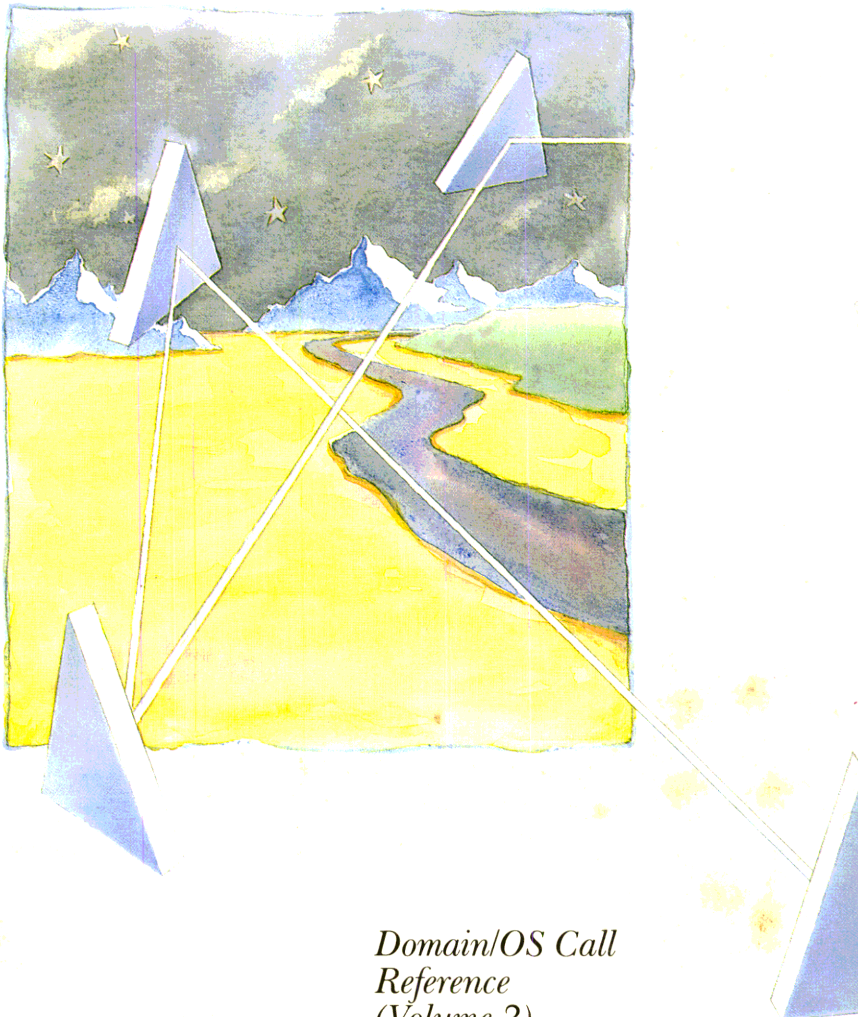


Domain/OS Call Reference (Volume 2)

012888-A00

apollo

Domain/OS Call Reference
Volume 2

Order No. 012888-A00

Apollo Computer Inc.
330 Billerica Road
Chelmsford, MA 01824

First Printing: June 1988

This manual is dedicated to the memory of Kriss Kellerman,
a valued friend and colleague.

Preface

The *Domain/OS Call Reference* describes unique programming interfaces to Domain®/OS available only on a Domain system. The interfaces documented here can be called from all three Domain/OS environments, and, except where noted, can be used freely together with UNIX* system and library calls. For instance, the stream IDs returned by the Domain/OS call **ios_$open** are identical to the file descriptors expected by the UNIX system calls **read** or **write**, and can even be passed to **fdopen** to obtain the file pointers used by the standard I/O library.

The declarations of Domain/OS interfaces are grouped by function into insert files. Each call and each predefined data type, variable, or constant has a prefix that generally identifies the insert file in which it's declared. Interfaces prefixed with **pad_$**, for example, allow programs to manipulated pads and windows on the display via the Display Manager. They are declared in the following insert files:

/usr/include/apollo/pad.h	for C
/sys/ins/pad.ins.pas	for Pascal
/sys/ins/pad.ins.ftn	for FORTRAN

Some extremely common data types and constants are defined in the "base" insert file for each language. The base insert files are

* UNIX is a registered trademark of AT&T in the USA and other countries.

/usr/include/apollo/base.h	for C
/sys/ins/base.ins.pas	for Pascal
/sys/ins/base.ins.ftn	for FORTRAN

For example, the Domain/OS completion status type, **status_$t**, the system clock value type, **time_$clock_t**, and the stream ID type, **ios_$id_t**, as well as the values for the standard streams, **ios_$stdin**, **ios_$stdout**, and **ios_$stderr**, are all defined in the base insert file.

The *Domain/OS Call Reference* is divided into sections reflecting the grouping of interface declarations into insert files. Information about an interface can usually be found in the section whose name is the prefix of that interface. The sections are ordered alphabetically by prefix.

Each section begins with an introduction to the interfaces it documents. The introduction contains a brief discussion of the function of the interfaces, and explains any predefined constants and data types. The introduction is followed by an individual description of each call ordered alphabetically by the name of the call.

Each section is preceded by a ''Chapter Table of Contents'' on colored stock to speed location of individual call descriptions.

Call Descriptions

All call descriptions and introductions use a common format consisting of the following parts:

- The **NAME** part contains the name of the Domain/OS call and a brief description of its purpose. The introductory sections all use the name ''intro''.

- The **SYNOPSIS** parts list the include files recommended for the call, followed by a declaration for the call or an example of its use. The **SYNOPSIS** parts of an introduction just show the include files that define the interface. Nearly all call descriptions contain three **SYNOPSIS** sections — one each for C, Pascal, and FORTRAN. C and Pascal synopses contain declarations of the call and its arguments in the appropriate language.

  Because FORTRAN lacks user-defined data types, a FORTRAN synopsis does not attempt to present a canonical declaration for the call it describes. Instead, a FORTRAN synopsis contains an example of declaring the necessary arguments and making the call in FORTRAN, and, by necessity, sometimes contains extra variables with arbitrary values chosen only for the example. Note that a FORTRAN **SYNOPSIS** is just one example of how to make the call in FORTRAN, and cannot be the ''best'' example for every application.

- The **DESCRIPTION** part describes the purpose of the call, the use of its arguments, and the value it returns, if any.
- The **EXAMPLES** part contains examples of using the call.
- The **FILE** part contains the pathnames of objects used by the call.
- The **NOTES** part contains additional helpful information that is not exclusively descriptive of the call itself. **NOTES** may explain how a closely related call differs, or point out problems with using the call in combination with others.
- The **SEE ALSO** part contains references to related calls and introductions.

Online Access

All of the information contained in this document is available online via the Aegis **help** command and is integrated into the UNIX online manual accessible via **man**.

In the Aegis environment, you can obtain information on a call by typing:

$ help calls *name_of_call*

where *name_of_call* is the name of the call exactly as it should appear in a program. To get information on **ios_$open**, for instance, type

$ help calls ios_$open

You can read the appropriate introduction two ways, by either typing only the prefix, or by typing *prefix*_**$intro**. For instance, either of the two following command lines will get the introduction to the **ios_$** calls:

$ help calls ios

$ help calls ios_$intro

In a UNIX environment, you can obtain information on a call by typing:

$ man a *name_of_call*

where *name_of_call* is the name of the Domain/OS call with the dollar sign ($) removed from the prefix. (The dollar sign was omitted so arguments to **man** do not have to be quoted to avoid variable substitution in UNIX shells.) To get information on **ios_$open**, for instance, type

$ man a ios_open

You can read the appropriate introduction two ways, by either typing only the prefix, or by typing *prefix*_**intro**. For instance, either of the two following command lines will get the introduction to the **ios_$** calls:

$ man a ios

$ man a ios_intro

Using an "a" for the section specifier to **man** restricts the manual search to Domain/OS call documentation and may be omitted if such a restriction is not necessary.

Related Manuals

The file **/install/doc/apollo/os.v.10.***n*__**manuals** lists current titles and revisions for all manuals available at update *n* of SR10. For example, at SR10.0 you can check **/install/doc/apollo/os.v.10.***n*__**manuals** to make sure you have the latest version of the manuals you use, or to find additional manuals you might need. (In the Aegis environment, you can access the same information by typing "**help manuals**".)

Refer to the *Domain Documentation Quick Reference* (002685) and the *Domain Documentation Master Index* (011242) for a complete list of related documents. For more information on programming in Domain/OS, refer to the following documents:

- *SysV Programmer's Reference* (005799)
- *BSD Programmer's Reference* (005801)
- *Programming with Domain/OS Calls* (005506)
- *Programming with Domain/OS Calls for IPC* (005696)
- *Domain/OS Programming Environment Reference* (011010)
- *Domain Distributed Debugging Environment Reference* (011024)
- *Domain Binder and Librarian Reference* (004977)
- *Domain C Language Reference* (002093)
- *Domain Pascal Language Reference* (000792)
- *Domain FORTRAN Language Reference* (000530)

Problems, Questions, and Suggestions

We appreciate comments from the people who use our system. To make it easy for you to communicate with us, we provide the Apollo® Product Reporting (APR) system for comments related to hardware, software, and documentation. By using this formal channel you make it easy for us to respond to your comments.

