

PAPI Conversion Cookbook

Introduction

So... you've instrumented your code and it's running fine under PAPI 2.x.y. Why would you want to upgrade to PAPI 3.0? There are some good reasons. 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.

With all this good stuff, where's the downside? Well... there are a few things you have to give up:

- **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.** That's right. Code rewritten to link to the PAPI 3 library won't any longer link to the PAPI 2 library. You've gotta break a few eggs to make an omelet. If you're a tool builder for whom version independence is an important issue, we've created a PAPIvi layer for version independence. More on that later. 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.

Enough marketing hype. Let's get to work!

Code Conversion

Below is a cookbook that should bring your code from PAPI 2 compatibility to PAPI 3 compatibility. Don't be alarmed; it's long, but very detailed. Follow along...

- 1) If you only use PAPI High Level calls, you're done. There are two new functions in the High Level API, but the functions from PAPI 2 are called identically in PAPI 3. Proceed only if you use the PAPI Low Level API.
- 2) Make a backup. We shouldn't have to explain why.
- 3) Modify existing functions that have changed in some way. There are 17 affected functions with 5 basic types of changes that will be discussed below.

a) Function name changes.

- i) Change all occurrences of 'PAPI_is_initialized' to 'PAPI_initialized'.
- ii) Change all 'PAPI_rem_event' to 'PAPI_remove_event'.
- iii) This should also have changed all 'PAPI_rem_events' to 'PAPI_remove_events'.

b) Identifier name changes.

Change the identifier names for PAPI_set_opt() and PAPI_get_opt().

NOTE: If you plan on converting to PAPIvi version independent code as described at the end of this document, you can skip this step.

- i) PAPI_SET_DEBUG and PAPI_GET_DEBUG → PAPI_DEBUG
- ii) PAPI_SET_MULTIPLEX and PAPI_GET_MULTIPLEX → PAPI_MULTIPLEX
- iii) PAPI_SET_DEFDOM and PAPI_GET_DEFDOM → PAPI_DEFDOM
- iv) PAPI_SET_DOMAIN and PAPI_GET_DOMAIN → PAPI_DOMAIN
- v) PAPI_SET_DEFGRN and PAPI_GET_DEFGRN → PAPI_DEFGRN
- vi) PAPI_SET_GRANUL and PAPI_GET_GRANUL → PAPI_GRANUL
- vii) PAPI_SET_INHERIT and PAPI_GET_INHERIT → PAPI_INHERIT
- viii) PAPI_GET_NUMCTRS and PAPI_SET_NUMCTRS → PAPI_NUMCTRS
- ix) PAPI_SET_PROFIL and PAPI_GET_PROFIL → PAPI_PROFIL

c) Dereferencing changes.

For the following seven functions, change the first parameter from &EventSet to EventSet. This extra layer of dereferencing is unneeded and slows things down.

- i) PAPI_add_event(&EventSet, Event) → PAPI_add_event(EventSet, Event)
- ii) PAPI_add_events(&EventSet, Event) → PAPI_add_events(EventSet, Event)
- iii) PAPI_cleanup_event(&EventSet, Event) → PAPI_cleanup_event(EventSet, Event)
- iv) PAPI_cleanup_eventset(&EventSet) → PAPI_cleanup_eventset(EventSet)

- v) PAPI_remove_event(&EventSet, Event) →
PAPI_remove_event(EventSet, Event)
 - vi) PAPI_remove_events(&EventSet, Event) →
PAPI_remove_events(EventSet, Event)
 - vii) PAPI_set_multiplex(&EventSet, Event) →
PAPI_set_multiplex (EventSet, Event)
- d) Parameter changes.
- Six functions have new parameters added, parameter type changes, or a parameter removed from the calling sequence:
- i) PAPI_thread_init() took two parameters in PAPI 2. The second parameter was unused and always zero. Get rid of it:
(1) PAPI_thread_init(id_fn, 0); → PAPI_thread_init(id_fn);
 - ii) PAPI_locks now support two user specified levels that don't conflict with PAPI system locks, PAPI_USR1_LOCK and PAPI_USR2_LOCK. As a first cut, add the parameter PAPI_USR1_LOCK to each call:
(1) PAPI_lock() → PAPI_lock(PAPI_USR1_LOCK)
(2) PAPI_unlock() → PAPI_unlock(PAPI_USR1_LOCK)
 - iii) PAPI_overflow() has been restructured to streamline overhead and support multiple overflowing events. Specifically, the prototype for the last parameter in the call -- the overflow handler -- has changed. In addition to changing this parameter in calls to PAPI_overflow(), you will need to rewrite your overflow handler routine itself. Examples of roughly equivalent routines can be seen in any of the overflow test cases in the /ctests directory. One such example is shown below:

