0x80000056	PAPI_L3_ICW	L3 instruction cache writes
0x80000057	PAPI_L1_TCH	L1 total cache hits
0x80000058	PAPI_L2_TCH	L2 total cache hits
0x80000059	PAPI_L3_TCH	L3 total cache hits
0x8000005A	PAPI_L1_TCA	L1 total cache accesses
0x8000005B	PAPI_L2_TCA	L2 total cache accesses
0x8000005C	PAPI_L3_TCA	L3 total cache accesses
0x8000005D	PAPI_L1_TCR	L1 total cache reads
0x8000005E	PAPI_L2_TCR	L2 total cache reads
0x8000005F	PAPI_L3_TCR	L3 total cache reads

0x80000060	PAPI_L1_TCW	L1 total cache writes
0x80000061	PAPI_L2_TCW	L2 total cache writes
0x80000062	PAPI_L3_TCW	L3 total cache writes
0x80000063	PAPI_FML_INS	Floating Multiply instructions
0x80000064	PAPI_FAD_INS	Floating Add instructions
0x80000065	PAPI_FDV_INS	Floating Divide instructions
0x80000066	PAPI_FSQ_INS	Floating Square Root instructions
0x80000067	PAPI_FNV_INS	Floating Inverse instructions
0x80000068	PAPI_FP_OPS	Floating point operations executed

Table 1: Standardized Event Definitions

Return Codes

All of the functions contained in the **PAPI** APIs return standardized error codes. Values greater than or equal to zero indicate success, less than zero indicates failure.

Value	Symbol	Definition
0	PAPI_OK	No error
-1	PAPI_EINVAL	Invalid argument
-2	PAPI_ENOMEM	Insufficient memory
-3	PAPI_ESYS	A System or C library call failed, please check `errno`
-4	PAPI_ESBSTR	Substrate returned an error, usually the result of an unimplemented feature
-5	PAPI_ECLOST	Access to the counters was lost or interrupted
-6	PAPI_EBUG	Internal error, please send mail to the developers
-7	PAPI_ENOEVNT	Hardware Event does not exist
-8	PAPI_ECNFLCT	Hardware Event exists, but cannot be counted due to counter resource limitations
-9	PAPI_ENOTRUN	No Events or EventSets are currently not counting
-10	PAPI_ISRUN	EventSet is currently running
-11	PAPI_ENOEVST	No such EventSet available
-12	PAPI_ENOTPRESET	Event is not a valid preset
-13	PAPI_ENOCNTR	Hardware does not support performance counters
-14	PAPI_EMISC	'Unknown error' code

Table 2: Return Codes

The Low Level API

The following functions represent the low level portion of the **PAPI API**. These functions provide greatly increased efficiency and functionality over the high level API presented in the next section. As mentioned in the introduction, the low level API is only as powerful as the substrate upon which it is built. Thus some features may not be available on every platform. The converse may also be true, that more advanced features may be available and defined in the header file. The user is encouraged to read the documentation for each platform carefully.

int PAPI_accum (int EventSet, long long *values)

This function accumulates (adds) the running or stopped counters in EventSet into the values array. In addition, it re-initializes the internal counters to zero.

The return value is an integer that indicates whether the call succeeded (PAPI_OK) or failed (not PAPI_OK).

int PAPI_add_event (int EventSet, int Event)

This function modifies an existing EventSet. In PAPI3.0, at any time at most one eventset can be active. Returns the error code PAPI_ENOEVNT if Event cannot be counted on this platform. The addition of a conflicting event to an event set will return an error unless PAPI_SET_MPXRES has been set.

The return value is an integer that indicates whether the call succeeded (a non-negative integer corresponding to the index in the EventInfoArray where this event is stored) or failed (an error code).

int PAPI_add_events (int EventSet, int *Events, int number)

This function modifies an existing EventSet. The Events contained in *Events will be loaded into the EventSet. In PAPI3.0, at any time at most one eventset can be active. Returns the error code PAPI_ENOEVNT if Events cannot be counted on this platform. The addition of a conflicting event to an event set will return an error unless PAPI_SET_MPXRES has been set.