You can get more information about how to submit an APR by consulting the appropriate Command Reference manual for your environment (Aegis™, BSD, or SysV). Refer to the **mkapr** command description. You can view the same description online by typing:

$ help mkapr (in the Aegis environment)

$ man 1 mkapr (in a UNIX environment)

Alternatively, you may use the Reader's Response form at the back of this manual to submit comments about the manual.

Documentation Conventions

This manual uses the following symbolic conventions:

literal symbols	Keywords and predefined symbols are printed in bold type. They must be used exactly as documented.
variable symbols	Placeholders for symbols that can be chosen by the programmer are printed in italics. For example, the names chosen for call arguments are printed in italics.
`examples`	Examples of program code appears in a constant width font to preserve indentation.
|	A vertical bar separates items in a list of choices.

Contents

aclm

cal

ctm

ec2

error

fault

fpp

gmf

ios

ios_dir

ipc

lib

loader

mbx

ms

mts

mutex

name

osinfo

pad

pbufs

pfm

pgm

pm

prf

proc1

proc2

rws

sio

status

task

time

tone

tpad

vec

vfmt

Permuted Index

pfm

Contents

NAME

intro – managing faults

SYNOPSIS (C)

#include <apollo/base.h>
#include <apollo/pfm.h>

SYNOPSIS (Pascal)

%include '/sys/ins/base.ins.pas';
%include '/sys/ins/pfm.ins.pas';

SYNOPSIS (FORTRAN)

%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/pfm.ins.ftn'

DESCRIPTION

The **pfm_$** calls allow programs to manage signals, faults, and exceptions. Programs can manage faults by establishing clean-up handlers and fault handlers.

Clean-up Handlers

A clean-up handler is a piece of code that ensures a program terminates gracefully when it receives a fatal error. A clean-up handler begins with a **pfm_$cleanup** call, and usually ends with a call to **pfm_$signal** or **pgm_$exit**, though it can also simply continue back into the program after the clean-up code.

A clean-up handler is not entered until all fault handlers established for a fault have returned. If there is more than one established clean-up handler for a program, the most recently established clean-up handler is entered first, followed by the next most recently established clean-up handler, and so on to the first established clean-up handler if necessary.

There is a default clean-up handler invoked after all user-defined handlers have completed. It releases any resources still held by the program, before returning control to the process that invoked it.

Fault Handlers

A fault handler is a function supplied by the user that is called when a fault occurs. When a fault occurs that has a handler established for it, the system interrupts the faulted program and calls the designated fault handler. When the fault handler returns, processing resumes where the program was interrupted.

There are three types of fault handlers. By default a handler is called only when a signal is received at the program level at which it was established. A ``multilevel'' fault handler is called whenever a signal is received at its or any subordinate program level. A ``backstop'' fault handler is called only after all other non-backstop handers have returned.

All fault handlers must take one argument of type **pfm_$fault_rec_t** and return a value of type **pfm_$fh_func_val_t**, but their implementation is otherwise left to the user. If there is more than one established handler for a fault, the most recent nonbackstop handler is called first, followed by the next most recent, if necessary, and so on until all the applicable non-backstop handlers have returned. Then any backstop handlers are called, if necessary, in the same last-established/first-called order.

Constants

pfm_$all_faults
Passed to **pfm_$establish_fault_hander** to esablish a handler for all possible faults.

Data Types

pfm_$cleanup_rec
A record type for passing process context among clean-up handler calls. It is an opaque data type.

pfm_$fault_func_p_t
A pointer to a fault handler function.

pfm_$fault_rec_t
A record type for passing status information among fault handlers. Only the **status** field is accessible to fault handler functions. The following diagram illustrates the data type:

15 8	7 0
Not Used	pattern
status	
15	0

pattern Reserved for system use.

status A fault status of type **status_$t**.

pfm_$fh_func_val_t
A value returned by a fault handler function. It specifies the action to be taken when the handler completes. It can be one of the following values:

pfm_$continue_fault_handling
Pass the fault to the next handler.

pfm_$return_to_faulting_code
Return control to the faulted program.

pfm_$fh_handle_t
A pointer to a fault handler function.

pfm_$fh_opt_set_t
A small set of options when establishing a fault handler. It can be any combination of the following values:

pfm_$fh_multi_level
Establish a ''multilevel'' handler for faults encountered at the current and all subordinate program levels.

pfm_$fh_backstop
Establish a ''backstop'' handler that is called after all nonbackstop handlers have been called.

By default, fault handlers are called consecutively on a last-in/first-out basis, starting with the most recently established handler and ending with the first fault handler established in the current program level. To establish a default handler in Pascal, pass an empty set [] for *fh_options*; in C, pass 0.

Errors

pfm_$bad_rls_order
Attempted to release a clean-up handler out of order.

pfm_$cleanup_not_found
There is no pending clean-up handler.

pfm_$cleanup_set
A clean-up handler was established successfully.

pfm_$cleanup_set_signalled
Attempted to use **pfm_$cleanup_set** as a signal.

pfm_$fh_not_found
Attempted to release non-existent fault handler.

pfm_$fh_wrong_level
Attempted to release fault handler at wrong level.

pfm_$invalid_cleanup_rec
Passed an invalid clean-up record to a call.

pfm_$no_space
Cannot allocate storage for a clean-up handler.

SEE ALSO
fault_$intro.

NAME

pfm_$cleanup – establish a clean-up handler

SYNOPSIS (C)

#include <apollo/base.h>
#include <apollo/pfm.h>

status_$t pfm_$cleanup(pfm_$cleanup_rec **cleanup_record***)**

SYNOPSIS (Pascal)

%include '/sys/ins/base.ins.pas';
%include '/sys/ins/pfm.ins.pas';

function pfm_$cleanup(
out *cleanup_record*: **pfm_$cleanup_rec): status_$t;**

SYNOPSIS (FORTRAN)

%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/pfm.ins.ftn'

integer*4 *status, cleanup_record***(16)**

status = **pfm_$cleanup(***cleanup_record***)**

DESCRIPTION

Pfm_$cleanup establishes a clean-up handler that is executed when a fault occurs. A clean-up handler is a piece of code executed before a program exits when a signal is received by the process. The clean-up handler begins where **pfm_$cleanup** is called; the **pfm_$cleanup** call registers an entry point with the system where program execution resumes when a fault occurs. When a fault occurs, execution resumes after the most recent call to **pfm_$cleanup**.

There can be more than one clean-up handler in a program. Multiple clean-up handlers are executed consecutively on a last-in/first-out basis, starting with the most recently established handler and ending with the first clean-up handler. The system provides a default clean-up handler esablished at program invocation. The default clean-up handler is always called last, just before a program exits, and releases any system resources still held, before returning control to the process that invoked the program.

When called to establish a clean-up handler, **pfm_$cleanup** returns the status **pfm_$cleanup_set** to indicate the clean-up handler was successfully established. When the clean-up handler is entered in response to a fault signal, **pfm_$cleanup** effectively returns the value of the fault that triggered the handler.

cleanup_record

A record of the context when **pfm_$cleanup** is called. A program should treat this as an opaque data structure, and not try to alter or copy its contents. It is needed by **pfm_$rls_cleanup** and **pfm_$reset_cleanup** to restore the context of the calling process at the clean-up handler entry point.

NOTES

Clean-up handler code runs with asynchronous faults inhibited. When **pfm_$cleanup** returns something other than **pfm_$cleanup_set**, indicating that a fault has occurred, there are four possible ways to leave the clean-up code:

- The program can call **pfm_$signal** to start the next clean-up handler with a different fault signal.
- The program can call **pgm_$exit** to start the next clean-up handler with the same fault signal.
- The program can continue with the code following the clean-up handler. It should generally call **pfm_$enable** to re-enable asynchronous faults. Execution continues from the end of the clean-up handler code; it does not resume where the fault signal was received.
- The program can re-establish the handler by calling **pfm_$reset_cleanup** before proceeding.

SEE ALSO

fault_$intro, pfm_$error_trap, pfm_$signal.

NAME

pfm_$enable – enable asynchronous faults

SYNOPSIS (C)

#include <apollo/base.h>
#include <apollo/pfm.h>

void pfm_$enable(void)

SYNOPSIS (Pascal)

%include '/sys/ins/base.ins.pas';
%include '/sys/ins/pfm.ins.pas';

procedure pfm_$enable;

SYNOPSIS (FORTRAN)

%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/pfm.ins.ftn'

call pfm_$enable

DESCRIPTION

Pfm_$enable enables asynchronous faults after they have been inhibited by a call to **pfm_$inhibit**. **Pfm_$enable** causes the operating system to pass asynchronous faults on to the calling process.

While faults are inhibited, the operating system holds at most one asynchronous fault. Consequently, when **pfm_$enable** returns, there can be at most one fault waiting on the process. If more than one fault was received between calls to **pfm_$disable** and **pfm_$enable**, the process receives the first asynchronous fault received while faults were inhibited.