PAPI 2:

```
int total = 0; /* total overflows */
void handler(int EventSet, int EventCode, int EventIndex,
             long_long *values, int *threshold, void *context)
{
    printf("handler(%d, %x, %d, %lld, %d, %p) Overflow at %p!\n",
          EventSet, EventCode, EventIndex, values[EventIndex],
          *threshold, context, PAPI_get_overflow_address(context));
    total++;
}
```

PAPI 3:

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

- iv) PAPI_profil() and PAPI_sprofil() now support multiple bin sizes for the profile buffer. The PAPI 2 and default PAPI 3 size is 16 bits (unsigned short).

To accommodate this change, you may need to cast the first parameter in the PAPI_profil() call, and/or the first field in the PAPI_sprofil_t structure, to 'void *':

- (1) PAPI_profil(unsigned short *buf, ...) →
PAPI_profil(void *buf, ...)
- (2) (PAPI_sprofil_t *)prof->pr_base = (unsigned short *)base; →
(PAPI_sprofil_t *)prof->pr_base = (void *)base;

e) Data Structure changes.

One function retains the same calling sequence, but returns a modified data structure, as discussed below:

- i) const PAPI_exe_info_t *PAPI_get_executable_info(void);
Fields in the PAPI_exe_info_t structure are now shielded by two new structures that require a modification of referencing syntax.
For every occurrence of the data structure PAPI_exe_info_t, modify existing references to these fields as follows:
 - (1) exe_info.text_start → exe_info.address_info.text_start
 - (2) exe_info.text_end → exe_info.address_info.text_end
 - (3) exe_info.data_start → exe_info.address_info.data_start
 - (4) exe_info.data_end → exe_info.address_info.data_end
 - (5) exe_info.bss_start → exe_info.address_info.bss_start
 - (6) exe_info.bss_end → exe_info.address_info.bss_end
 - (7) exe_info.lib_preload_env → exe_info.preload_info.lib_preload_env

4) Remove references to deprecated functions.

There are eight functions in the PAPI 2 API that were deprecated for PAPI 3. They fall roughly into four categories, as described below:

a) Unimplemented functions.

Three functions from PAPI 2 were never implemented and returned errors. If your code contains references to them, they will cause a link error and should be deleted:

- i) int PAPI_add_pevent(int *EventSet, int code, void *inout);
- ii) int PAPI_restore(void);
- iii) int PAPI_save(void);

b) Overflow address function.

Overflow handlers in PAPI 2 needed to compute the overflow address by calling PAPI_get_overflow_address(). PAPI 3 passes this address into the overflow handler as one of the parameters. Thus PAPI_get_overflow_address() is deprecated and should disappear from your code as you rewrite your overflow handler routines as outlined in section 3) c) iii).

c) Hardware profiling function.

In the unlikely event that your code calls PAPI_profil_hw(), replace that call with a call to PAPI_profil() as discussed in step 3).c).ii) above. This call was an experimental API, the function of which was folded into the standard PAPI_profil() call on those platforms that support hardware profiling.

d) Event description functions.

PAPI 2 implemented a collection of functions that allowed you to describe the status and attributes of preset events in the PAPI interface. The PAPI event description interface in PAPI 3 has been modified and extended to include the description of events native to a particular hardware substrate as well.

