PAPI USER'S GUIDE

TABLE OF CONTENTS

TABLE OF CONTENTS 1
PREFACE 4
INTENDED AUDIENCE 4
ORGANIZATION OF THIS DOCUMENT 4
INTRODUCTION TO PAPI 4
INSTALLING PAPI 4
C AND FORTRAN CALLING INTERFACES 4
EVENTS 4
PAPI COUNTER INTERFACES 4
PAPI TIMERS 4
PAPI SYSTEM INFORMATION 5
ADVANCED PAPI FEATURES 5
PAPI ERROR HANDLING 5
PAPI MAILING LISTS 5
APPENDICES 5
DOCUMENT CONVENTION 5
INTRODUCTION TO PAPI 5
WHAT IS PAPI? 5
WHAT'S NEW IN PAPI 3.0? 6
CONVERTING TO PAPI 3 7
BACKGROUND 7
ARCHITECTURE 7
INSTALLING PAPI 9
C AND FORTRAN CALLING INTERFACES 10
EVENTS 11
WHAT ARE EVENTS? 11
NATIVE EVENTS 11
WHAT ARE NATIVE EVENTS? 11
PRESET EVENTS 12
WHAT ARE PRESET EVENTS? 12
EVENT QUERY 13
EVENT TRANSLATION 16

PAPI'S COUNTER INTERFACES 18
HIGH-LEVEL API 18
WHAT IS THE HIGH-LEVEL API? 18
INITIALIZING THE HIGH-LEVEL API 18
EXECUTION RATE CALLS 20
READING, ACCUMULATING, AND STOPPING COUNTERS 21
LOW-LEVEL API 22
WHAT IS THE LOW-LEVEL API? 22
INITIALIZATION OF THE LOW-LEVEL API 23
EVENT SETS 24
WHAT ARE EVENT SETS? 24
CREATING AN EVENT SET 25
ADDING EVENTS TO AN EVENT SET 25
STARTING, READING, ADDING, AND STOPPING EVENTS IN AN EVENT SET 26
RESETTING EVENTS IN AN EVENT SET 28
REMOVING EVENTS IN AN EVENT SET 29
EMPTYING AND DESTROYING AN EVENT SET 30
THE STATE OF AN EVENT SET 31
GETTING AND SETTING OPTIONS 33
SIMPLE CODE EXAMPLES 36
HIGH-LEVEL API 36
LOW-LEVEL API 38
PAPI TIMERS 40
REAL TIME 40
VIRTUAL TIME 41
PAPI SYSTEM INFORMATION 44
EXECUTABLE INFORMATION 44
HARDWARE INFORMATION 45
ADVANCED PAPI FEATURES 48
MULTIPLEXING 48
WHAT IS MULTIPLEXING? 48
USING PAPI WITH MULTIPLEXING 48
ISSUES OF MULTIPLEXING 51
USING PAPI WITH PARALLEL PROGRAMS 51
THREADS 51
MPI 54
OVERFLOW 56
WHAT IS AN OVERFLOW? 56
BEGINNING OVERFLOWS IN EVENT SETS 56

STATISTICAL PROFILING **59**
WHAT IS STATISTICAL PROFILING? *59*
GENERATING A PC HISTOGRAM *59*
PAPI ERROR HANDLING **62**
ERROR CODES **62**
CONVERTING ERROR CODES TO ERROR MESSAGES *62*
PAPI MAILING LISTS **66**
FURTHER INFORMATION **67**
PAPI HOME PAGE **67**
PAPI PROGRAMMER'S REFERENCE **67**
PAPI CONVERSION COOKBOOK **67**
TABLE OF PRESET EVENTS **67**
SUPPORTED PLATFORMS **68**
SUPPORTED TOOLS **68**
HARDWARE REFERENCES **68**
BIBLIOGRAPHY **69**

PREFACE

INTENDED AUDIENCE

This document is intended to provide the PAPI user with a discussion of how to use the different components and functions of PAPI. The intended users are application developers and performance tool writers who need to access performance data to tune and model application performance. The user is expected to have some level of familiarity with either the C or Fortran programming language.

ORGANIZATION OF THIS DOCUMENT

INTRODUCTION TO PAPI

This section provides an introduction to PAPI by describing the project, its motivation, and its architecture.

INSTALLING PAPI

This section provides an installation guide for PAPI. It states the necessary steps in order to install PAPI on the various supported operating systems.

C AND FORTRAN CALLING INTERFACES

This section states the header files in which function calls are defined and the form of the function calls for both the C and Fortran calling interfaces. Also, it provides a table that shows the relation between certain pseudo-types and Fortran variable types.

EVENTS

This section provides an explanation of events as well as an explanation of native and preset events. The preset query and translation functions are also discussed in this section. There are code examples using native events, preset query, and preset translation with the corresponding output.

PAPI COUNTER INTERFACES

This section discusses the high-level and low-level interfaces in detail. The initialization and functions of these interfaces are also discussed. Code examples along with the corresponding output are included as well.

PAPI TIMERS

This section explains the PAPI functions associated with obtaining real and virtual time from the platform's timers. Code examples along with the corresponding output are included as well.

PAPI SYSTEM INFORMATION

This section explains the PAPI functions associated with obtaining hardware and executable information. Code examples along with the corresponding output are included as well.

ADVANCED PAPI FEATURES

This section discusses the advanced features of PAPI, which includes multiplexing, threads, MPI, overflows, and statistical profiling. The functions that are use to implement these features are also discussed. Code examples along with the corresponding output are included as well.

PAPI ERROR HANDLING

This section discusses the various negative error codes that are returned by the PAPI functions. A table with the names, values, and descriptions of the return codes are given as well as a discussion of the PAPI function that can be used to convert error codes to error messages along with a code example with the corresponding output.

PAPI MAILING LISTS

This section provides information on two PAPI mailing lists for the users to ask various questions about the project.

APPENDICES

These appendices provide various listings and tables, such as: a table of preset events and the platforms on which they are supported, a table of PAPI supported tools, more information on native events, multiplexing, overflow, and etc.

DOCUMENT CONVENTION

handle_error(1)

A function that passes the argument of 1. The user should provide this function to handle errors.

INTRODUCTION TO PAPI

WHAT IS PAPI?

PAPI is an acronym for **P**erformance **A**pplication **P**rogramming **I**nterface. The PAPI Project is being developed at the University of Tennessee's Innovative Computing Laboratory in the Computer Science Department. This project was created to design, standardize, and implement a portable and efficient API (Application Programming Interface) to access the hardware performance counters found on most modern microprocessors.

WHAT'S NEW IN PAPI 3.0?

If you're already familiar with PAPI but are new to PAPI 3, there are a number of changes to be aware of. For example,

- ***Better performance***. PAPI 3 has significantly reduced overheads for starting, stopping and reading the counters. Here are some representative timings:

Platform	PAPI_read() – PAPI 2.3.4	PAPI_read() – PAPI 3.0
Altix (Itanium 2 -Madison Chip)	2352 Cycles/Call	1357 Cycles/Call
IBM Power 4	6715 Cycles/Call	4034 Cycles/Call
Itanium 2 (libpfm 2.0)	2929 Cycles/Call	1606 Cycles/Call
Pentium 3 (perfctr 2.4.5)	3023 Cycles/Call	324 Cycles/Call
Pentium 4 (perfctr 2.4.5)	332 Cycles/Call *	401 Cycles/Call
SGI R12k	9636 Cycles/Call	3681 Cycles/Call
Ultrasparc II	3378 Cycles/Call	2150 Cycles/Call

* Implemented as PAPI-3.0 like structure without overlapping eventsets.

- ***Better native event support.*** Native events can now be added to eventsets by name, just like PAPI preset events.
- ***Better thread support.*** Several new functions have been added for handling threads and thread specific data.
- ***Overflow and Profiling enhancements.*** Where supported, overflow and profiling can both now deal with multiple simultaneous events. Profiling also supports larger bin sizes for longer counts.
- ***Myriad bug fixes and code cleanup.*** Which speaks for itself.
- ***You'll be on the upgrade path.*** PAPI 3 is the version we'll be working on from here forward. It's where fixes, new features and new platforms will be supported.

If you intend to convert your code to use PAPI 3 (which we heartily recommend!) there are a few other things you should be aware of:

- ***Overlapping eventsets***. PAPI 3 no longer allows you to nest, or overlap, separate eventsets. This was a feature very few were using, but it caused a great deal of unproductive overhead. We got rid of it.
- ***The time to convert***. Nobody likes to take time to convert APIs. We don't either. That's why we've written this document – to make that process as easy and painless as possible.
- ***Compatibility***. Code rewritten to link to the PAPI 3 library won't any longer link to the PAPI 2 library. If you're a tool builder for whom version independence is an important issue, we've created a PAPIvi layer for version independence. With a little care and a little extra work – and the PAPIvi layer – you should be able to get the same source to link to either PAPI 2.x or PAPI 3.

CONVERTING TO PAPI 3

If you want to convert your application to PAPI 3, a separate document is available on the PAPI website that carefully details this process. This document also includes details on the use of the PAPIvi layer to provide version independent use of PAPI calls.

That document is available in a variety of formats as the PAPI Conversion Cookbook at: http://icl.cs.utk.edu/papi/custom/index.html?lid=49&slid=79

BACKGROUND

Hardware counters exist on every major processor today, such as Intel Pentium, IA-64, AMD Athlon, and IBM POWER series. These counters can provide performance tool developers with a basis for tool development and application developers with valuable information about sections of their code that can be improved. However, there are only a few APIs that allow access to these counters, and most of them are poorly documented, unstable, or unavailable. In addition, performance metrics may have different definitions and different programming interfaces on different platforms.

These considerations motivated the development of the PAPI Project. Some goals of the PAPI Project are as follows:

- To provide a solid foundation for cross platform performance analysis tools
- To present a set of standard definitions for performance metrics on all platforms
- To provide a standardize API among users, vendors, and academics
- To be easy to use, well documented, and freely available

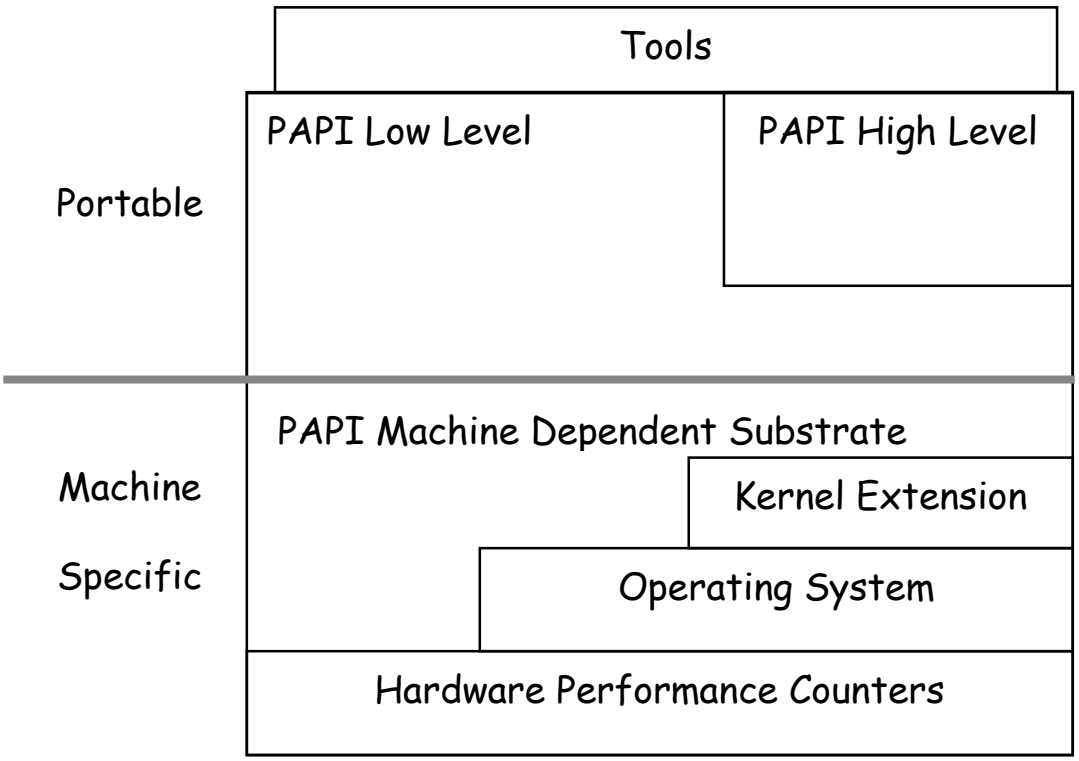
ARCHITECTURE

The **Figure** below shows the internal design of the PAPI architecture. In this figure, we can see the two layers of the architecture:

The **Portable Layer** consists of the API (low level and high level) and machine independent support functions.