NAME

pfm_$error_trap – simulate a fault and save a traceback

SYNOPSIS (C)

#include <apollo/base.h>
#include <apollo/pfm.h>

void pfm_$error_trap(status_$t &*fault_signal*)

SYNOPSIS (Pascal)

%include '/sys/ins/base.ins.pas';
%include '/sys/ins/pfm.ins.pas';

procedure pfm_$error_trap(in *fault_signal*: status_$t);

SYNOPSIS (FORTRAN)

%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/pfm.ins.ftn'

integer*4 *fault_signal*

call pfm_$error_trap(*fault_signal*)

DESCRIPTION

Pfm_$error_trap signals the fault specified by *fault_signal* to the calling process, and saves a traceback of the calling sequence. It is especially useful when debugging a program, or in writing debuggers and monitoring utilities.

fault_signal
A fault code.

NOTES

Pfm_$error_trap does not return when successful.

SEE ALSO

fault_$intro, pfm_$signal.

NAME

pfm_$establish_fault_handler – establish a fault handler

SYNOPSIS (C)

#include <apollo/base.h>
#include <apollo/pfm.h>

pfm_$fh_handle_t pfm_$establish_fault_handler(
int &*target_status*,
pfm_$fh_opt_set_t &*fh_options*,
pfm_$fault_func_p_t &*function_ptr*,
status_$t **status*)

SYNOPSIS (Pascal)

%include '/sys/ins/base.ins.pas';
%include '/sys/ins/pfm.ins.pas';

function pfm_$establish_fault_handler(
in *target_status*: **integer32;**
in *fh_options*: **pfm_$fh_opt_set_t;**
in *function_ptr*: **pfm_$fault_func_p_t;**
out *status*: **status_$t): pfm_$fh_handle_t;**

DESCRIPTION

Pfm_$establish_fault_handler establishes the function at *function_ptr* as a handler for the faults specified by *target_status*, and returns a handle for it.

target_status

A value specifying the type of fault the function at *function_ptr* should handle. Set *target_status* to **pfm_$all_faults** to establish a handler for all possible faults. To establish a handler for all faults produced by a set of Domain/OS calls, use one of that manager's predefined completion status values with the **code** field set to 0.

fh_options

. This is a small set of options that limit the scope of the fault handler. Specify any combination of the following values:

pfm_$fh_multi_level

Establish a "multilevel" handler for faults encountered at the current and all subordinate program levels.

pfm_$fh_backstop

Establish a "backstop" handler that is called after all nonbackstop handlers have been called.

By default, fault handlers are called consecutively on a last-in/first-out basis, starting with the most recently established handler and ending with the first fault handler established in the current program level. To establish a default handler in Pascal, pass an empty set [] for *fh_options*; in C, pass 0.

function_ptr
The address of a function to establish as a fault handler.

status The completion status.

NOTES

A fault handler remains in effect until released by passing the returned handle to **pfm_$release_fault_handler**.

NAME

pfm_$inhibit – inhibit asynchronous faults

SYNOPSIS (C)

#include <apollo/base.h>
#include <apollo/pfm.h>

void pfm_$inhibit(void);

SYNOPSIS (Pascal)

%include '/sys/ins/base.ins.pas';
%include '/sys/ins/pfm.ins.pas';

procedure pfm_$inhibit;

SYNOPSIS (FORTRAN)

%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/pfm.ins.ftn'

call pfm_$inhibit

DESCRIPTION

Pfm_$inhibit prevents asynchronous faults from being passed to the calling process. While faults are inhibited, the operating system holds at most one asynchronous fault. Consequently, a call to **pfm_$inhibit** can result in the loss of some signals. For that and other reasons, it is good practice to inhibit faults only when absolutely necessary.

NOTES

Pfm_$inhibit has no effect on the processing of synchronous faults such as floating-point and overflow exceptions, access violations, and so on.

SEE ALSO

pfm_$enable.

NAME

pfm_$release_fault_handler – release a fault handler

SYNOPSIS (C)

```
#include <apollo/base.h>
#include <apollo/pfm.h>

void pfm_$release_fault_handler(
        pfm_$fh_handle_t &fh_handle,
        status_$t *status)
```

SYNOPSIS (Pascal)

```
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/pfm.ins.pas';

procedure pfm_$release_fault_handler(
        in fh_handle: pfm_$fh_handle_t;
        out status: status_$t);
```

SYNOPSIS (FORTRAN)

```
%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/pfm.ins.ftn'

        integer*4 fh_handle, status

        call pfm_$release_fault_handler(fh_handle, status)
```

DESCRIPTION

Pfm_$release_fault_handler releases the fault handler on *fh_handle*. After **pfm_$release_fault_handler** returns, the function acting as the fault handler will no longer be called when the process receives the fault signal it was established to handle.

fh_handle
A handle on the fault handler to be released.

status The completion status.

NOTES

Releasing a fault handler established inside another fault handler may cause unpredictable results.

NAME

pfm_$reset_cleanup – reset a clean-up handler

SYNOPSIS (C)

```
#include &lt;apollo/base.h&gt;
#include &lt;apollo/pfm.h&gt;

void pfm_$reset_cleanup(
        pfm_$cleanup_rec &cleanup_record,
        status_$t *status)
```

SYNOPSIS (Pascal)

```
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/pfm.ins.pas';

procedure pfm_$reset_cleanup(
        in cleanup_record: pfm_$cleanup_rec;
        out status: status_$t);
```

SYNOPSIS (FORTRAN)

```
%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/pfm.ins.ftn'

        integer*4 cleanup_record(16), status

        call pfm_$reset_cleanup(cleanup_record, status)
```

DESCRIPTION

Pfm_$reset_cleanup re-establishes the clean-up handler last entered so that any subsequent errors enter it first. This procedure should only be used within clean-up handler code.

cleanup_record
: A record of the context at the clean-up handler entry point. It is supplied by **pfm_$cleanup**, when the clean-up handler if first established.

status
: The completion status.

NAME

pfm_$rls_cleanup – release clean-up handlers stack.

SYNOPSIS (C)

```
#include <apollo/base.h>
#include <apollo/pfm.h>

void pfm_$rls_cleanup(
        pfm_$cleanup_rec &cleanup_record,
        status_$t *status)
```

SYNOPSIS (Pascal)

```
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/pfm.ins.pas';

procedure pfm_$rls_cleanup(
        in cleanup_record: pfm_$cleanup_rec;
        out status: status_$t);
```

SYNOPSIS (FORTRAN)

```
%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/pfm.ins.ftn'

        integer*4 cleanup_record(16), status

        call pfm_$rls_cleanup(cleanup_record, status)
```

DESCRIPTION

Pfm_$rls_cleanup releases the clean-up handler associated with *cleanup_record*, and all clean-up handlers established after it.

cleanup_record
The clean-up record for the first clean-up handler to release.

status The completion status. If *status* is **pfm_$bad_rls_order**, it means that the caller attempted to release a clean-up handler before releasing all handlers established after it. This status is only a warning; the intended clean-up handler is released, along with all clean-up handlers established after it.

NAME

pfm_$signal – signal the calling process

SYNOPSIS (C)

#include <apollo/base.h>
#include <apollo/pfm.h>

void pfm_$signal(status_$t &*fault_signal*)

SYNOPSIS (Pascal)

%include '/sys/ins/base.ins.pas';
%include '/sys/ins/pfm.ins.pas';

procedure pfm_$signal(in *fault_signal*: status_$t);

SYNOPSIS (FORTRAN)

%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/pfm.ins.ftn'

integer*4 *fault_signal*

call pfm_$signal(*fault_signal*)

DESCRIPTION

Pfm_$signal signals the fault specified by *fault_signal* to the calling process. It is usually called to leave clean-up handlers.

fault_signal
A fault code.

NOTES

Pfm_$signal does not return when successful.

SEE ALSO

pfm_$error_trap.

pgm

Contents

NAME

intro – the program manager

SYNOPSIS (C)

#include <apollo/base.h>
#include <apollo/pgm.h>

SYNOPSIS (Pascal)

%include '/sys/ins/base.ins.pas';
%include '/sys/ins/pgm.ins.pas';

SYNOPSIS (FORTRAN)

%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/pgm.ins.ftn'

DESCRIPTION

The **pgm_$** calls allow programs to invoke other programs, pass system resources to child processes, and monitor their status.

Constants

pgm_$error
The severity level to indicate that an error occurred in an invoked program.

pgm_$false
A severity level to indicate that a condition was false.

pgm_$internal_fatal
A severity level to indicate that an internal fatal error occured in an invoked program.

pgm_$max_severity
The highest exit severity level that can be returned by an invoked program.

pgm_$ok
A severity level to indicate that an invoked program completed successfully.

pgm_$output_invalid
A severity level to indicate that the output from an invoked program is invalid.

pgm_$program_faulted
A severity level to indicate that an invoked program was faulted.

pgm_$true
A severity level to indicate that a condition was true.

pgm_$warning
A severity level to indicate a warning.