Specifically, four functions in the PAPI 2 interface have been deprecated and replaced by two functions in PAPI 3. Thus, more than just one-to-one coding changes are required to replace the deprecated functions. We discuss below possible code changes necessary to replace each of these deprecated functions:

i) `const PAPI_preset_info_t *PAPI_query_all_events_verbose(void);`

This function returned a pointer to a constant array of structures that described all of the preset events. Since the same interface now handles both preset and native events, this function is no longer supported. Instead, each preset is queried in turn, using a 'for' or 'while' loop. Compare the code in `ctests/avail.c` for PAPI 2 and PAPI 3, for an illustration of the changes. The example shown below provides further details:

PAPI 2:

```
/* retrieve the array of preset_info structures */
const PAPI_preset_info_t *info = PAPI_query_all_events_verbose();
if (info == NULL)
    handle_error();

/* print the names of all named preset events */
for (i=0; i<PAPI_MAX_PRESET_EVENTS; i++) {
    if (info[i].event_name)
        printf("Event Name [%d]: %s\n", i, info[i].event_name);
}
```

PAPI 3:

```
int i = PRESET_MASK; /* look for the first PRESET event */
int avail = TRUE; /* only return events available on this platform */
PAPI_event_info_t info;

do {
    /* return information for the requested event code */
    if (PAPI_get_event_info(i, &info) == PAPI_OK) {
        /* print the name of this preset event */
        printf("Event Name [%d]: %s\n", i, info.symbol);
    }
    /* update i with the next available event code */
} while (PAPI_enum_event(&i, avail) == PAPI_OK);
```

ii) `int PAPI_query_event_verbose(int EventCode, PAPI_preset_info_t *info);`

This function filled a structure for a single preset event. The same functionality is now found in `PAPI_get_event_info()`, as shown in the example above. Note that the `PAPI_preset_info_t` structure has been changed

to the more generic PAPI_event_info_t structure. When mapping the former to the latter, the following rules apply:

- (1) preset_info.event_name → event_info.symbol
- (2) preset_info.event_code → event_info.event_code
- (3) preset_info.event_descr → event_info.long_descr
- (4) preset_info.event_label → event_info.short_descr
- (5) (preset_info.avail == TRUE) → (event_info.count > 0)
- (6) preset_info.event_note → event_info.vendor_name
- (7) (preset_info.flags & PAPI_DERIVED) → (event_info.count > 1)

iii) int PAPI_describe_event(char *name, int *EventCode, char *description);

This PAPI 2 convenience function, when given either the name or event code of a PAPI preset event, returned the other value and the description of the event. For PAPI 3, two situations could occur:

- (1) if given an event code, simply call PAPI_get_event_info() and examine the long_descr field of the returned structure.
- (2) If given an event name, call PAPI_event_name_to_code() to convert the name to an event code. Follow this with a call to PAPI_get_event_info() to retrieve the description.

iv) int PAPI_label_event(int EventCode, char *label);

Replace calls to this function with a call to PAPI_get_event_info() followed by an examination of the short_descr field of the returned structure.

- 5) If you use native events, you should isolate and rethink the native event interface. Native event support in PAPI 3 is name-based rather than opcode based. It also supports the full range of calls associated with PAPI preset events. It is necessary to redesign this portion of your code to take full advantage of PAPI 3.
- 6) Compile and link against PAPI 3.
- 7) Identify and remove any use of overlapping eventsets. This should become readily apparent from runtime error messages.
- 8) Resolve any remaining issues.
- 9) Report to us anything we forgot to discuss, so we can improve this document for future users.

PAPI Version Independence

Once you've converted your code to PAPI 3, you need never look back, right?

Well, usually.

If you're a tool builder who must supply your tool to run on systems supporting either PAPI 2 or PAPI 3, or if you're running your code in mixed environments that haven't all been upgraded to PAPI 3, you may need source code that can compile and link to either the PAPI 2 or PAPI 3 library.

For that reason, and in response to requests from our users, we've developed the **PAPIvi** header. By including the `papivi.h` header instead of the `papi.h` file in your application code, and making some straightforward syntax changes, you can generally get your PAPI 3 code to compile and link to either PAPI 2 or PAPI 3.

A few caveats:

- We won't guarantee that all features of PAPI 3 will work with PAPI 2.
- We won't guarantee that you won't need any `#ifdefs` in your code.
- We won't guarantee that this syntax will provide version independence for future versions of PAPI.
- We will assert that using PAPIvi will get you *closer* to those goals, and save you some work in providing PAPI version independence to your users.