The **Machine Specific Layer** defines and exports a machine independent interface to machine dependent functions and data structures. These functions are defined in the substrate layer, which uses kernel extensions, operating system calls, or assembly language to access the hardware performance counters. PAPI uses the most efficient and flexible of the three, depending on what is available.

PAPI strives to provide a uniform environment across platforms. However, this is not always possible. Where hardware support for features, such as overflows and multiplexing is not supported, PAPI implements the features in software where possible. Also, processors do not support the same metrics, thus you can monitor different events depending on the processor in use. Therefore, the interface remains constant, but how it is implemented can vary. Throughout this guide, implementation decisions will be documented where it can make a difference to the user, such as overhead costs, sampling, and etc.


Tools
PAPI Low Level
PAPI High Level
Portable
PAPI Machine Dependent Substrate
Machine
Specific
Kernel Extension
Operating System
Hardware Performance Counters

INSTALLING PAPI

On some of the systems that PAPI supports (see Appendix D), you can install PAPI right out of the box without any additional setup. Others require drivers or patches to be installed first.

The general installation steps are below, but first find your particular Operating System's section of the /papi/INSTALL file for current information on any additional steps that may be necessary.

General Installation

1. Pick the appropriate Makefile.<arch> for your system in the papi source distribution, edit it (if necessary) and compile.

   % make -f Makefile.<arch>

2. Check for errors. Look for the libpapi.a and libpapi.so in the current directory. Optionally, run the test programs in the 'ftests' and 'ctests' directories.

   Not all tests will succeed on all platforms.

   % ./run_tests.sh

   This will run the tests in quiet mode, which will print PASSED, FAILED, or SKIPPED. Tests are SKIPPED if the functionality being tested is not supported by that platform.

3. Create a PAPI binary distribution or install PAPI directly.

   To directly install PAPI from the build tree:

   % make -f Makefile.<arch> DESTDIR=<install-dir> install

   Please use an absolute pathname for <install-dir>, not a relative pathname.

   To create a binary kit, papi-<arch>.tgz:

   % make -f Makefile.<arch> dist

C AND FORTRAN CALLING INTERFACES

PAPI is written in C. The function calls in the C interface are defined in the header file, **papi.h** and consist of the following form:

<returned data type> PAPI_*function_name*(*arg1, arg2*,…)

The function calls in the Fortran interface are defined in the header file, **fpapi.h** and consist of the following form:

PAPIF_*function_name*(*arg1, arg2*, …, check)

As you can probably see, the C function calls have equivalent Fortran function calls (PAPI_<call> becomes PAPIF_<call>). Well, this is true for most function calls, except for the functions that return C pointers to structures, such as PAPI_get_opt and PAPI_get_executable_info, which are either not implemented in the Fortran interface, or implemented with different calling semantics. **In the function calls of the Fortran interface, the return code of the corresponding C routine is returned in the argument, *check*.**

For most architectures, the following relation holds between the pseudo-types listed and Fortran variable types:

Pseudo-type	Fortran type	Description
C_INT	INTEGER	Default Integer type
C_FLOAT	REAL	Default Real type
C_LONG_LONG	INTEGER*8	Extended size integer
C_STRING	CHARACTER*(PAPI_MAX_STR_LEN)	Fortran string
C_INT FUNCTION	EXTERNAL INTEGER FUNCTION	Fortran function returning integer result

Array arguments must be of sufficient size to hold the input/output from/to the subroutine for predictable behavior. The array length is indicated either by the accompanying argument or by internal PAPI definitions.

Subroutines accepting **C_STRING** as an argument are on most implementations capable of reading the character string length as provided by Fortran. In these implementations, the string is truncated or space padded as necessary. For other implementations, the length of the character array is assumed to be of sufficient size. No character string longer than **PAPI_MAX_STR_LEN** is returned by the PAPIF interface.

EVENTS

WHAT ARE EVENTS?

Events are occurrences of specific signals related to a processor's function. Hardware performance counters exist as a small set of registers that count events, such as cache misses and floating point operations while the program executes on the processor. Monitoring these events facilitates correlation between the structure of source/object code and the efficiency of the mapping of that code to the underlying architecture. Each processor has a number of events that are *native* to that architecture. PAPI provides a software abstraction of these architecture-dependent native events into a collection of *preset* events that are accessible through the PAPI interface.

NATIVE EVENTS

WHAT ARE NATIVE EVENTS?

Native events comprise the set of all events that are countable by the CPU. There are generally far more native events available than can be mapped onto PAPI preset events. Even if no preset event is available that exposes a given native event, native events can still be accessed directly. To use native events effectively you should be very familiar with the particular platform in use. PAPI provides access to native events on all supported platforms through the low-level interface. Native events use the same interface as used when setting up a preset event, but since a PAPI preset event definition is not available for native events, a native event name must often be translated into an event code before it can be used.

Native event codes and names are platform dependent, so native codes for one platform most likely will not work for any other platform. To determine the native events for your platform, see the native event lists for the various platforms in the processor architecture manual. Every attempt is made to keep native event names used by PAPI as similar as possible to those used in the vendor documentation. This is not always possible. The test code `native_avail.c` provides insight into the names of the native events for a specific platform.

Native events are specified as arguments to the low-level function, PAPI_add_event in a manner similar to adding PAPI preset events. In the following code example, a native event name is converted to an event code and added to an eventset by using PAPI_add_event:

```
#include <papi.h>
#include<stdio.h>

main()
{
int retval, EventSet = PAPI_NULL;
unsigned int native = 0x0;
PAPI_event_info_t info;

/* Initialize the library */
retval = PAPI_library_init(PAPI_VER_CURRENT);

if (retval != PAPI_VER_CURRENT) {
  printf("PAPI library init error!\n");
  exit(1);
}

if (PAPI_create_eventset(&EventSet) != PAPI_OK)
   handle_error(1);

/* Find the first available native event */
native = NATIVE_MASK | 0;
if (PAPI_get_event_info(native, &info) != PAPI_OK) {
   if (PAPI_enum_event(&native, 0) != PAPI_OK)
      handle_error(1);
}

/* Add it to the eventset */
if (PAPI_add_event(EventSet, native) != PAPI_OK)
   handle_error(1);
}
```

For more code examples using native events, see `ctests/native.c` and `ctests/native_avail.c` in the papi source distribution.

PRESET EVENTS

WHAT ARE PRESET EVENTS?

Preset events, also known as predefined events, are a common set of events deemed relevant and useful for application performance tuning. These events are typically found in many CPUs that provide performance counters and give access to the memory hierarchy, cache coherence protocol events, cycle and instruction counts, functional unit, and pipeline status. Furthermore, preset events are mappings from symbolic names (PAPI preset name) to machine specific definitions (native countable events) for a particular hardware resource. For example, Total Cycles (in user mode) is PAPI_TOT_CYC. Also, PAPI supports presets that may be derived from the underlying hardware metrics. For example, Total L1 Cache Misses (PAPI_L1_TCM) might be the sum of L1 Data Misses and L1 Instruction Misses on a

given platform. A preset can be either directly available as a single counter, derived using a combination of counters, or unavailable on any particular platform.

The PAPI library names approximately 100 preset events, which are defined in the header file, **papiStdEventDefs.h**. For a given platform, a subset of these preset events can be counted though either a simple high-level programming interface or a more complete C or Fortran low-level interface. For a representative list of all the preset events on some supported platforms, see Appendix A. Note that processors and software are revised over time, and this list may not be up to date. To determine exactly which preset events are available on a specific platform, run `ctests/avail.c` in the papi source distribution.

The exact semantics of an event counter are platform dependent. PAPI preset names are mapped onto available events so as to map as many countable events as possible on different platforms. Due to hardware implementation differences, it is not necessarily feasible to directly compare the counts of a particular PAPI preset event obtained on different hardware platforms.

EVENT QUERY

The following low-level functions can be called to query about the existence of a preset or native event (in other words, if the hardware supports that certain event), and to get details about that event:

C:

```
PAPI_query_event(EventCode)
PAPI_get_event_info(EventCode, &info)
PAPI_enum_event(&EventCode, modifier)
```

Fortran:

```
PAPIF_query_event(EventCode, check)
PAPIF_get_event_info(EventCode, symbol, longDescr, shortDescr, count,
note, flags, check)
PAPIF_enum_event(&EventCode, modifier, check)
```

ARGUMENTS

EventCode -- a defined event, such as PAPI_TOT_INS.
symbol -- the event symbol, or name, such as the preset name, PAPI_BR_CN.
longDescr -- a descriptive string for the event of length less than **PAPI_MAX_STR_LEN**.
shortDescr -- a short descriptive string for the event of length less than 18 characters.
count -- zero if the event CANNOT be counted.
note -- additional text information about an event (if available).
flags -- provides additional information about an event, e.g., **PAPI_DERIVED** for an event derived from 2 or more other events.

modifier -- modifies the search criteria; for preset events, returns all events or only available events; for native events, the definition is platform dependent.

PAPI_query_event asks the PAPI library if the preset or native event can be counted on this architecture. If the event CAN be counted, the function returns PAPI_OK. If the event CANNOT be counted, the function returns an error code.

PAPI_get_event_info asks the PAPI library for a copy of an event descriptor. This descriptor can then be used to investigate the details about the event. In Fortran, the individual fields in the descriptor are returned as parameters.

PAPI_enum_event asks the PAPI library to return an event code for the next sequential event based on the current event code and the modifier. This function can be used to enumerate all preset or native events on any platform. See ctests/avail.c or ctests/native_avail.c for details.

EXAMPLE:

```
#include <papi.h>
#include <stdio.h>

main()
{
    int EventSet = PAPI_NULL;
    unsigned int native = 0x0;
    int retval, i;
    PAPI_preset_info_t info;
    PAPI_preset_info_t *infostructs;

    /* Initialize the library */
    retval = PAPI_library_init(PAPI_VER_CURRENT);

    if (retval != PAPI_VER_CURRENT) {
        fprintf(stderr,"PAPI library init error!\n");
        exit(1);
    }

    /* Check to see if the preset, PAPI_TOT_INS, exists */
    if (PAPI_query_event (PAPI_TOT_INS) != PAPI_OK) {
      fprintf (stderr,"No instruction counter? How lame.\n");
      exit(1);
    }

    /* Get details about the preset, PAPI_TOT_INS */
    if (PAPI_get_event_info(PAPI_TOT_INS,&info) != PAPI_OK) {
      fprintf (stderr,"No instruction counter? How lame.\n");
      exit(1);
    }

    if (info.count>0)
        printf ("This event is available on this hardware.\n");

    if (info.flags & PAPI_DERIVED)
        printf ("This event is a derived event on this hardware.\n");

    /* Count the number of available preset events between
       PAPI_TOT_INS and the end of the preset list */
    retval = 0;
    i = PAPI_TOT_INS;
    while (PAPI_enum_event(&i, TRUE) == PAPI_OK) {
       retval++;
    }
    return(retval);
}
```

OUTPUT (IF THE EVENT, PAPI_TOT_INS, IS AVAILABLE ON YOUR SYSTEM):

```
This event is available on this hardware.
```

In the above code example, PAPI_query_event is used to see if a preset (PAPI_TOT_INS) exists, PAPI_get_event_info is used to query details about the event, and PAPI_enum_event is used to count the number of events in the preset list after this preset.