The return value is an integer that indicates whether the call succeeded (a non-negative integer corresponding to the index in the EventInfoArray where this event is stored) or failed (an error code).

int PAPI_add_pevent (int EventSet, int code, void *inout)

This function adds a native programmable Event to an existing EventSet. Such EventSets can only consist of one event, namely that which is specified in this call. Its semantics are very similar to that of ioctl() system call. inout points to an opaque data structure that is specific to the value in code. Higher level macros may be provided in the header file. Please check the documentation for each substrate. This function has a C binding only. (not yet implemented)

The return value is an integer that indicates whether the call succeeded (PAPI_OK) or failed (not PAPI_OK).

int PAPI_cleanup_eventset (int EventSet)

This function effectively cleans the target EventSet. It removes all of the hardware events which have been added to the EventSet. It can then be removed from existence with a call to PAPI_destroy_eventset. The EventSet must be stopped in order for this call to succeed.

The return value is an integer that indicates whether the call succeeded (PAPI_OK) or failed (not PAPI_OK).

int PAPI_create_eventset (int *EventSet)

This function creates a new EventSet for use. This call is not thread safe.????

The return value is an integer that indicates whether the call succeeded (PAPI_OK) or failed (not PAPI_OK).

int PAPI_destroy_eventset (int *EventSet)

This function effectively removes an EventSet from existence. The EventSet must be empty in order for this call to succeed.

The return value is an integer that indicates whether the call succeeded (PAPI_OK) or failed (not PAPI_OK).

int PAPI_enum_event (int *EventCode, int modifier)

This function updates EventCode to next valid value;

modifer can specify {all / available} for presets, or other values for native tables and may be platform specific (Major groups / all mask bits; P / M / E chip, etc) The return value is an integer that indicates whether the call succeeded (PAPI_OK) or failed (not PAPI_OK).

int PAPI_event_code_to_name (int EventCode, char *out)

This function translates an EventCode from the user into the event name used by PAPI. Return code is PAPI_OK if successful, and appropriate error code otherwise.

int PAPI_event_name_to_code (char *in, int *out)

This function translates an event name to PAPI event code. Return value is PAPI_OK if successful and appropriate error code otherwise.

long PAPI_get_dmem_info(int option)

This function returns page size in bytes, Resident set size in pages or Size of process image in pages based on the option passed by the user. If the option is illegal, the function returns PAPI_EINVAL.

int PAPI_get_event_info(int EventCode, PAPI_event_info_t * info)

This function fills into a structure of type PAPI_event_info_t, which contains the description about an event, from the eventcode passed by the user if successful, and error if otherwise.

const PAPI_exe_info_t *PAPI_get_executable_info (void)

This function returns a pointer to a structure of type PAPI_exe_info_t, which contains path, name, start and end addresses for the program's text, data, and bss segments. For the definition of the structure, see papi.h Returns pointer to structure of type PAPI_exe_info_t if successful, and NULL if otherwise.

const PAPI_hw_info_t *PAPI_get_hardware_info (void)

This function returns a pointer to a structure of type PAPI_hw_info_t, which contains number of CPUs, nodes, vendor number/name for CPU, CPU revision, clock speed. For the definition of the structure, see papi.h Returns pointer to structure of type PAPI_hw_info_t if successful, and NULL if otherwise.

int PAPI_get_multiplex(int EventSet)

This function queries the multiplex status of the eventset. This function returns true(non zero) if the eventset’s status is multiplexing, and false(zero) if the eventset’s status is non-multiplexing.

int PAPI_get_opt (int option, PAPI_option_t *ptr)

This function queries the status of tunable options in the PerfAPI Library. "option" is an input/output parameter. The "ptr" structure is for input and output. Not all options fill the PAPI_option_t structure. This function has a C binding only.

The reader is urged to carefully read the PerfAPI Draft for a complete discussion of PAPI_get_opt. The file papi.h contains definitions for the structures unioned in the PAPI_option_t structure.

The return value is an integer that indicates whether the call succeeded (PAPI_OK) or failed (not PAPI_OK).

long long PAPI_get_real_cyc (void)

This function returns a value in cycles, and can be used at the beginning and end of a section of code to calculate number of total cycles elapsed while the section executed. Returns time of type long long if successful, appropriate error code otherwise.

long long PAPI_get_real_usec (void)

This function returns a real (wall) time in micro seconds, and can be used at the beginning and end of a section of code to calculate real time in micro seconds for the section. Returns time of type long long if successful, appropriate error code otherwise.

const PAPI_shlib_info_t *PAPI_get_shared_lib_info(void)

This function returns a pointer to a structure of type PAPI_shlib_info_t, which contains path, name, start and end addresses for the shared libraries used by the program. For the definition of the structure, see papi.h Returns pointer to structure of type PAPI_shlib_info_t if successful, and NULL if otherwise.

int PAPI_get_thr_specific(int tag, void **ptr)