Data Types

pgm_$arg A record for passing program arguments. The following diagram illustrates the **pgm_$arg** data type:

15			0
len			
chars		...	
15	8	7	0

len The number of significant bytes in **chars**.

chars A character array to hold the text of the argument.

pgm_$argv
An array of program arguments.

pgm_$argv_ptr
A pointer to a program argument.

pgm_$connv
An array for passing stream IDs to invoked programs.

pgm_$ec_key
A key specifying a process eventcount. Currently, **pgm_$child_proc** is the only valid value. It designates an eventcount that advances when a child process terminates.

pgm_$mode
A small set type to specify the mode in which to invoke a program. It can take on any combination of the following predefined values:

pgm_$back_ground
The invoked program executes independently of the invoking process, and there is no valid process handle for it. Consequently, there is no mechanism by which the new process can communicate its exit status to the invoking process. Use this mode when the exit status of the invoked process in not important.

pgm_$extra_proc
This mode forces the invoked program to run in a separate process.

pgm_$obj_only
In this mode, the invoke call will fail if the invoked program is not a pre-SR10 object; that is, an object of type **obj**.

pgm_$wait
The invoking program is suspended while the invoked program executes. The termination status of the invoked program is supplied in the completion status of **pgm_$invoke** when it returns.

pgm_$proc
A pointer type used as a process handle.

Errors

pgm_$no_arg
There is no argument in the argument vector at the specified position.

pgm_$arg_too_big
The argument requested is too big to fit in the buffer allocated for it.

pgm_$bad_connv
Attempted to pass too many stream IDs in the connection vector.

pgm_$no_entry_point
There is no entry point in the invoked program.

pgm_$not_a_program
Attempted to invoke a file that is not an object.

pgm_$on_signal_stack
Attempted to invoke a program inprocess while running on the signal stack. See BSD sigstack(2).

pgm_$option_conflict
Specified incompatible options to **pgm_$invoke**.

pgm_$process_vforked
Cannot call **pgm_$invoke** from a process created by a BSD vfork(2).

pgm_$wrong_format
Attempted to invoke a COFF object module with the **pgm_$obj_only** option.

NAME

pgm_$del_arg – delete a program argument

SYNOPSIS (C)

```
#include <apollo/base.h>
#include <apollo/pgm.h>

void pgm_$del_arg(short &arg_number)
```

SYNOPSIS (Pascal)

```
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/pgm.ins.pas';

procedure pgm_$del_arg(in arg_number: integer);
```

SYNOPSIS (FORTRAN)

```
%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/pgm.ins.ftn'

      integer*2 arg_number

      call pgm_$del_arg(arg_number)
```

DESCRIPTION

Pgm_$del_arg deletes an argument from the argument vector.

arg_number

The index into the argument vector of the argument to delete. Arguments are numbered starting from 0, where argument 0 is usually the program invocation name. When **pgm_$del_arg** returns, arguments coming after position *arg_number* in the argument vector will be shifted down in position.

NAME

pgm_$exit – exit a program

SYNOPSIS (C)

```
#include <apollo/base.h>
#include <apollo/pgm.h>

void pgm_$exit(void)
```

SYNOPSIS (Pascal)

```
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/pgm.ins.pas';

procedure pgm_$exit;
```

SYNOPSIS (FORTRAN)

```
%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/pgm.ins.ftn'

      call pgm_$exit
```

DESCRIPTION

Pgm_$exit exits from the calling program and returns control to the process that invoked it. When **pgm_$exit** is called any files left open by the program are closed, any storage acquired is released, and asynchronous faults are re-enabled if they were inhibited by the calling program.

NOTES

Pgm_$exit calls **pfm_$signal** with a fault code equal to the last severity level set by **pgm_$set_severity**, or **pgm_$ok** (the default) if **pgm_$set_severity** was not called.

NAME

pgm_$get_arg – get a program argument

SYNOPSIS (C)

```
#include <apollo/base.h>
#include <apollo/pgm.h>

short pgm_$get_arg(
        short &arg_number,
        char *arg_buffer,
        status_$t *status,
        short &buffer_length)
```

SYNOPSIS (Pascal)

```
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/pgm.ins.pas';

function pgm_$get_arg(
        in arg_number: integer;
        out arg_buffer: univ pgm_$name;
        out status: status_$t;
        in buffer_length: integer): integer;
```

SYNOPSIS (FORTRAN)

```
%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/pgm.ins.ftn'

      integer*4 status
      integer*2 arg_length, arg_number, buffer_length
      character arg_buffer*128

      arg_length = pgm_$get_arg(arg_number, arg_buffer, status, buffer_length)
```

DESCRIPTION

Pgm_$get_arg writes an argument from the caller's argument vector into the buffer at *arg_buffer* and returns its length.

arg_number
: The number of the argument to get.

arg_buffer
: A buffer allocated to receive the argument.

status
: The completion status. If *status* is **pgm_$arg_too_big**, the argument was more than *buffer_length* bytes long and was truncated to fit.

buffer_length
: The number of bytes allocated to receive the argument. **Pgm_$get_arg** will not write more than *buffer_length* bytes into *arg_buffer*. If the value of

buffer_length is less than the returned argument length, the argument was truncated to fit in *arg_buffer*.

NAME

pgm_$get_args – get the program argument vector

SYNOPSIS (C)

```
#include &lt;apollo/base.h&gt;
#include &lt;apollo/pgm.h&gt;

void pgm_$get_args(
        short *argument_count,
        pgm_$argv_ptr *arg_vector_ptr)
```

SYNOPSIS (Pascal)

```
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/pgm.ins.pas';

procedure pgm_$get_args(
        out argument_count: integer;
        out arg_vector_ptr: pgm_$argv_ptr);
```

SYNOPSIS (FORTRAN)

```
%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/pgm.ins.ftn'

    integer*2 argument_count, arg_len
    integer*4 arg_ptr_vector(128)
    character arg_chars*128

    integer*4 arg_ptr, arg_vector_ptr
    pointer /arg_ptr/ arg_len, arg_chars
    pointer /arg_vector_ptr/ arg_ptr_vector

        call pgm_$get_args(argument_count, arg_vector_ptr)
```

DESCRIPTION

Pgm_$get_args supplies a pointer to the caller's argument vector and the number of arguments in it. The argument vector is an array of type **pgm_$arg**.

argument_count
: The number of arguments in the argument vector.

arg_vector_ptr
: A pointer to the argument vector.

NAME

pgm_$get_ec – get a process eventcount

SYNOPSIS (C)

```
#include <apollo/base.h>
#include <apollo/pgm.h>

void pgm_$get_ec(
        pgm_$proc &process_handle,
        pgm_$ec_key &ec_key,
        ec2_$ptr_t *ec_pointer,
        status_$t *status)
```

SYNOPSIS (Pascal)

```
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/pgm.ins.pas';

procedure pgm_$get_ec(
        in process_handle: pgm_$proc;
        in ec_key: pgm_$ec_key;
        out ec_pointer: ec2_$ptr_t;
        out status: status_$t);
```

SYNOPSIS (FORTRAN)

```
%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/pgm.ins.ftn'

        integer*4 status, process_handle, ec_value
        integer*2 ec_key, event(3)

        equivalence (ec_value, event(1))

        integer*4 ec_pointer
        pointer /ec_pointer/ event

        call pgm_$get_ec(process_handle, ec_key, ec_pointer, status)
```

DESCRIPTION

Pgm_$get_ec supplies a pointer to an eventcount that advances when the process on *process_handle* terminates. When a child process is created, the eventcount value is 0. When a child process terminates, the process eventcount value is 1.

process_handle
: A process handle of a child process. A process handle is not valid for programs invoked in background mode.

ec_key
: An enumerated value specifying which process eventcount **pgm_$get_ec** should supply. The only allowable value currently is **pgm_$child_proc**.

ec_pointer
An pointer to the eventcount.

status The completion status.

NAME

pgm_$get_puid – get a process UID

SYNOPSIS (C)

```
#include <apollo/base.h>
#include <apollo/pgm.h>

void pgm_$get_puid(
        pgm_$proc &process_handle,
        uid_$t *process_uid,
        status_$t *status)
```

SYNOPSIS (Pascal)

```
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/pgm.ins.pas';

procedure pgm_$get_puid(
        in process_handle: pgm_$proc;
        out process_uid: uid_$t;
        out status: status_$t)
```

SYNOPSIS (FORTRAN)

```
%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/pgm.ins.ftn'

        integer*4 process_handle, process_uid(2), status

        call pgm_$get_puid(process_handle, process_uid, status)
```

DESCRIPTION

Pgm_$get_puid supplies the process UID (Unique Identifier) of the process on *process_handle*.

process_handle
: A process handle for a child process. A process handle is not valid if the program was invoked in background mode.

process_uid
: The process UID of the process on *process_handle*.

status
: The completion status.