On success, all three of these functions return PAPI_OK, and on error, a non-zero error code is returned.

EVENT TRANSLATION

A preset or native event can be referenced by name or by event code. Most PAPI functions require an event code, while most user input and output is in terms of names. Two low-level functions are provided to translated between these formats:

C:

```
PAPI_event_name_to_code(EventName, EventCode)
PAPI_event_code_to_name(EventCode, EventName)
```

Fortran:

```
PAPIF_event_name_to_code(EventName, EventCode, check)
PAPIF_event_code_to_name(EventCode, EventName, check)
```

ARGUMENTS

EventCode -- a preset or native event of integer type, such as PAPI_TOT_INS.
EventName -- the event name string, such as the preset name, "PAPI_BR_CN".

Note that the preset does not actually have to be available on a given platform to call these functions. Native event names are platform specific and where feasible match those given in the vendor documentation.

`PAPI_event_name_to_code` is used to translate an ASCII PAPI preset or native event name into an integer PAPI event code.

`PAPI_event_code_to_name` is used to translate an integer PAPI event code into an ASCII PAPI preset or native event name.

Using `PAPI_event_code_to_name` in conjunction with `PAPI_enum_event` is a good way to explore the names of native events on a specific platform, as shown in the following code example:

```
#include <papi.h>
#include <stdio.h>

main()
{
    int EventCode, retval;
    char EventCodeStr[PAPI_MAX_STR_LEN];

    /* Initialize the library */
    retval = PAPI_library_init(PAPI_VER_CURRENT);

    if  (retval != PAPI_VER_CURRENT) {
      fprintf(stderr, "PAPI library init error!\n");
      exit(1);
     }

    EventCode = 0 | NATIVE_MASK;
    do {
       /* Translate the integer code to a string */
       if (PAPI_event_code_to_name(EventCode, EventCodeStr) == PAPI_OK)

          /* Print all the native events for this platform */
          printf("Name: %s\nCode: %x\n", EventCodeStr, EventCode);

    } while (PAPI_enum_event(&EventCode, 0) == PAPI_OK);
}
```

OUTPUT:

```
Name: DATA_MEM_REFS
Code: 40000000
Name: DCU_LINES_IN
Code: 40000001
Name: DCU_M_LINES_IN
Code: 40000002
.
.
Name: SEG_REG_RENAMES_TOT
Code: 40000078
Name: RET_SEG_RENAMES
Code: 40000079
```

The output will vary depending on the platform. This was generated on an Intel Pentium III processor.

On success, all the functions return PAPI_OK and on error, a non-zero error code is returned.

PAPI'S COUNTER INTERFACES

HIGH-LEVEL API

WHAT IS THE HIGH-LEVEL API?

The high-level API (**A**pplication **P**rogramming **I**nterface) provides the ability to start, stop, and read the counters for a specified list of events. It is meant for programmers wanting simple event measurements using only PAPI preset events. Earlier versions of the high-level API were also not thread safe, but this restriction has been removed in PAPI 3. Some of the benefits of using the high-level API rather than the low-level API are that it is easier to use and requires less setup (additional calls). This ease of use comes with somewhat higher overhead and loss of flexibility.

It should also be noted that the high-level API can be used in conjunction with the low-level API and in fact does call the low-level API. However, the high-level API by itself is only able to access those events countable simultaneously by the underlying hardware.

There are eight functions that represent the high-level API that allow the user to access and count specific hardware events. *Note that these functions can be accessed from both C and Fortran.* For a code example of using the high-level interface, see Simple Code Examples: High Level API or ctests/high-level.c in the PAPI source distribution.

For full details on the calling semantics of these functions, please refer to the PAPI Programmer's Reference.

INITIALIZING THE HIGH-LEVEL API

The PAPI library is initialized implicitly by several high-level API calls. In addition to the three rate calls discussed later, either of the following two functions also implicitly initializes the library:

C:

```
PAPI_num_counters()
PAPI_start_counters(*events, array_length)
```

Fortran:

```
PAPIF_num_counters(check)
PAPIF_start_counters(*events, array_length, check)
```

ARGUMENTS

`*events` -- an array of codes for events such as PAPI_INT_INS or a native event code.
`array_length` -- the number of items in the *events* array.

PAPI_num_counters returns the optimal length of the *values* array for high-level functions. This value corresponds to the number of hardware counters supported by the current substrate. PAPI_num_counters initializes the PAPI library using PAPI_library_init if necessary.

PAPI_start_counters initializes the PAPI library (if necessary) and starts 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 the *events* array should be no longer than the value returned by PAPI_num_counters.

In the following code example, PAPI_num_counters is used to initialize the library and to get the number of hardware counters available on the system. Also, PAPI_start_counters is used to start counting events:

```
#include <papi.h>

main()
{
    int Events[2] = { PAPI_TOT_CYC, PAPI_TOT_INS };
    int num_hwcntrs = 0;

    /* Initialize the PAPI library and get the number of counters available */
    if ((num_hwcntrs = PAPI_num_counters()) <= PAPI_OK)
        handle_error(1);

    printf("This system has %d available counters.", num_hwcntrs);

    if (num_hwcntrs > 2)
        num_hwcntrs = 2;

     /* Start counting events */
     if (PAPI_start_counters(Events, num_hwcntrs) != PAPI_OK)
         handle_error(1);
}
```

POSSIBLE OUTPUT (VARIES ON DIFFERENT SYSTEMS):

```
This system has 4 available counters.
```

On success, PAPI_num_counters returns the number of hardware counters available on the system and on error, a non-zero error code is returned.

***Optionally*, the PAPI library can be initialized explicitly by using PAPI_library_init. This can be useful if you wish to call PAPI low-level API functions before using the high-level functions.**

EXECUTION RATE CALLS

Three PAPI high-level functions are available to measure floating point or total instruction rates. These three calls are shown below:

C:

```
PAPI_flips(*real_time, *proc_time, *flpins, *mflips)
PAPI_flops(*real_time, *proc_time, *flpins, *mflops)
PAPI_ipc(*real_time, *proc_time, *ins, *ipc)
```

Fortran:

```
PAPIF_flips(real_time, proc_time, flpins, mflips, check)
PAPIF_flops(real_time, proc_time, flpins, mflops, check)
PAPIF_ipc(real_time, proc_time, ins, ipc, check)
```

ARGUMENTS

**real_time* -- the total real (wallclock) time since the first rate call.
**proc_time* -- the total process time since the first rate call.
**flpins* -- the total floating point instructions since the first rate call.
**mflips, *mflops* – Millions of floating point operations or instructions per second achieved since the latest rate call.
**ins* -- the total instructions executed since the first PAPI_ipc call.
**ipc* – instructions per cycle achieved since the latest PAPI_ipc call.

The first execution rate call initializes the PAPI library if needed, sets up the counters to monitor either PAPI_FP_INS, PAPI_FP_OPS or PAPI_TOT_INS (depending on the call), and PAPI_TOT_CYC events, and starts the counters. Subsequent calls to the same rate function will read the counters and return total real time, total process time, total instructions or operations, and the appropriate rate of execution since the last call. A call to PAPI_stop_counters will reinitialize all values to 0. Sequential calls to different execution rate functions will return an error.

Note that on many platforms there may be subtle differences between floating point instructions and operations. Instructions are typically those execution elements most directly measured by the hardware counters. They may include floating point load and store instructions, and may count instructions such as FMA as one, even though two floating point operations have occurred. Consult the hardware documentation for your system for more details. Operations represent a derived value where an attempt is made, when possible, to more closely map to the expected definition of a floating point event.

On success, the rate calls return PAPI_OK and on error, a non-zero error code is returned.

For a code example, see ctest/flops.c or ctest/ipc.c in the papi source distribution.

READING, ACCUMULATING, AND STOPPING COUNTERS

Counters can be read, accumulated, and stopped by calling the following high-level functions, respectively:

C:

```
PAPI_read_counters(*values, array_length)
PAPI_accum_counters(*values, array_length)
PAPI_stop_counters(*values, array_length)
```

Fortran:

```
PAPIF_read_counters(*values, array_length, check)
PAPIF_accum_counters(*values, array_length, check)
PAPIF_stop_counters(*values, array_length, check)
```

ARGUMENTS

`*values` -- an array where to put the counter values.
`array_length` -- the number of items in the *values array.

`PAPI_read_counters`, `PAPI_accum_counters` and `PAPI_stop_counters` all capture the values of the currently running counters into the array, *values*. Each of these functions behaves somewhat differently.

`PAPI_read_counters` copies the current counts into the elements of the *values* array and leaves the counters running.

`PAPI_accum_counters` adds the current counts into the elements of the values array and resets the counters to zero, leaving the counters running. Care should be exercised not to mix calls to `PAPI_accum_counters` with calls to the execution rate functions. Such intermixing is likely to produce unexpected results.

`PAPI_stop_counters` stops the counters and copies the current counts into the elements of the *values* array. This call can also be used to reset the rate functions if used with a NULL pointer to the *values* array.

In the following code example, `PAPI_read_counters` and `PAPI_stop_counters` are used to copy and stop event counters in an array, respectively:

```
#include <papi.h>

#define NUM_EVENTS 2

main()
{
    int Events[NUM_EVENTS] = {PAPI_TOT_INS, PAPI_TOT_CYC};
    long_long values[NUM_EVENTS];

    /* Start counting events */
    if (PAPI_start_counters(Events, NUM_EVENTS) != PAPI_OK)
       handle_error(1);

    /* Do some computation here*/

    /* Read the counters */
    if (PAPI_read_counters(values, NUM_EVENTS) != PAPI_OK)
       handle_error(1);

    /* Do some computation here */

    /* Stop counting events */
    if (PAPI_stop_counters(values, NUM_EVENTS) != PAPI_OK)
       handle_error(1);
}
```

On success, all of these functions return PAPI_OK and on error, a non-zero error code is returned.

LOW-LEVEL API

WHAT IS THE LOW-LEVEL API?

The low-level API (**A**pplication **P**rogramming **I**nterface) manages hardware events in user-defined groups called Event Sets. It is meant for experienced application programmers and tool developers wanting fine-grained measurement and control of the PAPI interface. *Unlike the high-level interface, it allows both PAPI preset and native events.* Another features of the low-level API are the ability to obtain information about the executable and the hardware as well as to set options for multiplexing and overflow handling. Some of the benefits of using the low-level API rather than the high-level API are that it increases efficiency and functionality.

It should also be noted that the low-level interface could be used in conjunction with the high-level interface, as long as attention is paid to insure that the PAPI library is initialized prior to the first low-level PAPI call.

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 on every platform and defined in the header file. Therefore, the user is encouraged to read the documentation for each platform carefully. *There are approximately 50 functions that represent the low-*

level API. For a code example of using the low-level interface, see Simple Code Examples: Low-Level API or `ctests/low_level.c` in the PAPI source distribution.

Note that most functions are implemented in both C and Fortran, but some are implemented in only one of these two languages. For full details on the calling semantics of these functions, please refer to the PAPI Programmer's Reference.

INITIALIZATION OF THE LOW-LEVEL API

The PAPI library must be initialized before it can be used. It can be initialized explicitly by calling the following low-level function:

C:

`PAPI_library_init(`*`version`*`)`

Fortran:

`PAPIF_library_init(check)`

ARGUMENT

`version` -- upon initialization, PAPI checks the argument against the internal value of **PAPI_VER_CURRENT** when the library was compiled. This guards against portability problems when updating the PAPI shared libraries on your system.