This function returns PAPI_OK if successful, and appropriate error code otherwise. Each thread can store up to PAPI_MAX_THREAD_STORAGE pointers and query them through this function. The pointer is stored in the address pointed by ptr. Tag is used to identify which pointer you want to retrieve.

long long PAPI_get_virt_cyc (void)

This function returns a value in cycles, and can be used at the beginning and end of a section of code to calculate number of virtual (process or thread) cycles elapsed in the section. Returns time of type long long if successful, appropriate error code otherwise.

long long PAPI_get_virt_usec (void)

This function returns a virtual time in micro seconds, and can be used at the beginning and end of a section of code to calculate user time in micro seconds for the section. Returns time of type long long if successful, appropriate error code otherwise.

int PAPI_is_initialized(void)

This function returns the initialization status of PAPI. It returns 0 if PAPI_library_init has not yet been called, or returns PAPI_LOW_LEVEL_INITED if PAPI_library_init is called by low level papi functions, or returns PAPI_HIGH_LEVEL_INITED if PAPI_library_init is called by high level papi functions.

int PAPI_library_init (int version)

This function initializes the PAPI library and has to be called before the low level PAPI can be used. The argument should always be set to PAPI_VER_CURRENT. The reason for this is that you may be linked with a shared library, so this will detect version skew. Don't forget PAPI_thread_init also has to be called before low level PAPI calls can be used in a threaded application.

int PAPI_list_events (int EventSet, int *Events, int *number)

This function decomposes EventSet into the hardware Events it contains. number is both an input and output parameter.

The return value is an integer that indicates whether the call succeeded (PAPI_OK) or failed (not PAPI_OK).

void PAPI_lock (int lock)

Grabs access to the PAPI mutex variable. This function is provided to the user to have a platform independent call to an (hopefully) efficiently implemented mutex. This function has no return value. The user can only set lock to PAPI_USR1_LOCK or PAPI_USR2_LOCK. These two locks are equivalent.

int PAPI_multiplex_init (void)

This function enables and initializes multiplex support in the PAPI library. This allows a user to count more events than total physical counters by time sharing the existing counters at some loss in

precision. Applications that make no use of multiplexing do not need to call this routine.

The return value is an integer that indicates whether the call succeeded (PAPI_OK) or failed (not PAPI_OK).

```
int PAPI_set_multiplex (int EventSet)
```

This function converts a standard EventSet created by a call to PAPI_create_eventset() into an event set capable of handling multiplexed events. This must be done after calling PAPI_multiplex_init() , but prior to calling PAPI_start(). Events can be added to an eventset either before or after converting it into a multiplexed set, but the conversion must be done prior to using it as a multiplexed set.

The return value is an integer that indicates whether the call succeeded (PAPI_OK) or failed (not PAPI_OK).

```
int   PAPI_num_hwctrs(void)
```

This function returns the total physical performance counters.

```
int PAPI_num_events(int EventSet)
```

This function returns the total events in the eventset if successful, appropriate error code otherwise.

```
int PAPI_overflow (int EventSet, int EventCode, int threshold, int flags,
     PAPI_overflow_handler_t handler)
```

This function sets up an EventSet such that when it is PAPI_start()'ed, it begins to register overflows. This EventSet may have multiple events as overflow triggers. To turn off overflow, set the threshold to zero.

The return value is an integer that indicates whether the call succeeded (PAPI_OK) or failed (not PAPI_OK).

```
int PAPI_perror (int code, char *destination, int length)
```

This function copies length worth of the error description string corresponding to code into destination. The resulting string is always null terminated. If length is 0, then the string is printed on stderr. The return value is an integer that indicates whether the call succeeded (PAPI_OK) or failed (not PAPI_OK).

```
int PAPI_profil (unsigned short *buf, unsigned bufsiz, unsigned long offset,
     unsigned scale, int EventSet, int EventCode, int threshold, int flags)
```

This function sets the values in the PAPI_sprofil_t structure, if profiling is to be enabled for this EventSet. The EventSet must be in the stopped state for this call to succeed.

The return value is an integer that indicates whether the call succeeded (PAPI_OK) or failed (not PAPI_OK).

int PAPI_query_event (int EventCode)

This function tests if the event designated by EventCode is supported by the current substrate.

If the answer is yes, the function returns PAPI_OK. If the answer is no, the function returns an error code.

int PAPI_read (int EventSet, long long *values)

This function copies the running or stopped counters in EventSet into the values array. Internal counters will not be re-initialized to zero.

The return value is an integer that indicates whether the call succeeded (PAPI_OK) or failed (not PAPI_OK).