NAME

pgm_$invoke – invoke a program

SYNOPSIS (C)

```
#include <apollo/base.h>
#include <apollo/pgm.h>

void pgm_$invoke(
        char *path_name,
        short &path_length,
        short &argument_count,
        pgm_$arg_ptr *argument_vector,
        short &stream_count,
        stream_$id_t *stream_vector,
        pgm_$mode &invoke_mode,
        pgm_$proc *process_handle,
        status_$t *status)
```

SYNOPSIS (Pascal)

```
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/pgm.ins.pas';

procedure pgm_$invoke(
        in path_name: univ name_$pname_t;
        in path_length: integer;
        in argument_count: integer;
        in argument_vector: univ pgm_$argv;
        in stream_count: integer;
        in stream_vector: univ pgm_$connv;
        in invoke_mode: pgm_$mode;
        out process_handle: univ pgm_$proc;
        out status: status_$t);
```

SYNOPSIS (FORTRAN)

```
%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/pgm.ins.ftn'

        integer*4 process_handle, status
        integer*2 path_length, argument_count, arg_len, stream_count
        integer*2 stream_vector(128), invoke_mode
        character path_name*1023, arg_chars*128

        integer*4 arg_ptr, argument_vector(128)
        pointer /arg_ptr/ arg_len, arg_chars

      call pgm_$invoke(path_name, path_length, argument_count,
```

    & *argument_vector, stream_count,*
    & *stream_vector, invoke_mode,*
    & *process_handle, status)*

DESCRIPTION

Pgm_$invoke invokes the program at *path_name* and supplies a process handle for it in *process_handle*. The behavior of an invoked program depends on the mode in which the program is invoked.

path_name
The absolute pathname of the program to invoke.

path_length
The number of bytes in *path_name*.

argument_count
The number of arguments in *argument_vector*.

argument_vector
An array of program arguments of type **pgm_$arg**.

stream_count
The number of stream IDs in *stream_vector*.

stream_vector
An array of stream IDs to pass to the invoked program.

invoke_mode
A small set specifying the mode in which to invoke the program. Choose a combination of the following values:

pgm_$back_ground
The invoked program executes independently of the invoking process, and there is no valid process handle for it. Consequently, there is no mechanism by which the new process can communicate its exit status to the invoking process. Use this mode when the exit status of the invoked process in not important.

pgm_$extra_proc
This mode forces the invoked program to run in a separate process.

pgm_$obj_only
In this mode, the invoke call will fail if the invoked program is not a pre-SR10 object; that is, an object of type **obj**.

pgm_$wait
The invoking program is suspended while the invoked program executes. The termination status of the invoked program is supplied in the completion status of **pgm_$invoke** when it returns.

To invoke in default mode, specify 0 in C and FORTRAN, or [] in Pascal. When a program is invoked in default mode, **pgm_$invoke** supplies a process handle for the child process by which the parent process can monitor its status.

process_handle
A process handle for the invoked program. The process handle is not valid if the program was invoked in background mode.

status The completion status. If **pgm_$invoke** was called to invoke a program in wait mode, then *status* is the completion status of the invoked program or one of the following predefined severity levels:

pgm_$true
The value of a tested condition is **true**.

pgm_$false
The value of a tested condition is **false**.

pgm_$warning
An unusual, but not fatal, condition was detected.

pgm_$error
There were syntactic or semantic errors in the input, but the output is structurally sound.

pgm_$invalid_output
There were syntactic or semantic errors in the input, and the output is not structurally sound.

pgm_$internal_fatal
A fatal internal error detected.

NOTES

When a program is invoked in default mode, some of the parent's resources are reserved to keep track of the child processes whose handles are held by the parent process. A parent process that indiscriminately invokes programs in default mode will eventually exhaust system resources unless it orphans its child processes by calling **pgm_$make_orphan**.

SEE ALSO

pm_$intro.

NAME

pgm_$make_orphan – orphan a process

SYNOPSIS (C)

```
#include <apollo/base.h>
#include <apollo/pgm.h>

void pgm_$make_orphan(
        pgm_$proc &process_handle,
        uid_$t *process_uid,
        status_$t *status)
```

SYNOPSIS (Pascal)

```
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/pgm.ins.pas';

procedure pgm_$make_orphan(
        in process_handle: pgm_$proc;
        out process_uid: uid_$t;
        out status: status_$t);
```

SYNOPSIS (FORTRAN)

```
%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/pgm.ins.ftn'

        integer*4 process_handle, process_uid(2), status

        call pgm_$make_orphan(process_handle, process_uid, status)
```

DESCRIPTION

Pgm_$make_orphan changes the process on *process_handle* from a child into an orphan process. An orphan process executes in background mode, and runs independently of its parent process. A parent process can use **pgm_$make_orphan** to change a child process invoked in default mode into a process running in background mode. When **pgm_$make_orphan** returns, **process_handle** is no longer valid.

process_handle
The process handle of a child process.

process_uid
The process UID of the orphaned process.

status The completion status.

NAME

pgm_$proc_wait – wait on a process

SYNOPSIS (C)

```
#include <apollo/base.h>
#include <apollo/pgm.h>

void pgm_$proc_wait(
        pgm_$proc &process_handle,
        status_$t *status)
```

SYNOPSIS (Pascal)

```
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/pgm.ins.pas';

procedure pgm_$proc_wait(
        in process_handle: pgm_$proc;
        out status: status_$t);
```

SYNOPSIS (FORTRAN)

```
%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/pgm.ins.ftn'

        integer*4 process_handle, status

        call pgm_$proc_wait(process_handle, status)
```

DESCRIPTION

Pgm_$proc_wait suspends the calling process until the process on *process_handle* terminates. **Pgm_$proc_wait** returns with the completion status of the process waited for in *status*.

Using **pgm_invoke** to invoke a program in default mode and then calling **pgm_$proc_wait** is effectively the same as invoking a program in wait mode.

process_handle
The process handle of the process to wait for. A process cannot wait on a program invoked in background mode.

status The child process completion status.

NAME

pgm_$set_severity – set the exit severity level

SYNOPSIS (C)

```
#include <apollo/base.h>
#include <apollo/pgm.h>

void pgm_$set_severity(pgm_$mode &severity_level)
```

SYNOPSIS (Pascal)

```
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/pgm.ins.pas';

procedure pgm_$set_severity(in severity_level: integer);
```

SYNOPSIS (FORTRAN)

```
%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/pgm.ins.ftn'

      integer*2 severity_level

      call pgm_$set_severity(severity_level)
```

DESCRIPTION

Pgm_$set_severity sets the severity level for the calling process to *severity_level*. Every invoked program returns a severity level to its invoking process, and by default the severity level is **pgm_$ok**.

Choosing an Exit Severity Level

The following are examples of appropriate choices for an exit severity level:

- **Pgm_$true** or **pgm_$false** would be returned by a program that compared two values or tested some other condition.
- **Pgm_$warning** would be returned by a file deleting program when its target file didn't exist.
- **Pgm_$error** would be returned by a compiler if the input program contained a correctable syntax error, but the object was correct.
- **Pgm_$output_invalid** would be returned by a compiler if an error in the input program resulted in an incorrect object.
- **Pgm_$internal_fatal** would be returned by a program that could not proceed because its data was corrupted.
- **Pgm_$program_faulted** would be returned by a program that received a fault and wished to inform the invoking program without resignaling the fault.

severity_level

The exit severity level to return to the invoking process. Specify only one of the following values:

pgm_$ok
The program completed successfully and performed the requested action.

pgm_$true
The program completed successfully. Its purpose was to test a condition, and the value of that condition was true.

pgm_$false
The program completed successfully. Its purpose was to test a condition, and the value of that condition was false.

pgm_$warning
The program completed successfully and performed the requested action. However, an unusual (but nonfatal) condition was detected.

pgm_$error
The program could not perform the requested action because of syntactic or semantic errors in the input, though the output is sound.

pgm_$output_invalid
The program could not perform the requested action because of syntactic or semantic errors in the input, and the output is not sound.

pgm_$internal_fatal
The program detected an internal fatal error and ceased processing. The state of the output is not defined.

pgm_$program_faulted
The program detected and handled a fault.

If *severity_level* is set greater than **pgm_$max_severity**, the status returned to the invoking process will be just **pgm_$max_severity**.

NOTES

C programs that use **pgm_$set_severity** must be bound with the **-entry** option to **/com/bind** or the **-e** option to **/bin/ld** to specify the start routine manually (normally ''main''). The best alternative to **pgm_$set_severity** is to use a UNIX exit(2) or a **return** to return a program status.

pm

Contents

NAME

intro – the Domain/OS Process Manager

SYNOPSIS (C)

#include <apollo/base.h>
#include <apollo/pm.h>

SYNOPSIS (Pascal)

%include '/sys/ins/base.ins.pas';
%include '/sys/ins/pm.ins.pas';

SYNOPSIS (FORTRAN)

%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/pm.ins.ftn'

DESCRIPTION

The **pm_$** calls supply information from the environment of the calling process.

Pm_$get_home_txt reports the home directory of the calling process, and **pm_$get_sid_txt** reports the Subject Identifier (SID) of the calling process.

Variable

pm_$errout

A per-process external variable whose value is the stream ID of the standard error output. Domain/OS supports two different object formats: the old object format (obj) and the new Common Object File Format (COFF) which use different standard stream IDs. **Pm_$errout** allows dynamically loaded code to use one model or the other depending on the object format.

Data Type

pm_$sidtext_t

An array for passing SIDs as strings.