As proof of principle and to provide example code, we have converted many of the test cases from the `/ctests` directory of the PAPI distribution into PAPIvi compatible code. You can examine these files, found in the `/victests` directory, to better understand the conversion steps below. You can also make a backup copy of the regular `/ctests` directory and replace it with a renamed copy of the `/victests` directory and recompile PAPI normally to see PAPIvi in action.

Using PAPIvi

If, after reading the above, you want to use PAPIvi, follow the steps below to make your code PAPIvi compatible:

- 1) Make a copy of your PAPI 3 code.
- 2) Replace all references to `papi.h` with `papivi.h`
 - a) `#include "papi.h" → #include "papivi.h"`
- 3) Rename all references to `PAPI_xxx()` function calls to `PAPIvi_xxx()`.
 - a) e.g. `PAPI_library_init → PAPIvi_library_init`
 - b) DO NOT rename macros, identifiers and structures.
 - i) `PAPI_VER_CURRENT → PAPI_VER_CURRENT`
- 4) Rename references to two structures:
 - a) `PAPI_hw_info_t → PAPIvi_hw_info_t`
 - b) `PAPI_exe_info_t → PAPIvi_exe_info_t`
- 5) If you use any PAPI 3 identifiers in calls to `PAPI_set_opt()` or `PAPI_get_opt()`, replace them with the appropriate PAPI 2 identifiers:
 - a) `PAPI_DEBUG → PAPI_SET_DEBUG` or `PAPI_GET_DEBUG`

- b) PAPI_MULTIPLEX → PAPI_SET_MULTIPLEX or PAPI_GET_MULTIPLEX
- c) PAPI_DEFDOM → PAPI_SET_DEFDOM or PAPI_GET_DEFDOM
- d) PAPI_DOMAIN → PAPI_SET_DOMAIN or PAPI_GET_DOMAIN
- e) PAPI_DEFGRN → PAPI_SET_DEFGRN or PAPI_GET_DEFGRN
- f) PAPI_GRANUL → PAPI_SET_GRANUL or PAPI_GET_GRANUL
- g) PAPI_INHERIT → PAPI_SET_INHERIT or PAPI_GET_INHERIT
- h) PAPI_NUMCTRS → PAPI_GET_NUMCTRS or PAPI_SET_NUMCTRS
- i) PAPI_PROFIL → PAPI_SET_PROFIL or PAPI_GET_PROFIL

- 6) If you use any of the following five PAPI 3 calls, they will need to be conditionally compiled for PAPI 3 only. This can be done as shown below:

```
#ifndef PAPI_VERSION
    /* PAPI 2 support goes here */
#elif (PAPI_VERSION_MAJOR(PAPI_VERSION) == 3)
    /* PAPI 3 support goes here */
#else
    /* PAPI future support goes here */
#error Compiling against a not yet released PAPI version
#endif
```

Calls supported only in PAPI 3:

- a) PAPIvi_get_shared_lib_info()
- b) PAPIvi_get_thr_specific()
- c) PAPIvi_num_events()
- d) PAPIvi_register_thread()
- e) PAPIvi_set_thr_specific()

- 7) This step only applies if you call PAPI_overflow(). The overflow handler routines for PAPI 2 and PAPI 3 are significantly different. If you have converted your code from a PAPI 2 implementation, retrieve the overflow handler code from your PAPI 2 backup and insert it into your PAPIvi code, wrapping it and the new PAPI 3 overflow handler with the conditional compilation macros shown in step 6 above. This will provide the proper overflow handler for whichever version of the PAPI library to which you link.
- 8) If you use native events in your application code, you must recognize that PAPI 2 and PAPI 3 provide very different and incompatible interfaces to native events. These differences are NOT covered by the PAPIvi header. You should isolate your native event code, and provide conditionally compiled versions for PAPI 2 and PAPI 3 as shown in step 6 above.
- 9) Compile and link against either PAPI library.
- 10) Resolve any remaining issues.
- 11) Report to us anything we forgot to discuss, so we can improve this document for future users.