Int PAPI_register_thread(void)

This function notifies PAPI that a thread has 'appeared'. We lookup the thread list, if it does not exist we add it to the list.

The return value is an integer that indicates whether the call succeeded (PAPI_OK) or failed (not PAPI_OK).

int PAPI_remove_event (int EventSet, int EventCode)

This function removes an Event from EventSet.

The return value is an integer that indicates whether the call succeeded (PAPI_OK) or failed (not PAPI_OK).

int PAPI_remove_events (int EventSet, int *Events, int number)

This function removes the events listed in the Events array from EventSet.

The return value is an integer that indicates whether the call succeeded (PAPI_OK) or failed (not PAPI_OK).

int PAPI_reset (int EventSet)

This function initializes the internal counters of the hardware Events contained in EventSet to zero.

The return value is an integer that indicates whether the call succeeded (PAPI_OK) or failed (not PAPI_OK).

int PAPI_restore (void)

PAPI_save and PAPI_restore are for use with external libraries that wish to preserve the state of PAPI and the hardware counters. For instance a C++ instrumentation library will probably want to call PAPI_save() upon entry to it's functions and PAPI_restore() upon exit. These function calls map to whatever is the most efficient on the underlying platform for saving and restoring. Returns PAPI_OK if successful, and appropriate error code otherwise. (not yet implemented)

int PAPI_save (void)

see PAPI_restore description (not yet implemented)

int PAPI_set_debug (int level)

This function sets the default debug level for the PAPI library to one of three debug levels as defined in the papi.h header file. The current debug level is internally stored in the PAPI library and is used by the default internal PAPI error handler subroutine. The error handler is called by library routines on the occurence of recoverable errors.

The return value is an integer that indicates whether the call succeeded (PAPI_OK) or failed (not PAPI_OK).

int PAPI_set_domain (int domain)

This function sets the execution domain in which events are counted. Here domain is one of the constants PAPI_DOM_USER, PAPI_DOM_MIN, PAPI_DOM_KERNEL, PAPI_DOM_OTHER, PAPI_DOM_ALL, PAPI_DOM_MAX, or PAPI_DOM_HWSPEC as defined in the header file. These constants are listed below and can also be found in the online PerfAPI Draft. The return value is an integer that indicates whether the call succeeded (PAPI_OK) or failed (not PAPI_OK).

int PAPI_set_granularity (int granularity)

This function sets the measurement granularity in which the counters function. By default, the granularity is set to the most restrictive supported by the substrate. Returns PAPI_OK if successful, and appropriate error code otherwise. Granularity settings include per thread, process, process group, current cpu, and all cpus. For more information, see papi.h

```
int PAPI_set_opt (int option, PAPI_option_t *ptr)
```

This function sets specific options of the PerfAPI Library, its substrate, or specific EventSets. The PAPI_option_t structure represents a union of all the structures that can be arguments to the different options. In addition, there may exist machine specific options so please check the header file for documentation. This function has a C binding only.

The reader is urged to carefully read the PerfAPI Draft for a complete discussion of PAPI_set_opt. The file papi.h contains definitions for the structures unioned in the PAPI_option_t structure.

The return value is an integer that indicates whether the call succeeded (PAPI_OK) or failed (not PAPI_OK).

```
int   PAPI_set_thr_specific(int tag, void *ptr)
```

This function returns PAPI_OK if successful, and appropriate error code otherwise. The ptr is stored in an array with index tag. The ptr can be read late by PAPI_get_thr_specific function with the same tag parameter.

```
void PAPI_shutdown (void)
```

This is an exit function used by the PAPI Library to free resources and shut down when certain error conditions arise. This call is not necessary, but allows the user the capability to free memory and resources used by the PAPI Library.

The return value is an integer that indicates whether the call succeeded (PAPI_OK) or failed (not PAPI_OK).

```
int PAPI_sprofil (PAPI_sprofil_t *prof, int profcnt, int EventSet,
    int EventCode, int threshold, int flags)
```

This function assumes a pre-initialized PAPI_sprofil_t structure and enables profiling for this EventSet. The EventSet must be in the stopped state for this call to succeed.

The return value is an integer that indicates whether the call succeeded (PAPI_OK) or failed (not PAPI_OK).

int PAPI_start (int EventSet)

This function starts counting all of the hardware events contained in EventSet. All counters are implicitly set to zero before counting.

The return value is an integer that indicates whether the call succeeded (PAPI_OK) or failed (not PAPI_OK).

int PAPI_state (int EventSet, int *status)

This function returns the state of the entire EventSet in status. If the call succeeds, then status is either PAPI_RUNNING or PAPI_STOPPED.