Note that this function must be called before calling any other low-level PAPI function.

On success, this function returns PAPI_VER_CURRENT.

On error, a *positive* return code other than PAPI_VER_CURRENT indicates a library version mismatch and a *negative* return code indicates an initialization error.

Beginning with PAPI 3.0, there are a number of options for examining the current version number of PAPI:

- PAPI_VERSION produces an integer containing the complete current version including MAJOR, MINOR, and REVISION components. Typically the REVISION component changes with bug fixes or minor enhancements, the MINOR component changes with feature additions or API changes, and the MAJOR component changes with significant API structural changes.
- PAPI_VER_CURRENT contains the MAJOR and MINOR components and is useful for determining library compatibility changes.
- PAPI_VERSION_MAJOR,
- PAPI_VERSION_MINOR,
- PAPI_VERSION_REVISION are macros that extract specified component from the version number.

The following is a code example of using PAPI_library_init to initialize the PAPI library:

```
#include <papi.h>
#include <stdio.h>
int retval;

main()
{
    /* Initialize the PAPI library */
    retval = PAPI_library_init(PAPI_VER_CURRENT);

    if (retval != PAPI_VER_CURRENT && retval > 0) {
      fprintf(stderr,"PAPI library version mismatch!\n");
      exit(1);
    }

    if (retval < 0) {
      fprintf(stderr, "Initialization error!\n");
      exit(1);
    }

    fprintf(stdout, "PAPI Version Number\nMAJOR: %d\nMINOR: %d\n",
       PAPI_MAJOR(retval), PAPI_MINOR(retval));
}
```

OUTPUT FOR PAPI VERSION 3.0

```
PAPI Version Number
MAJOR: 3
MINOR: 0
```

EVENT SETS

WHAT ARE EVENT SETS?

Event Sets are user-defined groups of hardware events (preset or native), which are used in conjunction with one another to provide meaningful information. The user specifies the events to be added to an Event Set, and other attributes, such as: the counting domain (user or kernel), whether or not the events in the Event Set are to be multiplexed, and whether the Event Set is to be used for overflow or profiling. Other settings for the Event Set are maintained by PAPI, such as: what low-level hardware registers to use, the most recently read counter values, and the state of the Event Set (running/not running). Event Sets provide an effective abstraction for the organization of information associated with counting hardware events. The PAPI library manages the memory for Event Sets with a user interface through integer handles to simplify calling conventions. The user is free to allocate and use any number of them provided the substrate can provide the required resources. Only one Event Set can be in active use at any time in a given thread or process.

CREATING AN EVENT SET

An event set can be created by calling the following the low-level function:

C:

```
PAPI_create_eventset (*EventSet)
```

Fortran:

```
PAPIF_create_eventset(EventSet, check)
```

ARGUMENT

`EventSet` -- Address of an integer location to store the new EventSet handle.

Once it has been created, the user may add hardware events to the *EventSet* by calling PAPI_add_event or PAPI_add_events.

On success, this function returns PAPI_OK. On error, a non-zero error code is returned.

For a code example using this function, see the next section.

ADDING EVENTS TO AN EVENT SET

Hardware events can be added to an event set by calling the following the low-level functions:

C:

```
PAPI_add_event(EventSet, EventCode)
PAPI_add_events(EventSet, *EventCode, number)
```

Fortran:

```
PAPIF_add_event(EventSet, EventCode, check)
PAPIF_add_events(EventSet, EventCode, number, check)
```

ARGUMENTS

`EventSet` -- an integer handle for a PAPI Event Set as created by PAPI_create_eventset.
`EventCode` -- a defined event such as PAPI_TOT_INS.
`*EventCode` – address of an array of defined events.
`number` -- an integer indicating the number of events in the array **EventCode*.

`PAPI_add_event` adds a single hardware event to a PAPI event set.

`PAPI_add_events` does the same as `PAPI_add_event`, but for an array of hardware event codes.

In the following code example, the preset event, PAPI_TOT_INS is added to an event set:

```
#include <papi.h>
#include <stdio.h>

main()
{
    int EventSet = PAPI_NULL;
    int retval;

    /* Initialize the PAPI library */
    retval = PAPI_library_init(PAPI_VER_CURRENT);

    if (retval != PAPI_VER_CURRENT) {
      fprintf(stderr, "PAPI library init error!\n");
      exit(1);
    }

    /* Create an EventSet */
    if (PAPI_create_eventset(&EventSet) != PAPI_OK)
      handle_error(1);

    /* Add Total Instructions Executed to our EventSet */
    if (PAPI_add_event(EventSet, PAPI_TOT_INS) != PAPI_OK)
      handle_error(1);
}
```

On success, both of these functions return PAPI_OK and on error, a non-zero error code is returned.

STARTING, READING, ADDING, AND STOPPING EVENTS IN AN EVENT SET

Hardware events in an event set can be started, read, added, and stopped by calling the following low-level functions, respectively:

C:

```
PAPI_start(EventSet)
PAPI_read(EventSet, *values)
PAPI_accum(EventSet, *values)
PAPI_stop(EventSet, *values)
```

Fortran:

```
PAPIF_start(EventSet, check)
PAPIF_read(EventSet, values, check)
PAPIF_accum(EventSet, values, check)
```

PAPIF_stop(*EventSet, values,* check)

ARGUMENTS

EventSet -- an integer handle for a PAPI Event Set as created by PAPI_create_eventset.
**values* -- an array to hold the counter values of the counting events.

PAPI_start starts the counting events in a previously defined event set.

PAPI_read reads (copies) the counters of the indicated event set into the array, *values*. The counters are left counting after the read without resetting.

PAPI_accum adds the counters of the indicated event set into the array, *values*. The counters are reset and left counting after the call of this function.

PAPI_stop stops the counting events in a previously defined event set and return the current events.

The following is a code example of using PAPI_start to start the counting of events in an event set, PAPI_read to read the counters of the same event set into the array *values*, and PAPI_stop to stop the counting of events in the event set:

```
#include <papi.h>
#include <stdio.h>

main()
{
    int retval, EventSet = PAPI_NULL;
    long_long values[1];

    /* Initialize the PAPI library */
    retval = PAPI_library_init(PAPI_VER_CURRENT);

    if (retval != PAPI_VER_CURRENT) {
      fprintf(stderr, "PAPI library init error!\n");
      exit(1);
    }

    /* Create the Event Set */
    if (PAPI_create_eventset(&EventSet) != PAPI_OK)
        handle_error(1);

    /* Add Total Instructions Executed to our EventSet */
    if (PAPI_add_event(EventSet, PAPI_TOT_INS) != PAPI_OK)
        handle_error(1);

    /* Start counting */
    if (PAPI_start(EventSet) != PAPI_OK)
        handle_error(1);

    /* Do some computation here */

    if (PAPI_read(EventSet, values) != PAPI_OK)
        handle_error(1);

    /* Do some computation here */

    if (PAPI_stop(EventSet, values) != PAPI_OK)
        handle_error(1);
}
```

On success, these functions return PAPI_OK and on error, a non-zero error code is returned.

RESETTING EVENTS IN AN EVENT SET

The hardware event counts in an event set can be reset to zero by calling the following low-level function:

C:

`PAPI_reset(`*`EventSet`*`)`

Fortran:

```
PAPI_reset(EventSet, check)
```

ARGUMENT

EventSet -- an integer handle for a PAPI event set as created by PAPI_create_eventset.

Note that the event set must be running or stopped in order to call PAPI_reset.

For example, the *EventSet* in the code example of the previous section could have been reset to zero by adding the following lines:

```
if (PAPI_reset(EventSet) != PAPI_OK)
  handle_error(1);
```

On success, this function returns PAPI_OK and on error, a non-zero error code is returned.

REMOVING EVENTS IN AN EVENT SET

A hardware event and an array of hardware events can be removed from an event set by calling the following low-level functions, respectively:

C:

```
PAPI_remove_event(EventSet, EventCode)
PAPI_remove_events(EventSet, EventCode, number)
```

Fortran:

```
PAPIF_remove_event(EventSet, EventCode, check)
PAPIF_remove_events(EventSet, EventCode, number, check)
```

ARGUMENTS

EventSet -- an integer handle for a PAPI event set as created by PAPI_create_eventset.
EventCode -- a defined event such as PAPI_TOT_INS or a native event.
**EventCode* -- an array of defined events.
number -- an integer indicating the number of events in the array **EventCode*.

PAPI_remove_event removes a single hardware event from a PAPI event set.

PAPI_remove_events, does the same as PAPI_remove_event, but for an array of hardware event codes.

In the following code example, PAPI_remove_event is used to removed the event, PAPI_TOT_INS, from an event set:

```
#include <papi.h>
#include <stdio.h>
main()
{
    int retval, EventSet = PAPI_NULL;

    /* Initialize the PAPI library */
    retval = PAPI_library_init(PAPI_VER_CURRENT);

    if (retval != PAPI_VER_CURRENT) {
      fprintf(stderr, "PAPI library init error!\n");
      exit(1);
    }

    /* Create an EventSet */
    if (PAPI_create_eventset(&EventSet) != PAPI_OK)
      handle_error(1);

    /* Add Total Instructions Executed to our EventSet */
    if (PAPI_add_event(EventSet, PAPI_TOT_INS) != PAPI_OK)
      handle_error(1);

    /* Remove event */
    if (PAPI_remove_event(EventSet, PAPI_TOT_INS) != PAPI_OK)
      handle_error(1);
}
```

On success, these functions return PAPI_OK and on error, a non-zero error code is returned.

EMPTYING AND DESTROYING AN EVENT SET

All the events in an event set can be emptied and destroyed by calling the following low-level functions, respectively:

C:

```
PAPI_cleanup_eventset(EventSet)
PAPI_destroy_eventset(EventSet)
```

Fortran:

```
PAPIF_cleanup_eventset(EventSet, check)
PAPIF_destroy_eventset(EventSet, check)
```

ARGUMENT

EventSet -- an integer handle for a PAPI event set as created by `PAPI_create_eventset`.

Note that the event set must be empty in order to use PAPI_destroy_eventset.

In the following code example, `PAPI_cleanup_eventset` is used to empty all the events from an event set and `PAPI_remove_eventset` is used to deallocate the memory associated with the empty event set:

```
#include <papi.h>
#include <stdio.h>

main()
{
    int retval, EventSet = PAPI_NULL;

    /* Initialize the PAPI library */
    retval = PAPI_library_init(PAPI_VER_CURRENT);

    if (retval != PAPI_VER_CURRENT) {
      fprintf(stderr, "PAPI library init error!\n");
      exit(1);
    }

    /* Create the EventSet */
    if (PAPI_create_eventset(&EventSet) != PAPI_OK)
        handle_error(1);

    /* Add Total Instructions Executed to our EventSet */
    if (PAPI_add_event(&EventSet, PAPI_TOT_INS) != PAPI_OK)
        handle_error(1);

    /* Remove all events in the eventset */
    if (PAPI_cleanup_eventset(&EventSet) != PAPI_OK)
      handle_error(1);

    /* Free all memory and data structures, EventSet must be empty. */
    if (PAPI_destroy_eventset(&EventSet) != PAPI_OK)
      handle_error(1);
}
```

On success, these functions return PAPI_OK and on error, a non-zero error code is returned.