NAME

pm_$get_home_txt – get the home directory

SYNOPSIS (C)

#include <apollo/base.h>
#include <apollo/pm.h>

```
void pm_$get_home_txt(
        short &buffer_length,
        char *path_name,
        short *path_length)
```

SYNOPSIS (Pascal)

```
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/pm.ins.pas';

procedure pm_$get_home_txt(
        in buffer_length: integer;
        out path_name: univ name_$long_pname_t;
        out path_length: integer);
```

SYNOPSIS (FORTRAN)

```
%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/pm.ins.ftn'

        integer*2 buffer_length, path_length
        character path_name*1023

        call pm_$get_home_txt(buffer_length, path_name, path_length)
```

DESCRIPTION

Pm_$get_home_txt supplies the home directory associated with the SID of the calling process in *path_name*.

buffer_length
: The number of bytes in the buffer allocated to receive the home directory. **Pm_$get_home_txt** will not write more than *buffer_length* characters into *path_name*.

path_name
: The pathname of the home directory for the calling process. The home directory supplied in *path_name* is the one associated with the SID of the calling process and is the same as the **HOME** variable in the caller's environment.

path_length
: The number of bytes in the pathname of the home directory. If the value supplied in *path_length* equals or exceeds the value passed in *buffer_length* then the pathname written in *path_name* might be truncated.

SEE ALSO

name_$get_ndir_lc, name_$get_wdir_lc.

NAME

pm_$get_sid_txt – get the SID

SYNOPSIS (C)

```
#include <apollo/base.h>
#include <apollo/pm.h>

void pm_$get_sid_txt(
        short &buffer_length,
        char *sid,
        short *sid_length);
```

SYNOPSIS (Pascal)

```
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/pm.ins.pas';

procedure pm_$get_sid_txt(
        in buffer_length: integer;
        out sid: univ pm_$sidtext_t;
        out sid_length: integer);
```

SYNOPSIS (FORTRAN)

```
%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/pm.ins.ftn'

        integer*2 buffer_length, sid_length
        character sid*140

        call pm_$get_sid_txt(buffer_length, sid, sid_length)
```

DESCRIPTION

Pm_$get_sid_txt supplies the SID for the calling process in *sid*. A SID is a key to the permissions granted to a process, and has the form ''*PERSON.GROUP.ORGANIZATION*'', where *PERSON* is the login of the owner of the process, *GROUP* is the process owner's current group, and *ORGANIZATION* is the process owner's current organization or ''project.''

buffer_length
The number of bytes in the buffer allocated to receive the SID. **Pm_$get_sid_txt** will not write more than *buffer_length* characters into *sid*.

sid The SID for the calling process.

sid_length
The number of bytes in the SID of the calling process. If the value supplied in *sid_length* equals or exceeds the value passed in *buffer_length*, then the SID string written in *sid* might be truncated.

prf

Contents

NAME

intro – the Domain/OS print library

SYNOPSIS (C)

#include <apollo/base.h>
#include <apollo/prf.h>

SYNOPSIS (Pascal)

%include '/sys/ins/base.ins.pas';
%include '/sys/ins/prf.ins.pas';

SYNOPSIS (FORTRAN)

%include '/sys/ins/base.ins.ftn'

DESCRIPTION

The **prf_$** calls allow an application to use Domain/OS printing services.

Domain/OS supports a network-wide collection of print managers which route print jobs to appropriate print servers. The **prf_$get_sites** and **prf_$get_printers** system calls supply information about the printing resources currently available on a Domain internet.

Starting Print Jobs

A print job consists of a print file containing data to be printed, and an associated print request. A print request is a structured file containing a list of print option name/value pairs that control the result of a print operation. The print file can be spooled from a stream, a spooled copy of a file retained by a user, or even another file not on the print spool.

The data structure queued in a print request is allocated whenever a process accesses the print library, and initialized via the **prf_$init** call. Thereafter, a process can modify the results of a print job by changing print option values from their defaults. The print option names are strings, but the print option values may be specified as strings or as numeric values. When specified numerically the print option values have type **num_var_t**.

After setting options via **prf_$config_file**, and **prf_$set_option** a process can queue a print request with **prf_$name_print**, **prf_$queue_file**, or **prf_$stream_print**.

Until the next call to **prf_$init**, the process may find out and alter the status of the print services relevant to the queued print job with **prf_$edit_job**. The **prf_$read_queue** and **prf_$signal_printer** calls allow a process to find out what jobs are pending or in progress and delete or abort them.

Print Options

The following print options apply to all types of files:

SEARCH_DIR ON|OFF
: **ON** tells the print server to search through the working directories of all the active processes on the node for the files to be printed. **OFF**, the default, tells the print server to search only the current working directory of the calling process.

COPIES *n*

Print *n* copies of the file. One copy is printed by default.

PRINTER *name*

Print the file on the printer named *name*. A printer's name is defined in its configuration file. If this option is not specified, the print server uses the default printer name "P".

SITE *node_entry_dir*

Queue the print request on the node whose node entry directory is *node_entry_dir*.

DELETE ON|OFF

ON tells the print server to delete the print file after it has been printed. The default state is on; that is, if this option is not specified, the print server deletes the print file indicated by the print request after printing.

USER_NAME *name*

Print *name* as the user name on the banner page of the printed file. If the **SIGNAL** option is set to **ALARM**, the print server also sends an alarm to the user specified in *name* when printing is complete. Therefore, *name* must be a valid log-in name when requesting an alarm from the print server.

SIGNAL ALARM|OFF

ALARM tells the print server to notify a user when printing is finished. The print server sends an alarm to the user specified with the **USER_NAME** option if it is set, or to the current log-in name for the calling process if the **USER_NAME** option is not set. This option's default value is **OFF**.

BANNER ON|OFF

ON tells the print server to print a banner page displaying a job name, log-in name, and system information. **OFF**, the default, tells the print server to not print a banner page.

CONFIG_FILE *pathname*

Read print request options from *pathname*. If *pathname* is null, the print server reads print request options from the configuration file at **~/user_data/prf.db**.

TEXT Specifies text mode for printing ASCII files. **TEXT** is the default print mode.

PLOT ON|OFF

Use **PLOT ON** to print GPR bitmaps or to print from graphics metafiles. **PLOT OFF** is the default.

TRANSPARENT ON|OFF

ON tells the print server to pass the file directly to the printer driver routine with no processing by the print server; that is, the printer can interpret the file directly. **OFF**, the default, tells the print server to do it's normal interpretation for the target printer.

PAPER_SIZE A|B|LEGAL|A3|A4|A5|B4|B5
Use the specified page size on the Domain/Laser-26 printer.

Paper Size Codes	
size code	**paper dimensions in inches (mm)**
A	8.50 × 11.00 (218 × 282)
B	11.00 × 17.00 (282 × 436)
LEGAL	8.50 × 14.00 (218 × 359)
A3	11.69 × 16.54 (297 × 420)
A4	8.27 × 11.69 (210 × 297)
A5	5.38 × 8.27 (137 × 210)
B4	9.84 × 13.90 (257 × 364)
B5	5.93 × 9.89 (182 × 257)

This option is only meaningful for Domain/Laser-26 and APPLE LaserWriter printers.

The following print options apply to text files only:

MARGINS ON|OFF
ON tells the print server to use the margin settings specified with the **TOP**, **BOTTOM**, **RIGHT**, and **LEFT** options. **OFF** tells the print server to ignore the **TOP**, **BOTTOM**, **RIGHT**, and **LEFT** options. The default state is **ON**.

TOP *n* The top margin is *n* inches, where *n* is a real number.

BOTTOM *n*
The bottom margin is *n* inches, where *n* is a real number.

RIGHT *n*
The right margin is *n* inches, where *n* is a real number.

LEFT *n*
The left margin is *n* inches, where *n* is a real number.

HEADERS ON|OFF

ON tells the print server to use the header and footer defined with the **HEAD_STRING** and **FOOT_STRING** options. **OFF** tells the print server to ignore the header and footer defined with the **HEAD_STRING** and **FOOT_STRING** options. The default state is **ON**.

HEAD_STRING *left/center/right*

Use this option to define a page header. Specify the header as a single string of three components delimited with slashes (/). The resulting header has *left* printed flush against the left margin, *center* centered on the page, and *right* printed flush against the right margin. Components may be empty; that is, "//" is a valid string and results in a blank header. The print server interprets the following special character sequences and substitutes the strings indicated when the characters appear in the header string:

Special Character	Substituted String
@	The escape character. Turns off any special interpretation of the following character. For instance, "@*" results in a "*" instead of a space.
#	The current page number with a leading and a trailing space.
%	The current date.
!	The name of the file being printed.
&	The time and date the file being printed was last modified.
*	A space. Inserts a space in the header. Literal spaces are not permitted in header strings.

The header string "Page*@##/!/Printed*on**%" will produce a header with the label "Page #" followed by the page number flush left, the file name centered, and the label "Printed on " followed by the current date flush right.

FOOT_STRING *left/center/right*

Use this option to define a page footer. The format is the same as for **HEAD_STRING**.

FTN ON|OFF

ON tells the print server to use FORTRAN forms control even if the file does not have the FORTRAN carriage control flag. With **FTN ON**, the print server interprets the first character of each line as a FORTRAN carriage control character and doesn't print it, so the first character of every line will be lost if the file does not use FORTRAN carriage control. The default state is **OFF**.