The return value is an integer that indicates whether the call succeeded (PAPI_OK) or failed (not PAPI_OK).

int PAPI_stop (int EventSet, long long *values)

This function terminates the counting of all hardware events contained in EventSet. In addition, the counters contained in that EventSet are copied into the values array.

The return value is an integer that indicates whether the call succeeded (PAPI_OK) or failed (not PAPI_OK).

char *PAPI_strerror (int code)

This function returns the corresponding English error string from the passed code. Returns NULL if code is invalid.

int PAPI_thread_init (unsigned long int (*id_fn)(void))

This function initializes thread support. The argument is a pointer to a function that returns the Thread ID of the currently running thread.

The return value is an integer that indicates whether the call succeeded (PAPI_OK) or failed (not PAPI_OK).

unsigned long int PAPI_thread_id (void)

This function calls the thread id function registered by PAPI_thread_init().

The return value is an unsigned long integer representing the thread id or (unsigned long int)-1.

void PAPI_unlock (int lock)

Unlocks the mutex acquired by a call to **PAPI_lock()** . This function has no return value.

int PAPI_write (int EventSet, long long *values)

This function assigns the values contained in the values array to the internal counters of the Events contained in the EventSet. Returns PAPI_OK if successful, and appropriate error code otherwise.

The High Level API

The simple interface implemented by the following eight routines allows the user to access and count specific hardware events. It should be noted that this API can be used *in conjunction with the low level API.* However, the high level API by itself is only able to access those events countable simultaneously by the underlying hardware. Note that the high level interface performs initialization implicitly and is *not thread safe*. Under the covers it calls PAPI_library_init(PAPI_VER_CURRENT) and PAPI_thread_init(NULL).

int PAPI_accum_counters(long long *values, int array_len)

Add the running counter values to the values in the values array. This call implicitly re-initializes the counters to zero and lets them continue to run upon return. Returns PAPI_OK if successful, and an appropriate error code otherwise.

int PAPI_num_counters(void)

This function returns the optimal length of the values array for the high level functions. This value corresponds to the number of hardware counters supported by the current substrate.

int PAPI_read_counters(long long *values, int array_len)

Read the running counter values into the values array. This call implicitly re-initializes the counters to zero and lets them continue to run upon return. Returns PAPI_OK if successful, and an appropriate error code otherwise.

int PAPI_start_counters(int *events, int array_len)

Start counting the events named in the events array. This function implicitly stops and initializes any counters running as a result of a previous call to PAPI_start_counters(). It is the user's responsibility to choose events that can be counted simultaneously by reading the vendor's documentation. The length of this array should be no longer than PAPI_MAX_EVENTS. Returns PAPI_OK if successful, and an appropriate error code otherwise.

int PAPI_stop_counters(long long *values, int array_len)

Stop the running counters and copy the counts into the values array. This is to be used in conjunction with PAPI_start_counters. Returns PAPI_OK if successful, and an appropriate error code otherwise.

int PAPI_flips(float *rtime, float *ptime, long_long *flpins, float *mflips)

Simplified single call to measure the number of floating point instructions executed and the MegaFlip rate, defined as the number of floating point ***instructions*** per microsecond. Note that not all floating point instructions are created equal: Some platforms implement a floating point multiply/add (FMA) as a single instruction; and a floating point *square root* is often more costly than a floating point *add.* Caution must be used in comparing measurements on different platforms. Each call to PAPI_flips returns the real (clock) time and process time, floating point instructions executed, and MFlips for the period since the last call. A call to PAPI_flips with flpins = -1 resets the counters. Returns PAPI_OK if successful, and an appropriate error code otherwise.

int PAPI_flops(float *rtime, float *ptime, long_long *flpops, float *mflops)

Simplified single call to measure the theoretical floating point operations executed rather than simple instructions , and the MegaFlop rate, defined as the number of floating point ***operations*** per microsecond. It uses the PAPI_FP_OPS event which attempts to 'correctly' account for, e.g., FMA undercounts and FP Store overcounts, etc. Caution must be used in comparing measurements on different platforms. Each call to PAPI_flops returns the real (clock) time and process time, floating point operations executed, and MFlops for the period since the last call. A call to PAPI_flops with flpops = -1 resets the counters. Returns PAPI_OK if successful, and an appropriate error code otherwise.

int PAPI_ipc(float *rtime, float *ptime, long_long *ins, float *ipc)

Simplified single call to measure the information on the instruction rate using the PAPI_TOT_INS event. Returns PAPI_OK if successful, and an appropriate error code otherwise.