THE STATE OF AN EVENT SET

The counting state of an Event Set can be obtained by calling the following low-level function:

C:

```
PAPI_state(EventSet, *status)
```

Fortran:

```
PAPIF_state(EventSet, status, check)
```

ARGUMENTS

EventSet -- an integer handle for a PAPI event set as created by PAPI_create_eventset.

status -- an integer containing a Boolean combination of one or more of the following nonzero constants as defined in the PAPI header file, papi.h:

PAPI_STOPPED	*EventSet* is stopped
PAPI_RUNNING	*EventSet* is running
PAPI_PAUSED	*EventSet* temporarily disabled by the library
PAPI_NOT_INIT	*EventSet* defined, but not initialized
PAPI_OVERFLOWING	*EventSet* has overflow enabled
PAPI_PROFILING	*EventSet* has profiling enabled
PAPI_MULTIPLEXING	*EventSet* has multiplexing enabled
PAPI_ACCUMULATING	*EventSet* has accumulating enabled
PAPI_ACCUMULATING	*EventSet* has accumulating enabled
PAPI_ HWPROFILING	*EventSet* has hardware profiling enabled

In the following code example, PAPI_state is used to return the counting state of an *EventSet*:

```
#include <papi.h>
#include <stdio.h>

main ()
{
    int retval, status = 0, EventSet = PAPI_NULL;

    /* Initialize the PAPI library */
    retval = PAPI_library_init(PAPI_VER_CURRENT);

    if (retval != PAPI_VER_CURRENT) {
      fprintf(stderr, "PAPI library init error!\n");
      exit(1);
    }

    /* Create the EventSet */
    if (PAPI_create_eventset(&EventSet) != PAPI_OK)
      handle_error(1);

    /* Add Total Instructions Executed to our EventSet */
    if (PAPI_add_event(&EventSet, PAPI_TOT_INS) != PAPI_OK)
      handle_error(1);

    /* Start counting */
    if (PAPI_state(EventSet, &status) != PAPI_OK)
      handle_error(1);

    printf("State is now %d\n", status);

    if (PAPI_start(EventSet) != PAPI_OK)
      handle_error(1);

    if (PAPI_state(EventSet, &status) != PAPI_OK)
      handle_error(1);

    printf("State is now %d\n", status);
}
```

OUTPUT:

```
State is now 1
State is now 2
```

On success, this function returns PAPI_OK and on error, a non-zero error code is returned.

GETTING AND SETTING OPTIONS

The options of the PAPI library or a specific event set can be obtained and set by calling the following low-level functions, respectively:

C:

```
PAPI_get_opt(option, ptr)
PAPI_set_opt(option, ptr)
```

Fortran:

```
PAPIF_get_clockrate(clockrate)
PAPIF_get_domain(EventSet, domain, mode, check)
PAPIF_get_granularity(EventSet, granularity, mode, check)
PAPIF_get_preload(preload, check)
```

ARGUMENTS

option -- is an input parameter describing the course of action. The Fortran calls are implementations of specific options. Possible values are defined in **papi.h** and briefly described below:

Predefined name	**get**	**set**	**Explanation**
General information requests			
PAPI_ CLOCKRATE	x		Return clockrate in MHz.
PAPI_ MAX_CPUS	x		Return number of CPUs.
PAPI_ MAX_HWCTRS	x		Return number of counters.
PAPI_EXEINFO	x		Addresses for text/data/bss.
PAPI_HWINFO	x		Info. about hardware.
PAPI_PRELOAD	x		Get ''LD_PRELOAD'' environment equivalent.
PAPI_SHLIBINFO	x		Get shared library information.
Defaults for the global library			
PAPI_DEFDOM	x	x	Get / Set the default counting domain for newly created event sets.
PAPI_DEFGRN	x		Get / Set the default counting granularity.
PAPI_DEBUG	x	x	Get / Set the PAPI debug state. The available debug states are defined in papi.h. The debug state is available in ptr->debug
Multiplexing control			
PAPI_ MULTIPLEX	x	x	Get / Set options for multiplexing.
Manipulating individual event sets			

PAPI_DOMAIN	x	x	Get / Set domain for a single event set. The event set is specified in ptr->domain.eventset
PAPI_GRANUL			Get / Set granularity for a single event set. The event set is specified in ptr->granularity.eventset. Currently unimplemented.
PAPI_MAXMEM		x	Set the maximum available memory. Currently unimplemented.

ptr -- is a pointer to a structure that acts as both an input and output parameter. It is defined in **papi.h** and below.
EventSet -- input; a reference to an EventSetInfo structure
clockrate -- output; cycle time of this CPU in MHz; *may* be an estimate generated at init time with a quick timing routine
domain -- output; execution domain for which events are counted
granularity -- output; execution granularity for which events are counted
mode -- input; determines if domain or granularity are default or for the current event set
preload -- output; environment variable string for preloading libraries

PAPI_get_opt and PAPI_set_opt query or change the options of the PAPI library or a specific event set created by PAPI_create_eventset. In C interface, these functions pass a pointer to the *PAPI_option_t* structure. Not all options require or return information in this structure. The Fortran interface is a series of calls implementing various subsets of the C interface. **Not all options in C are available in Fortran.** Note that some options, such as PAPI_SET_DOMAIN, are also available as separate entry points in both C and Fortran.

The file, **papi.h**, contains definitions for the structures combined in the *PAPI_option_t* structure. Users should use the definitions in **papi.h** that correspond with the library used.

In the following code example, PAPI_get_opt is used to acquire the option, PAPI_MAX_HWCTRS, of an event set and PAPI_set_opt is used to set the option, PAPI_DOMAIN, to the same event set:

```
#include <papi.h>
#include <stdio.h>

main()
{
    int num, retval, EventSet = PAPI_NULL;
    PAPI_option_t options;

    /* Initialize the PAPI library */
    retval = PAPI_library_init(PAPI_VER_CURRENT);

    if (retval != PAPI_VER_CURRENT) {
      fprintf(stderr, "PAPI library init error!\n");
      exit(1);
    }

    if ((num = PAPI_get_opt(PAPI_MAX_HWCTRS,NULL)) <= 0)
      handle_error();

    printf("This machine has %d counters.0,num);

    if (PAPI_create_eventset(&EventSet) != PAPI_OK)
      handle_error();

    /* Set the domain of this EventSet
       to counter user and kernel modes for this
       process */

    memset(&options,0x0,sizeof(options));

    options.domain.eventset = EventSet;
    options.domain.domain = PAPI_DOM_ALL;
    if (PAPI_set_opt(PAPI_DOMAIN, &options) != PAPI_OK)
      handle_error();
}
```

POSSIBLE OUTPUT (VARIES ON DIFFERENT PLATFORMS):

```
This machine has 4 counters.
```

On success, these functions return PAPI_OK and on error, a non-zero error code is returned.

For more code examples, see `ctests/second.c` or `ctests/third.c` in the PAPI source distribution.

SIMPLE CODE EXAMPLES

HIGH-LEVEL API

The following is a simple code example of using the high-level API:

```
#include <papi.h>

#define NUM_FLOPS 10000
#define NUM_EVENTS 1

main()
{
    int Events[NUM_EVENTS] = {PAPI_TOT_INS};
    long_long values[NUM_EVENTS];

    /* Start counting events */
    if (PAPI_start_counters(Events, NUM_EVENTS) != PAPI_OK)
       handle_error(1);

    /* Defined in tests/do_loops.c in the PAPI source distribution */
    do_flops(NUM_FLOPS);

    /* Read the counters */
    if (PAPI_read_counters(values, NUM_EVENTS) != PAPI_OK)
       handle_error(1);

    printf("After reading the counters: %lld\n",values[0]);

    do_flops(NUM_FLOPS);

    /* Add the counters */
    if (PAPI_accum_counters(values, NUM_EVENTS) != PAPI_OK)
       handle_error(1);
    printf("After adding the counters: %lld\n", values[0]);

    do_flops(NUM_FLOPS);

    /* Stop counting events */
    if (PAPI_stop_counters(values, NUM_EVENTS) != PAPI_OK)
       handle_error(1);

    printf("After stopping the counters: %lld\n", values[0]);
}
```

POSSIBLE OUTPUT:

```
After reading the counters: 441027
After adding the counters: 891959
After stopping the counters: 443994
```

Notice that on the second line (after adding the counters) the value is approximately twice as large as the first line (after reading the counters) because `PAPI_read_counters` resets and leaves the counters running, then `PAPI_accum_counters` adds the value of the current counter into the *values* array.

LOW-LEVEL API

The following is a simple code example that does the same technique as the above example, except it uses the Low-Level API:

```
#include <papi.h>
#include <stdio.h>

#define NUM_FLOPS 10000

main()
{
    int retval, EventSet=PAPI_NULL;
    long_long values[1];

    /* Initialize the PAPI library */
    retval = PAPI_library_init(PAPI_VER_CURRENT);
    if (retval != PAPI_VER_CURRENT) {
      fprintf(stderr, "PAPI library init error!\n");
      exit(1);
    }

    /* Create the Event Set */
    if (PAPI_create_eventset(&EventSet) != PAPI_OK)
        handle_error(1);

    /* Add Total Instructions Executed to our Event Set */
    if (PAPI_add_event(EventSet, PAPI_TOT_INS) != PAPI_OK)
        handle_error(1);

    /* Start counting events in the Event Set */
    if (PAPI_start(EventSet) != PAPI_OK)
        handle_error(1);

    /* Defined in tests/do_loops.c in the PAPI source distribution */
    do_flops(NUM_FLOPS);

    /* Read the counting events in the Event Set */
    if (PAPI_read(EventSet, values) != PAPI_OK)
        handle_error(1);

    printf("After reading the counters: %lld\n",values[0]);

    /* Reset the counting events in the Event Set */
    if (PAPI_reset(EventSet) != PAPI_OK)
      handle_error(1);

    do_flops(NUM_FLOPS);

    /* Add the counters in the Event Set */
    if (PAPI_accum(EventSet, values) != PAPI_OK)
       handle_error(1);
    printf("After adding the counters: %lld\n",values[0]);

    do_flops(NUM_FLOPS);

    /* Stop the counting of events in the Event Set */
    if (PAPI_stop(EventSet, values) != PAPI_OK)
        handle_error(1);

    printf("After stopping the counters: %lld\n",values[0]);
}
```

POSSIBLE OUTPUT:

```
After reading the counters: 440973
After adding the counters: 882256
After stopping the counters: 443913
```

Notice that in order to get the desired results (the second line approximately twice as large as the first line), PAPI_reset was called to reset the counters, since PAPI_read did not reset the counters.

PAPI TIMERS

PAPI timers use the most accurate timers available on the platform in use. These timers can be used to obtain both real and virtual time on each supported platform. The real time clock runs all the time (e.g. a wall clock) and the virtual time clock runs only when the processor is running in user mode.

REAL TIME

Real time can be acquired in clock cycles and microseconds by calling the following low-level functions, respectively:

C:

```
PAPI_get_real_cyc()
PAPI_get_real_usec()
```

Fortran:

```
PAPIF_get_real_cyc(check)
PAPIF_get_real_usec(check)
```

Both of these functions return the total real time passed since some arbitrary starting point and are equivalent to wall clock time. Also, *these functions always succeed (error-free) since they are guaranteed to exist on every PAPI supported platform.*

In the following code example, PAPI_get_real_cyc() and PAPI_get_real_usec() are used to obtain the real time it takes to create an event set in clock cycles and microseconds, respectively:

```
#include <papi.h>

 main()
{
    long_long start_cycles, end_cycles, start_usec, end_usec;
    int EventSet = PAPI_NULL;

    if (PAPI_library_init(PAPI_VER_CURRENT) != PAPI_VER_CURRENT)
      exit(1);

    /* Gets the starting time in clock cycles */
    start_cycles = PAPI_get_real_cyc();

    /* Gets the starting time in microseconds */
    start_usec = PAPI_get_real_usec();

    /*Create an EventSet */
    if (PAPI_create_eventset(&EventSet) != PAPI_OK)
      exit(1);

    /* Gets the ending time in clock cycles */
    end_cycles = PAPI_get_real_cyc();

    /* Gets the ending time in microseconds */
    end_usec = PAPI_get_real_usec();

    printf("Wall clock cycles: %lld\n", end_cycles - start_cycles);
    prinf("Wall clock time in microseconds: %lld\n", end_usec - start_usec);
}
```

POSSIBLE OUTPUT:

```
Wall clock cycles: 100173
Wall clock time in microseconds: 136
```

VIRTUAL TIME

Virtual time can be acquired in clock cycles and microseconds by calling the following low-level functions, respectively:

C:

```
PAPI_get_virt_cyc()
PAPI_get_virt_usec()
```

Fortran:

```
PAPIF_get_virt_cyc(check)
PAPIF_get_virt_usec(check)
```

Both of these functions return the total number of virtual units from some arbitrary starting point. Virtual units accrue every time a process is running in user-mode. *Like the real time counters, these functions always succeed (error-free) since they are guaranteed to exist on every PAPI supported platform.* However, the resolution can be as bad as 1/Hz as defined by the operating system on some platforms.