WRAP ON|OFF
: **WRAP ON** tells the print server to wrap any lines that exceed the right margin onto the next line. **WRAP OFF** tells the print server to truncate lines that exceed the right margin. The default state is **OFF**.

The following print options apply to plot files; that is, graphics metafiles and GPR bitmaps:

RESOLUTION *n*
: Print the file with a resolution as close to *n* dots per inch as possible, where *n* is an integer. This option applies only to plot files.

WHITE_SPACE *n*
: Leave *n* inches of white space between plots, where *n* is a real number. The default is 3 inches, or "WHITE_SPACE 3".

BW_REV ON|OFF
: **ON** tells the print server to reverse black and white values in bitmaps. The default state is **OFF**.

MAGNIFICATION *n*
: Use this option to specify a bitmap magnification value, where *n* is an integer in the range -1 to 16. **MAGNIFICATION -1** tells the print server to scale the bitmap to fill the available page space. **MAGNIFICATION 0** tells the print server to print a graphics metafile with a one-to-one scaling between the display and the printer. If the print file is a GPR bitmap, **MAGNIFICATION 0** is equivalent to **MAGNIFICATION 1**. A magnification value of 1 through 16 scales the bitmap by that value. Regions of the magnified bitmap that extend beyond the printer's page boundaries are clipped.

The following print options are for printers that support variable font and pitch sizes:

PITCH *n*
: Set the printer's pitch to *n* characters per inch. The following pitch settings are supported:

Printer Pitch Settings	
Printer	**Pitch**
Printronix	10
Spinwriter	12
IMAGEN	8.5, 10, 12, 15, 17.1
GE 3000	10, 12, 13.1, 16.7
Versatec	12
LaserWriter	1 to 100
Laser-26	1 to 100

POINT *n*
: Set the size of the font to *n* points.

WEIGHT LIGHT|MEDIUM|BOLD
: Use this option to set the font weight for a GE 3000 printer; it is not valid with other printers. The default is **WEIGHT MEDIUM**.

LQ ON|OFF
: **ON** tells the print server to print the document at letter quality. **OFF** tells the print server to print in draft mode. The default state is **OFF**. The **LQ** option is valid only for GE 3000 printers.

The following options are for printers that contain a POSTSCRIPT interpreter, such as the Domain/Laser-26 and APPLE LaserWriter.

POSTSCRIPT ON|OFF
: **ON** tells the print server to invoke the printer's POSTSCRIPT interpreter while printing. **OFF**, the default, tells the print server to bypass the printer's POSTSCRIPT interpreter.

COLUMNS 1|2
: Format text in the specified number of columns. This option is meaningful only to a POSTSCRIPT printer. **COLUMNS 1** is the default.

LPI *n* Set the printer's line spacing to *n* lines per inch, where *n* is a real number.

ORIENTATION PORTRAIT|LANDSCAPE
: **PORTRAIT** tells the print server to print text lines or the x-axis of bitmaps parallel to the short edge of the page. **LANDSCAPE** tells the print server to print text lines or the x-axis of bitmaps perpendicular to the short leading edge. This option applies only to printers that include

the POSTSCRIPT interpreter (i.e., Laser-26, LaserWriter, GENICOM, and V80 printers with the POSTSCRIPT decomposer software). The default state is **PORTRAIT**. Specifiying this option will override any auto-rotation performed on bitmaps.

Data Types

num_var_t

This is a variant record type for the numeric value of a print request option.

15	0
real32	
real32	
15	0

OR

15	0
int32	
int32	
15	0

OR

15	0
NOT USED	
int16	
15	0

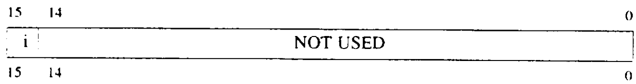
OR

15	0
NOT USED	
int16_n	
15	0

OR

15			0
NOT USED			
NOT USED		int8	
15	8	7	0

OR



real32 A 32-bit real number.

int32 A positive 31-bit number.

int16 A positive 16-bit number.

int16_n A positive or negative 16-bit number.

int8 A positive 8-bit integer.

int1 (labeled as "i" in the diagram) A boolean value.

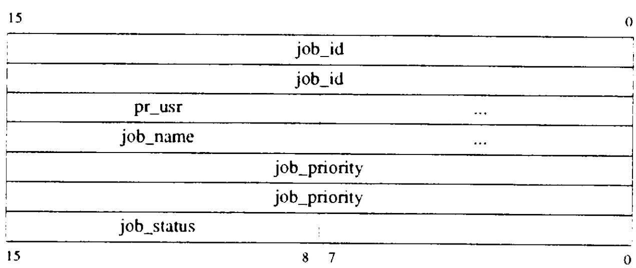
prf_$edit_job_t
An enumerated type for specifying the intended operation in the **prf_$edit_job** call. It takes one of the following values:

prf_$del_job
Delete the job.

prf_$prior
Alter the job's priority.

prf_$time_print
Postpone printing the job.

prf_$job_entry_t
A record for storing print job information.



job_id An integer label for the job.

pr_usr The user name or printer name for the job. It has type **prf_$name_t**.

job_name
The name of the job. It has type **prf_$name_t**.

job_priority
The priority of the job.

job_status
The status of the job. It has type **prf_$name_t**.

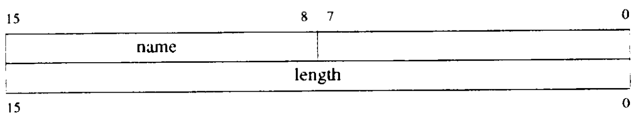
prf_$job_status_t
An enumerated type for specifying the status of a print job. It takes one of the following values:

prf_$prntg
The job is printing.

prf_$idle
The job is idle and waiting to print.

prf_$susp
The job is suspended awaiting a continue signal.

prf_$name_t
A record for passing string arguments to **prf_$** calls.



name The string. It may be up to 32 bytes long.

length The number of bytes in *name*.

prf_$pr_sig_t
An enumerated type for specifying the signal to send with **prf_$signal_printer**. It takes one of the following values:

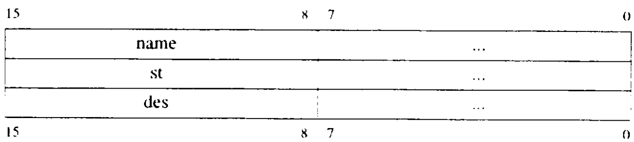
prf_$abort_job
Abort the job.

prf_$suspend
Suspend the job.

prf_$continue
Continue the job.

prf_$printer_array_t
An array of type **prf_$printer_t**.

prf_$printer_t
A record type for holding information on a printer.



name The name of the printer. It has type **prf_$name_t**.

st The status of the printer. It has type **prf_$name_t**.

des A description of the printer. It has type **prf_$name_t**.

prf_$read_opt_t
An enumerated type for restricting the information returned by **prf_$read_queue**. The type can currently take on only one possible value:

prf_$read_by_printer
Return a list of jobs by printer.

NOTES

APPLE and LaserWriter are registered trademarks of Apple Computer, Inc. GENICOM is a trademark of Genicom corporation. IMAGEN is a registered trademark of the IMAGEN Corporation. POSTSCRIPT is a registered trademark of Adobe Systems. Spinwriter is a registered trademark of NEC. Versatec is a trademark of Versatec, Inc.

NAME

prf_$config_file – set print options from a file

SYNOPSIS (C)

```
#include <apollo/base.h>
#include <apollo/prf.h>

void prf_$config_file(
        char *name,
        short &name_length,
        status_$t *status)
```

SYNOPSIS (Pascal)

```
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/prf.ins.pas';

procedure prf_$config_file(
        in name: name_$pname_t;
        in name_length: integer;
        out status: status_$t);
```

SYNOPSIS (FORTRAN)

```
%include '/sys/ins/base.ins.ftn'

      integer*2 name_length
      integer*4 status
      character name*1024

      call prf_$config_file(name, name_length, status)
```

DESCRIPTION

This call reads print request options from the file at *name* into the current print data structure.

name The full pathname of the print configuration file.

name_length
The number of bytes in *name*.

status The completion status.

SEE ALSO

prf_$inq_option, prf_$set_option.

NAME

prf_$edit_job – edit a print job at the current site

SYNOPSIS (C)

```
#include <apollo/base.h>
#include <apollo/prf.h>

void prf_$edit_job(
        long &job_id,
        prf_$edit_job_t &edit_op,
        status_$t *status)
```

SYNOPSIS (Pascal)

```
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/prf.ins.pas';

procedure prf_$edit_job(
        in job_id: integer32;
        in edit_op: prf_$edit_job_t;
        out status: status_$t);
```

SYNOPSIS (FORTRAN)

```
%include '/sys/ins/base.ins.ftn'

        integer*4 job_id, status
        integer*2 edit_op

        call prf_$edit_job(job_id, edit_op, status)
```

DESCRIPTION

This call performs the operation specified by *edit_op* on the job specified by *job_id*. The *job_id* must refer to an entry in the queue at the current site.

job_id The ID of the print job to edit. The *job_id* is the first member of a record of type **prf_$job_entry_t**.

edit_op The edit operation to perform on the job. Specify one of the following:

prf_$del_job
Delete the job.

prf_$prior
Alter the job's priority.

prf_$time_print
Postpone printing the job.

status The completion status.