In the following code example, PAPI_get_virt_cyc() and PAPI_get_virt_usec() are used to obtain the

```
#include <papi.h>

main()
{
    long_long start_cycles, end_cycles, start_usec, end_usec;
    int EventSet = PAPI_NULL;

    if (PAPI_library_init(PAPI_VER_CURRENT) != PAPI_VER_CURRENT)
      exit(1);

    /* Gets the starting time in clock cycles */
    start_cycles = PAPI_get_virt_cyc();

    /* Gets the starting time in microseconds */
    start_usec = PAPI_get_virt_usec();

    /*Create an EventSet */
    if (PAPI_create_eventset(&EventSet) != PAPI_OK)
      exit(1);

    /* Gets the ending time in clock cycles */
    end_cycles = PAPI_get_virt_cyc();

    /* Gets the ending time in microseconds */
    end_usec = PAPI_get_virt_usec();

    printf("Virtual clock cycles: %lld\n", end_cycles - start_cycles);
    prinf("Virtual clock time in microseconds: %lld\n", end_usec - start_usec);
}
```

virtual time it takes to create an event set in clock cycles and microseconds, respectively:

POSSIBLE OUTPUT:

```
Virtual clock cycles: 715408
Virtual clock time in microseconds: 976
```

PAPI SYSTEM INFORMATION

EXECUTABLE INFORMATION

Information about the executable's address space can be obtained by using the following low-level function:

C:

```
PAPI_get_executable_info()
```

Fortran:

```
PAPIF_get_exe_info(fullname, name, text_start, text_end, data_start, data_end,
bss_start, bss_end, lib_preload_env, check)
```

ARGUMENTS

The following arguments are implicit in the structure returned by the C function, or explicitly returned by Fortran:

fullname -- fully qualified path + filename of the executable
name -- filename of the executable with no path information
text_start, text_end -- Start and End addresses of program text segment
data_start, data_end -- Start and End addresses of program data segment
bss_start, bss_end -- Start and End addresses of program bss segment
lib_preload_env -- environment variable for preloading libraries

Note that the arguments, *text_start* and *text_end*, are the only fields that are filled on every architecture.

In C, this function returns a pointer to a structure containing information about the current program, such as the start and end addresses of the text, data, and bss segments.

In Fortran, the fields of the structure are returned explicitly.

In the following code example, PAPI_get_executable_info() is used to acquire information about the start and end addresses of the program's text segment:

```
#include <papi.h>
#include <stdio.h>

main()
{
    const PAPI_exe_info_t *prginfo = NULL;

    if (PAPI_library_init(PAPI_VER_CURRENT) != PAPI_VER_CURRENT)
      exit(1);

    if ((prginfo = PAPI_get_executable_info()) == NULL)
      exit(1);

    printf("Start of user program is at %p\n",prginfo->text_start);
    printf("End of user program is at %p\n",prginfo->text_end);
}
```

POSSIBLE OUTPUT:

```
Start of user program is at 0x4000000000000f20
End of user program is at 0x4000000000034e00
```

In C, on success, the function returns a non-NULL pointer and on error, NULL is returned.

In Fortran, on success, the function returns PAPI_OK and on error, a non-zero error code is returned.

HARDWARE INFORMATION

Information about the system hardware can be obtained by using the following low-level function:

C:

```
PAPI_get_hardware_info()
```

Fortan:

```
PAPIF_get_hardware_info (ncpu, nnodes, totalcpus, vendor,
vendor_string, model, model_string, revision, mhz)
```

ARGUMENTS

The following arguments are implicit in the structure returned by the C function, or explicitly returned by Fortran.

ncpu -- number of CPUs in an SMP Node

nnodes -- number of Nodes in the entire system
totalcpus -- total number of CPUs in the entire system
vendor -- vendor id number of CPU
vendor_string -- vendor id string of CPU
model -- model number of CPU
model_string -- model string of CPU
revision -- Revision number of CPU
mhz -- Cycle time of this CPU; *may* be an estimate generated at initial time with a quick timing routine

In C, this function returns a pointer to a structure containing information about the hardware on which the program runs, such as: the number of CPUs, CPU model information, and the cycle time of the CPU.

In Fortran, the values of the structure are returned explicitly.

Note that if this function were called before `PAPI_library_init`*, it would be undefined*.

In the following code example, `PAPI_get_hardware_info` is used to acquire hardware

```
#include <papi.h>
#include <stdio.h>

main()
{
    const PAPI_hw_info_t *hwinfo = NULL;

    if (PAPI_library_init(PAPI_VER_CURRENT) != PAPI_VER_CURRENT)
      exit(1);

    if ((hwinfo = PAPI_get_hardware_info()) == NULL)
      exit(1);

    printf("%d CPU's at %f Mhz.\n",hwinfo->totalcpus,hwinfo->mhz);
}
```

information about the total number of CPUs and the cycle time of the CPU:

POSSIBLE OUTPUT:

```
1 CPUs at 733.000000 Mhz.
```

In C, on success, this function returns a non-NULL pointer and on error, NULL is returned.

In Fortran, on success, this function returns PAPI_OK and on error, a non-zero error code is returned.

ADVANCED PAPI FEATURES

MULTIPLEXING

WHAT IS MULTIPLEXING?

Multiplexing allows more events to be counted than can be supported by the hardware. When a microprocessor has a limited number of hardware counters, a large application with many hours of run time may require days or weeks of profiling in order to gather enough information on which to base a performance analysis. Multiplexing overcomes this limitation by subdividing the usage of the counter hardware over time (timesharing) among a large number of performance events.

USING PAPI WITH MULTIPLEXING

INITIALIZATION OF MULTIPLEX SUPPORT

Multiplex support in the PAPI library can be enabled and initialized by calling the following low-level function:

C:

```
PAPI_muliplex_init()
```

Fortran:

```
PAPIF_multiplex_init(check)
```

The above function sets up the internal structures to allow more events to be counted than there are physical counters. It does this by timesharing the existing counters at some loss in precision. **This function should be used after calling `PAPI_library_init`.** After this function is called, the user can proceed to use the normal PAPI routines. *It should be also noted that applications that make no use of multiplexing should not call this function.*

On success, this function returns PAPI_OK and on error, a non-zero error code is returned.

For a code example, see the next section.

CONVERTING AN EVENT SET INTO A MULTIPLEXED EVENT SET

In addition, a standard event set can be converted to a multiplexed event set by the calling the following low-level function:

C:

```
PAPI_set_multiplex(EventSet)
```

Fortran:

```
PAPIF_set_multiplex(EventSet)
```

ARGUMENT

EventSet -- an integer handle for a PAPI event set as created by PAPI_create_eventset.

The above function converts a standard PAPI event set created by a call to `PAPI_create_eventset` into an event set capable of handling multiplexed events. **This function must be used after calling `PAPI_multiplex_init` and `PAPI_create_eventset`, but prior to calling `PAPI_start`.** Events can be added to an event set either before or after converting it into a multiplexed set, but the conversion must be done prior to using it as a multiplexed set.

In the following code example, `PAPI_set_multiplex` is used to convert a standard event set into a multiplexed event set:

```
#include <papi.h>

int retval, i, EventSet = PAPI_NULL, max_to_add = 6, j = 0;
long_long *values;
const PAPI_preset_info_t *pset;

main()
{
  /* Initialize the PAPI library */
  retval = PAPI_library_init(PAPI_VER_CURRENT);
  if (retval != PAPI_VER_CURRENT)
    handle_error(1);

  /* Enable and initialize multiplex support */
  if (PAPI_multiplex_init() != PAPI_OK)
    handle_error(1);

 /* Create an EventSet */
  if (PAPI_create_eventset(&EventSet) != PAPI_OK)
    handle_error(1);

 /* Convert the EventSet to a multiplexed event set */
  if (PAPI_set_multiplex(EventSet) != PAPI_OK)
    handle_error(1);

  for (i=0;i<PAPI_MAX_PRESET_EVENTS;i++)
  {
    if ((PAPI_query_event (i | PAPI_PRESET) == PAPI_OK)
    && ((i | PAPI_PRESET) != PAPI_TOT_CYC))
     {
        if (PAPI_add_event(&EventSet, pset->event_code != PAPI_OK)
            handle_error(1);

        if (++j >= max_to_add)
            break;
     }
  }

  values = (long_long *)malloc(max_to_add*sizeof(long_long));
  if (values == NULL)
    handle_error(1);

  /* Start counting events */
  if (PAPI_start(EventSet) != PAPI_OK)
    handle_error(1);
}
```

On success, both functions return PAPI_OK and on error, a non-zero error code is returned.

For more code examples, see `ctests/multiplex1.c` in the papi source distribution.

ISSUES OF MULTIPLEXING

The following are some issues concerning multiplexing that the PAPI user should be aware of:

- **Hardware multiplexing is not supported by all platforms.** On those platforms where it is supported, PAPI takes advantage of it. Otherwise, PAPI implements software multiplexing through the use of a high-resolution interval timer. For more information on which platforms support hardware or software multiplexing, see Appendix H.
- Multiplexing unavoidably incurs a small amount of overhead when switching events. In addition, no single event is measured for the full analysis time. These factors can adversely affect the precision of reported counter values. In other words, the more events that are multiplexed, the more likely that the results will be statistically skewed. The amount of time spent in the measured regions should be greater than the multiplexing time slice times the number of events measured in order to get acceptable results. The time slice is currently fixed at 25000 clock cycles.
- To prevent naïve use of multiplexing by the novice user, the high level API can only access those events countable simultaneously by the underlying hardware, unless a low level function has been called to explicitly enable multiplexing.

USING PAPI WITH PARALLEL PROGRAMS

THREADS

WHAT ARE THREADS?

A thread is an independent flow of instructions that can be scheduled to run by the operating system. Multi-threaded programming is a form of parallel programming where several controlled threads are executing concurrently in the program. All threads execute in the same memory space, and can therefore work concurrently on shared data. Threads can run in parallel on several processors, allowing a single program to divide its work between several processors, thus running faster than a single-threaded program, which runs on only one processor at a time.

PAPI only supports thread level measurements with *kernel or bound threads*, which are threads that have a scheduling entity known and handled by the operating system's kernel. In most cases, such as with SMP or OpenMP complier directives, *bound* threads will be the default. Each thread is responsible for the creation, start, stop, and read of its own counters. When a thread is created, it inherits no PAPI information from the calling thread. There are some threading packages or APIs that can be used to manipulate threads with PAPI, particularly Pthreads and OpenMP. For those using Pthreads, the user should take care to set the scope of each thread to PTHREAD_SCOPE_SYSTEM attribute, unless the system is known to have a non-hybrid thread library implementation.

In addition, PAPI does support *unbound or non-kernel threads*, but the counts will reflect the total events for the process. Measurements that are done in other threads will get all the same values, namely the counts for the total process. For *unbound* threads, it is not necessary to call PAPI_thread_init, which will be discussed in the next section.

When threads are in use, PAPI allows the user to provide a routine to its library that returns the thread ID of the currently running thread (for example, pthreads_self for Pthreads) and this thread ID is used as a lookup function for the internal data structures.

INITIALIZATION OF THREAD SUPPORT

Thread support in the PAPI library can be initialized by calling the following low-level function:

C:

`PAPI_thread_init(`*`handle`*`)`

Fortran:

`PAPIF_thread_init(`*`handle`*`, check)`

ARGUMENTS

`handle` -- Pointer to a routine that returns the current thread ID.

This function should be called only once, just after PAPI_library_init, and before any other PAPI calls. If the function is called more than once, the application will exit. Also, *applications that make no use of threads do not need to call this function*.

C:

```
#include <papi.h>
#include <omp.h>
if (PAPI_thread_init(omp_get_thread_num) != PAPI_OK)
  handle_error(1);
```

Fortran:

```
#include "fpapi.h"
#include "omp.h"
EXTERNAL omp_get_thread_num
C Fortran dictates that in order to a pass a subroutine
C as an argument, the subroutine must be
C declared external!
call PAPIF_thread_init(omp_get_thread_num, error)
```

The following example shows the correct syntax for using PAPI_thread_init with **OpenMP**:

On success, the function, PAPI_thread_init, returns PAPI_OK and on error, a non-zero error code is returned.

For a code example of using PAPI_thread_init with Pthreads, see the next section.

THREAD ID

The identifier of the current thread can be obtained by calling the following low-level function:

C:

```
PAPI_thread_id()
```

Fortran:

```
PAPIF_thread_id(check)
```

This function calls the thread id function registered by PAPI_thread_init and returns an unsigned long integer containing the thread identifier.

In the following code example, PAPI_thread_init and PAPI_thread_id are used to initialize thread support in the PAPI library and to acquire the identifier of the current thread, respectively, with **Pthreads**:

```
#include <papi.h>
#include <pthread.h>

main()
{
    unsigned long int tid;

    if (PAPI_library_init(PAPI_VER_CURRENT) != PAPI_VER_CURRENT)
      exit(1);

    if (PAPI_thread_init(pthread_self) != PAPI_OK)
      exit(1);

    if ((tid = PAPI_thread_id()) == (unsigned long int)-1)
      exit(1);

    printf("Initial thread id is: %lu\n",tid);
}
```

OUTPUT:

```
Initial thread id is: 0
```

On success, this function returns a valid thread identifier and on error, (unsigned long int) –1 is returned.

Three more utility functions related to threads are available in PAPI. These functions allow you to register a newly created thread to make it available for reference by PAPI, and to create and access thread-specific storage in a platform independent fashion for use with PAPI. These functions are shown below:

C:

```
PAPI_register_thread()
PAPI_get_thr_specific(tag, ptr)
PAPI_set_thr_specific(tag, ptr)
```

ARGUMENTS

tag -- Integer value specifying one of 4 storage locations.
ptr -- Pointer to the address of a data structure.

For more code examples of using Pthreads and OpenMP with PAPI, see ctests/zero_pthreads.c and ctests/zero_omp.c in the papi source distribution, respectively. Also, for a code example of using SMP with PAPI, see ctests/zero_smp.c in the papi source distribution.

MPI

MPI is an acronym for **M**essage **P**assing **I**nterface. MPI is a library specification for message-passing, proposed as a standard by a broadly based committee of vendors, implementers, and users. MPI was designed for high performance on both massively parallel machines and on workstation clusters. More information on MPI can be found at http://www-unix.mcs.anl.gov/mpi.

PAPI supports MPI. When using timers in applications that contain multiplexing, profiling, and overflow, MPI uses a default virtual timer and must be converted to a real timer in order to for the application to work properly. Otherwise, the application will exit.

Optionally, the supported tools, Tau and SvPablo, can be used to implement PAPI with MPI.

The following is a code example of using MPI's PI program with PAPI:

```
#include <papi.h>
#include <mpi.h>
#include <math.h>
#include <stdio.h>

int main(argc,argv)
int argc;
char *argv[];
{
  int done = 0, n, myid, numprocs, i, rc, retval, EventSet = PAPI_NULL;
  double PI25DT = 3.14159265358979323846264338327950288419716939937510;
  double mypi, pi, h, sum, x, a;
  long_long values[1] = {(long_long) 0};

  MPI_Init(&argc,&argv);
  MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
  MPI_Comm_rank(MPI_COMM_WORLD,&myid);

  /*Initialize the PAPI library */
  retval = PAPI_library_init(PAPI_VER_CURRENT);
  if (retval != PAPI_VER_CURRENT) {
    fprintf(stderr, "PAPI library init error!\n");
    exit(1);
}

  /* Create an EventSet */
  if (PAPI_create_eventset(&EventSet) != PAPI_OK)
    handle_error(1);

/* Add Total Instructions Executed to our EventSet */
  if (PAPI_add_event(EventSet, PAPI_TOT_INS) != PAPI_OK)
    handle_error(1);

  /* Start counting */
  if (PAPI_start(EventSet) != PAPI_OK)
    handle_error(1);

  while (!done)
  {
    if (myid == 0) {
        printf("Enter the number of intervals: (0 quits) ");
        scanf("%d",&n);
    }
    MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
    if (n == 0) break;

    h   = 1.0 / (double) n;
    sum = 0.0;
    for (i = myid + 1; i <= n; i += numprocs) {
        x = h * ((double)i - 0.5);
        sum += 4.0 / (1.0 + x*x);
    }
    mypi = h * sum;

    MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0,MPI_COMM_WORLD);

    if (myid == 0)
        printf("pi is approximately %.16f, Error is %.16f\n",
               pi, fabs(pi - PI25DT));
    }

   /* Read the counters */
   if (PAPI_read(EventSet, values) != PAPI_OK)
     handle_error(1);

   printf("After reading counters: %lld\n",values[0]);

   /* Start the counters */
   if (PAPI_stop(EventSet, values) != PAPI_OK)
     handle_error(1);

   printf("After stopping counters: %lld\n",values[0]);

   MPI_Finalize();
}
```

POSSIBLE OUTPUT (AFTER ENTERING 50, 75, AND 100 AS INPUT):

```
Enter the number of intervals: (0 quits) 50
pi is approximately 3.1416259869230028, Error is 0.0000333333332097
Enter the number of intervals: (0 quits) 75
pi is approximately 3.1416074684045965, Error is 0.0000148148148034
Enter the number of intervals: (0 quits) 100
pi is approximately 3.1416009869231254, Error is 0.0000083333333323
Enter the number of intervals: (0 quits) 0
After reading counters: 117393
After stopping counters: 122921
```

OVERFLOW

WHAT IS AN OVERFLOW?

An overflow happens when the number of occurrences of a particular hardware event exceeds a specified threshold. PAPI provides the ability to call user-defined handlers when an overflow occurs. This can be done in hardware, if the processor generates an interrupt signal when the counter reaches a specified value, or in software, by setting up a high-resolution interval timer and installing a timer interrupt handler. For software based overflow, PAPI compares the current counter value against the threshold every time the timer interrupt occurs. If the current value exceeds the threshold, then the user's handler is called from within the signal context with some additional arguments. These arguments allow the user to determine which event overflowed, by how much it overflowed, and at what location in the source code the overflow occurred.

Using the same mechanism as for user programmable overflow, PAPI also guards against register precision overflow of counter values. Each counter can potentially be incremented multiple times in a single clock cycle. This fact combined with increasing clock speeds and the small precision of some of the physical counters means that an overflow is likely to occur on platforms where 64-bit counters are not supported in hardware or by the operating system. In those cases, the PAPI implements 64-bit counters in software using the very same mechanism that handles overflow dispatch.

For more information on which platforms support hardware or software overflow, see Appendix I.

BEGINNING OVERFLOWS IN EVENT SETS

An event set can begin registering overflows by calling the following low-level function:

C:

`PAPI_overflow(`*`EventSet, EventCode, threshold, flags, handler`*`)`

ARGUMENTS

EventSet -- a reference to the event set to use
EventCode -- the event to be used for overflow detection
threshold -- the overflow threshold value to use
flags -- bit map that controls the overflow mode of operation. **This is currently not used and should be set to 0**.
handler -- the handler function to call upon overflow

This function marks a specific *EventCode* in an *EventSet* to generate an overflow signal after every threshold events are counted. **Mutiple events within an event set can be programmed to overflow by making successive calls to this function, but only a single overflow handler can be registered.** Subsequent calls to PAPI_overflow replace earlier calls. *To turn off overflow, set the handler to NULL.*

The handler function is a user-supplied callback routine that performs whatever special processing needed to handle the overflow interrupt, including sorting multiple overflowing events from each other. It must conform to the following prototype:

C:

```
PAPI_overflow_handler(EventSet, address, overflow_vector, void
*context)
```

ARGUMENTS

EventSet -- a reference to the event set in use
address – the address of the program counter when the overflow occurred
overflow_vector – a 64-bit vector that specifies which counter(s) generated the overflow. Bit 0 corresponds to counter 0. The handler should be able to deal with multiple overflow bits per call if more than one event may be set to overflow.
context -- a platform dependent structure containing information about the state of the machine when the overflow occurred. This structure is provided for completeness, but can generally be ignored by most users.

In the following code example, PAPI_overflow is used to mark PAPI_TOT_INS in order to generate an overflow signal after every 100,000 counted events:

```
#include <papi.h>
#include <stdio.h>

#define THRESHOLD  100000

int total = 0;      /* total overflows */

void handler(int EventSet, void *address, long_long overflow_vector, void
*context)
{
    fprintf(stderr, "handler(%d) Overflow at %p! vector=0x%llx\n",
        EventSet, address, overflow_vector);
        total++;
}

main()
{
    int retval, EventSet = PAPI_NULL;

    /* Initialize the PAPI library */
    retval = PAPI_library_init(PAPI_VER_CURRENT);
    if (retval != PAPI_VER_CURRENT)
           handle_error(1);

    /* Create the EventSet */
    if (PAPI_create_eventset(&EventSet) != PAPI_OK)
           handle_error(1);

    /* Add Total Instructions Executed to our EventSet */
    if (PAPI_add_event(EventSet, PAPI_TOT_INS) != PAPI_OK)
           handle_error(1);

    /* Call handler every 100000 instructions */
    retval = PAPI_overflow(EventSet, PAPI_TOT_INS, THRESHOLD, 0, handler);
    if (retval != PAPI_OK)
           handle_error(1);

    /* Start counting */
    if (PAPI_start(EventSet) != PAPI_OK)
           handle_error(1);
}
```

On success, this function returns PAPI_OK and on error, a non-zero error code is returned.

For more code examples, see `ctests/overflow.c, ctests/overflow_twoevents.c` or `ctests/overflow_pthreads.c` in the papi source distribution.

STATISTICAL PROFILING

WHAT IS STATISTICAL PROFILING?

Statistical Profiling involves periodically interrupting a running program and examining the program counter at the time of the interrupt. If this is done for a reasonable number of interrupting intervals, the resulting program counter distribution will be statistically representative of the execution profile of the program with respect to the interrupting event. Performance tools like UNIX prof sample the program address with respect to time and hash the value into a histogram. At program completion, the histogram is analyzed and associated with symbolic information contained in the executable. GNU prof in conjunction with the –p option of the GCC compiler performs exactly this analysis using the process time as the interrupting trigger. PAPI aims to generalize this functionality so that a histogram can be generated using any countable hardware event as the basis for the interrupt signal.

GENERATING A PC HISTOGRAM

A PC histogram can be generated on any countable event by calling either of the following low-level functions:

C:

```
PAPI_profil(buf, bufsiz, offset, scale, EventSet, EventCode,
threshold, flags)
PAPI_sprofil(prof, profcnt, EventSet, EventCode, threshold, flags)
```

Fortran:

```
PAPI_profil(buf, bufsiz, offset, scale, EventSet, EventCode,
threshold, flags, check)
```

AGRUMENTS

**buf* -- pointer to profile buffer array.
bufsiz -- number of entries in *buf.
offset -- starting value of lowest memory address to profile.
scale -- scaling factor for bin values.
EventSet -- The PAPI EventSet to profile when it is started.
EventCode -- Code of the Event in the EventSet to profile.
threshold -- threshold value for the Event triggers the handler.
flags -- bit pattern to control profiling behavior. The defined bit values for the flags variable are shown in the table below:

Defined bit	Description
PAPI_PROFIL_POSIX	Default type of profiling.
PAPI_PROFIL_RANDOM	Drop a random 25% of the samples.
PAPI_PROFIL_WEIGHTED	Weight the samples by their value.

PAPI_PROFIL_COMPRESS	Ignore samples if hash buckets get big.
PAPI_PROFIL_BUCKET_16	Save samples in 16-bit hash buckets.
PAPI_PROFIL_ BUCKET_32	Save samples in 32-bit hash buckets.
PAPI_PROFIL_ BUCKET_64	Save samples in 64-bit hash buckets.

`*prof` -- pointer to PAPI_sprofil_t structure.
`profcnt` -- number of buffers for hardware profiling (**reserved**).

PAPI_profil creates a histogram of overflow counts for a specified region of the application code by using its first four parameters to create the data structures needed by PAPI_sprofil and then calls PAPI_sprofil to do the work. PAPI_sprofil assumes a pre-initialized PAPI_sprofil_t structure and enables profiling for the *EventSet* based on its value. **Note that the EventSet must be in the stopped state in order for either call to succeed.**

More than one hardware event can be profiled at the same time by making multiple independent calls to these functions for the same EventSet before calling PAPI_start. This can be useful for the simultaneous generation of profiles of two or more related events, for example L1 cache misses and L2 cache misses.

On success, these functions return PAPI_OK and on error, a non-zero error code is returned.

For more code examples, see profile.c, profile_32.c, profile_twoevents.c or sprofile.c in the ctests directory of the PAPI source distribution.

In the following code example, PAPI_profil is used to generate a PC histogram:

```
#include <papi.h>
#include <stdio.h>

main()
{
     int retval;
     int EventSet = PAPI_NULL;
     unsigned long start, end, length;
     PAPI_exe_info_t *prginfo;
     unsigned short *profbuf;

    /* Initialize the PAPI library */
    retval = PAPI_library_init(PAPI_VER_CURRENT);
    if (retval != PAPI_VER_CURRENT & retval > 0) {
      fprintf(stderr,"PAPI library version mismatch!0);
      exit(1);
    }

    if (retval < 0)
      handle_error(retval);

    if ((prginfo = PAPI_get_executable_info()) == NULL)
      handle_error(1);

    start = (unsigned long)prginfo->text_start;
    end = (unsigned long)prginfo->text_end;
    length = end - start;

    profbuf = (unsigned short *)malloc(length*sizeof(unsigned short));
    if (profbuf == NULL)
      handle_error(1);

    memset(profbuf,0x00,length*sizeof(unsigned short));

    if (PAPI_create_eventset(&EventSet) != PAPI_OK)
      handle_error(retval);

    /* Add Total FP Instructions Executed to our EventSet */
    if (PAPI_add_event(EventSet, PAPI_FP_INS) != PAPI_OK)
      handle_error(retval);

    if (PAPI_profil(profbuf, length, start, 65536, EventSet, PAPI_FP_INS,
    1000000, PAPI_PROFIL_POSIX | PAPI_PROFIL_BUCKET_16) != PAPI_OK)
      handle_error(1);

    /* Start counting */
    if (PAPI_start(EventSet) != PAPI_OK)
      handle_error(1);
}
```

PAPI ERROR HANDLING

ERROR CODES

All of the functions contained in the PAPI library return standardized error codes in which the values that are greater than or equal to zero indicate success and those that are less than zero indicate failure, as shown in the table below:

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 event sets are currently not counting
-10	PAPI_EISRUN	Event Set is currently running
-11	PAPI_ENOEVST	No such event set 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

CONVERTING ERROR CODES TO ERROR MESSAGES

Error codes can be converted to error messages by calling the following low-level functions:

C:

```
PAPI_perror(code, destination, length)
PAPI_strerror(code)
```

Fortan:

```
PAPIF_perror(code, destination, check)
```

ARGUMENTS

code -- the error code to interpret
**destination* -- "the error message in quotes"

`length` -- either 0 or strlen(destination)

PAPI_perror fills the string, *destination,* with the error message corresponding to the error code (*code)*. The 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 to stderr.

PAPI_strerror returns a pointer to the error message corresponding to the error code (*code)*. If the call fails, the function returns a NULL pointer. Otherwise, a non-NULL pointer is returned. **Note that this function is not implemented in Fortran.**

In the following code example, PAPI_perror is used to convert error codes to error messages:

```
#include <papi.h>
#include <stdio.h>

main()
{
    int EventSet = PAPI_NULL;
    int native = 0x0;
    char error_str[PAPI_MAX_STR_LEN];

    /* Initialize the PAPI library */
    retval = PAPI_library_init(PAPI_VER_CURRENT);

    if (retval != PAPI_VER_CURRENT && retval > 0) {
      fprintf(stderr,"PAPI library version mismatch!\n");
      exit(1);
    }

    if ((retval = PAPI_create_eventset(&EventSet)) != PAPI_OK) {
         fprintf(stderr, "PAPI error %d: %s\n",retval,
            PAPI_strerror(retval));
         exit(1);
    }

    /* Add Total Instructions Executed to our EventSet */
    if ((retval = PAPI_add_event(&EventSet, PAPI_TOT_INS)) != PAPI_OK) {
         PAPI_perror(retval,error_str,PAPI_MAX_STR_LEN);
         fprintf(stderr,"PAPI_error %d: %s\n",retval,error_str);
         exit(1);
    }

    /* Add POWER4 native event */
    retval = PAPI_event_name_to_code("PM_LD_MISS_L1", &native);
    if ((retval = PAPI_add_event(&EventSet, native)) != PAPI_OK) {
         /* Dump error string directly to stderr. */
          PAPI_perror(retval,NULL,NULL);
         exit(1);
    }

    /* Start counting */
    if ((retval = PAPI_start(EventSet)) != PAPI_OK)
      handle_error(retval);
}
```

OUTPUT:

```
Invalid argument
```

Notice that the above output was generated from the last call to PAPI_perror.

On success, PAPI_perror returns PAPI_OK and on error, a non-zero error code is returned.

PAPI MAILING LISTS

PAPI has the two following mailing lists for users to ask any questions about the project:

To contact a general users' discussion list for PAPI software:

Send mail to ptools-perfapi@ptools.org

This list is a good place for newbie questions and general conversation about how to use PAPI or tools that use PAPI.

To contact a list of developers of PAPI, performance tools and kernel patches:

Send mail to perfapi-devel@ptools.org

This list is intended for more technical discussions about PAPI. It is intended for developers of PAPI and other performance tools and kernel patches to share observations and insights. Interested hackers are welcomed. All the CVS log messages go here.

To subscribe to either of these mailing lists:

Visit:
http://www.nacse.org/mailman/listinfo/ptools-perfapi or
http://www.nacse.org/mailman/listinfo/perfapi-devel
and follow the instructions.

Subscription instructions can tend to change from time to time, so check the PAPI website at:
http://icl.cs.utk.edu/papi/custom/index.html?lid=50&slid=67
for the latest information on mailing lists.

Should you become hopelessly confused, send mail to the administrator.

To view archives of either of these mailing lists:

Visit
http://www.ptools.org/archive/perfapi/ or
http://www.ptools.org/archive/perfapi-devel/

FURTHER INFORMATION

In addition to the information in this Users Guide on Programming with PAPI, a number of other information sources are available. Most of these are available on the PAPI website. Below we list some of these resources and indicate where they can be found.

PAPI HOME PAGE

The PAPI Project home page can be found at:
http://icl.cs.utk.edu/papi/

PAPI PROGRAMMER'S REFERENCE

Function by function documentation for PAPI can be found in a number of formats.

PAPI man pages are part of the standard installation package. For a properly installed PAPI, you should be able to display a man page for any PAPI function by typing:

```
> man PAPI_xxx
```

where "PAPI_xxx" is a PAPI function name.

In addition, this information is also available in HTML format on the PAPI website at:
http://icl.cs.utk.edu/projects/papi/files/html_man3/papi.html

If you want a printable version of the Programmer's Reference, you can find it in Word, PDF and PostScript formats under the 'Documentation' tab on the PAPI website.

PAPI CONVERSION COOKBOOK

A 'cookbook' to aid in the conversion of PAPI 2 syntax to PAPI 3 syntax can be found in multiple formats under the 'Documentation' tab on the PAPI website.

TABLE OF PRESET EVENTS

A table of present events, their standard definitions, and the platforms on which they are defined can be found at:
http://icl.cs.utk.edu/projects/papi/presets.html

SUPPORTED PLATFORMS

A table of currently supported hardware platforms and operating systems can be found on the PAPI website under the "Supported Platforms" tab.

SUPPORTED TOOLS

A list of tools that support PAPI, along with a list of links to related projects can be found on the PAPI website under the "Links" tab, at the bottom of the page.

HARDWARE REFERENCES

A series of links to vendor and third party hardware documentation on performance counter resources can also be found on the PAPI website under the "Links" tab.

BIBLIOGRAPHY

Browne, S., J. Dongarra J., Garner N., London K., and Mucci, P., "A Portable Programming Interface for Performance Evaluation on Modern Processors," University of Tennessee Technical Report, Knoxville, Tennessee, July 2000. http://icl.cs.utk.edu/papi/documents

Browne, S., Dongarra J., Garner N., London K., and Mucci, P., "A Scalable Cross-Platform Infrastructure for Application Performance Tuning Using Hardware Counters," Proc. *SC'2000*, November 2000. http://icl.cs.utk.edu/papi/documents

Dongarra, J., London, K., Moore, S., Mucci, P., and Terpstra, D. "Using PAPI for Hardware Performance Monitoring on Linux Systems," *Conference on Linux Clusters: The HPC Revolution*, Urbana, Illinois, June 25-27, 2001. http://icl.cs.utk.edu/papi/documents

London, K., Moore, S., Mucci, P., Seymour, K., and Luczak, R. "The PAPI Cross-Platform Interface to Hardware Performance Counters," Department of Defense Users' Group Conference Proceedings, Biloxi, Mississippi, June 18-21, 2001. http://icl.cs.utk.edu/papi/documens

London, K., Dongarra, J., Moore, S., Mucci, P., Seymour, K., and Spencer, T. "End-user Tools for Application Performance Analysis Using Hardware Counters," *International Conference on Parallel and Distributed Computing Systems*, Dallas, TX, August 8-10, 2001. http://icl.cs.utk.edu/papi/documents

Mucci, P., Moore, S., and Smeds, Nils. "Performance Tuning Using Hardware Counter Data," Proc. *SC'2001,* November 2001. http://icl.cs.utk.edu/papi/documents

Mucci, P. "The IA64 Hardware Performance Monitor and PAPI", *The 2001 International Conference on Parallel and Distributed Processing Techniques and Applications,* June 2001. http://icl.cs.utk.edu/papi/documents

Mucci, P. "PAPI -- The Performance Application Programming Interface," April 2000. http://icl.cs.utk.edu/papi/documents

"POSIX Threads Programming." http://www.llnl.gov/computing/tutorials/workshops/workshop/pthreads/MAIN.html