NOTES

Use **prf_$read_queue** to get an array of job entries, of type **prf_$job_entry_t**, from the queue at the current print site.

SEE ALSO

prf_$delete_entry.

NAME

prf_$get_printers – get a list of printers at the current site

SYNOPSIS (C)

```
#include &lt;apollo/base.h&gt;
#include &lt;apollo/prf.h&gt;

void prf_$get_printers(
        prf_$name_t *site,
        long *entry,
        long &max_printers,
        long *ret_printers,
        prf_$printer_t *printer_list,
        status_$t *status)
```

SYNOPSIS (Pascal)

```
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/prf.ins.pas';

procedure prf_$get_printers(
        in site: univ prf_$name_t;
        var entry: integer32;
        in max_printers: integer32;
        out ret_printers: integer32;
        out printer_list: univ prf_$printer_array_t;
        out status: status_$t);
```

SYNOPSIS (FORTRAN)

```
%include '/sys/ins/base.ins.ftn'

      integer*2 n_printers, example
      parameter (n_printers = 32, example = 3)

      integer*4 entry, max_printers, ret_printers
      character printer_list(n_printers)*102

      equivalence (printer_name, printer_list(example))
      equivalence (printer_name_length, printer_list(example)(33:34))
      equivalence (printer_status, printer_list(example)(35:66))
      equivalence (printer_status_length, printer_list(example)(67:68))
      equivalence (printer_description, printer_list(example)(69:100))
      equivalence (printer_description_length, printer_list(example)(101:102))

      call prf_$get_printers(site, entry, max_printers,
     &                      ret_printers, printer_list, status)
```

DESCRIPTION

This call supplies up to *max_printers* printer names from the current site in the *printer_list* argument. The *entry* argument passed to **prf_$get_printers** controls which portion of the current site's printer list is supplied in *printer_list*.

site The print site to list printers for. If *site* is null, then all sites on the network are polled.

entry An index into the printer list at the current site. *Index* specifies the position in the printer list of the first printer requested when **prf_$get_printers** is called; it contains the position in the list of the next printer to request when **prf_$get_printers** returns.

When calling **prf_$get_printers**, an *entry* of 1 will cause **prf_$get_printers** to supply the first *max_printers* printer names from the site's printer list in *printer_list*. Then when **prf_$get_printers** returns, *entry* will be the position of the next unread printer name in the list.

If the difference between the call and return values of *entry* is less than *max_printers*, then *printer_list* contains the name of the last printer in the site list. If the difference is equal to *max_printers*, there may be more printers at the site; that is, *printer_list* may not contain the name of the last printer in the printer list. If the printer list is exhausted when **prf_$get_printers** returns, *entry* contains the position of the last printer, or, more simply, just the number of printers at the current site.

max_printers

The maximum number of printers that **prf_$get_printers** should supply. The value of *max_printers* depends on the size of the buffer allocated to receive the *printer_list* argument.

ret_printers

The number of printer descriptions supplied in *printer_list*.

printer_list

An array of type **prf_$name_t** containing a list of printer descriptions. Each element of the array contains a printer's name, its description, and its status.

status The completion status.

SEE ALSO

prf_$get_sites, prf_$read_queue.

NAME

prf_$get_sites – get a list of print sites

SYNOPSIS (C)

```
#include <apollo/base.h>
#include <apollo/prf.h>

void prf_$get_sites(
        long *index,
        long &max_sites,
        long *ret_sites,
        prf_$name_t *site_list,
        status_$t *status)
```

SYNOPSIS (Pascal)

```
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/prf.ins.pas';

procedure prf_$get_sites(
        var index: integer32;
        in max_sites: integer32;
        out ret_sites: integer32;
        out site_list: univ prf_$name_array_t;
        out status: status_$t);
```

SYNOPSIS (FORTRAN)

```
%include '/sys/ins/base.ins.ftn'

        integer*2 n_sites, example
        parameter (n_sites = 32, example = 3)

        integer*4 index, max_sites, ret_sites, status
        character site_list(n_sites)*34

        integer*2 site_location_length
        character site_location*32

        equivalence (site_location, site_list(example))
        equivalence (site_location_length, site_list(example) (33:34))

        call prf_$get_sites(index, max_sites, ret_sites, site_list, status)
```

DESCRIPTION

This call supplies the locations of up to *max_sites* print sites on the network. The *index* argument passed to **prf_$get_sites** controls which portion of the network site list is supplied in *site_list*.

index An index into the list of sites supplied by **prf_$get_sites**. *Index* specifies the position in the network site list of the first site requested when **prf_$get_sites** is called; it contains the position in the list of the next site to request when **prf_$get_sites** returns.

When calling **prf_$get_sites**, an *index* of 1 will cause **prf_$get_sites** to supply the first *max_sites* site locations. Then when **prf_$get_sites** returns, *index* will be the position of the next unread site location in the list.

If the difference between the call and return values of *index* is less than *max_sites*, then *site_list* contains the location of the last site in the network site list. If the difference is equal to *max_sites*, there may be more sites on the network; that is, *site_list* may not contain the location of the last print site in the site list. If the site list is exhausted when **prf_$get_sites** returns, *index* contains the position of the last site, or, more simply, just the number of sites on the network.

max_sites
The maximum number of sites that **prf_$get_sites** should supply. The value of *max_sites* depends on the size of the buffer allocated to receive the *site_list* argument.

ret_sites
The number of sites supplied in *site_list*.

site_list An array of type **prf_$name_t** containing a list of sites. Each element of the array is a string containing the site's location followed by the number of bytes in the string.

status The completion status.

SEE ALSO
prf_$get_printers.

NAME

prf_$init – initialize print request options

SYNOPSIS (C)

```
#include <apollo/base.h>
#include <apollo/prf.h>

void prf_$init(
        ios_$id_t &stream_id,
        status_$t *status)
```

SYNOPSIS (Pascal)

```
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/prf.ins.pas';

procedure prf_$init(
        in stream_id: ios_$id_t;
        out status: status_$t);
```

SYNOPSIS (FORTRAN)

```
%include '/sys/ins/base.ins.ftn'

        integer*2 stream_id
        integer*4 status

        call prf_$init(stream_id, status)
```

DESCRIPTION

This call resets the options in the current print data structure to their default values and establishes a stream to use for error and status information returned from the print server.

stream_id
: The stream to write error and status messages to.

status
: The completion status.

SEE ALSO

prf_$config_file, prf_$inq_option, prf_$set_option.

NAME

prf_$inq_option – get a print request option

SYNOPSIS (C)

```
#include <apollo/base.h>
#include <apollo/prf.h>

void prf_$inq_option(
        char *option,
        short *option_length,
        num_var_t *number_value,
        char *string_value,
        short *string_length,
        status_$t *status)
```

SYNOPSIS (Pascal)

```
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/prf.ins.pas';

procedure prf_$inq_option(
        var option: univ string;
        var option_length: integer;
        out number_value: num_var_t;
        out string_value: univ string;
        out string_length: integer;
        out status: status_$t)
```

SYNOPSIS (FORTRAN)

```
%include '/sys/ins/base.ins.ftn'

      integer*2 opt_len
      parameter (opt_len = 120)

      integer*2 option_length, string_length
      integer*4 number_value, status
      character option*(opt_len), string_value*256

      call prf_$inq_option(option, option_length, number_value,
     &                     string_value, string_length, status)
```

DESCRIPTION

This call gets the string and numeric value for the option specified in *option*.

option The name of the option inquired about.

option_length
The number of bytes in *option*.

number_value
The numeric value of the option requested in *option*.

string_value
The string value of the option requested in *option*.

string_length
The number of bytes in *string_value*.

status The completion status.

SEE ALSO
prf_$config_file, prf_$set_option.

NAME

prf_$name_print – spool a file for printing

SYNOPSIS (C)

```
#include <apollo/base.h>
#include <apollo/prf.h>

void prf_$name_print(
        char *pathname,
        short &pathname_length,
        char *queue,
        short &queue_length,
        status_$t *status)
```

SYNOPSIS (Pascal)

```
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/prf.ins.pas';

procedure prf_$name_print(
        in pathname: univ name_$pname_t;
        in pathname_length: integer ;
        out queue: name_$pname_t;
        out queue_length: integer;
        out status: status_$t);
```

SYNOPSIS (FORTRAN)

```
%include '/sys/ins/base.ins.ftn'

      integer*2 pathname_length, queue_length
      integer*4 status
      character pathname*1024, queue*1024

      call prf_$name_print(pathname, pathname_length, queue,
     &                     queue_length, status)
```

DESCRIPTION

This call copies the file at *pathname* to the print spool and queues a print request for the job using the option values in the current print data structure. **Prf_$name_print** supplies the pathname of the queued print request in *queue* when it returns.

pathname
: The pathname of the file to print.

pathname_length
: The number of bytes in *pathname*.

queue
: The full pathname of the queued print request.

queue_length
The number of bytes in *queue*.

status The completion status.

NOTES

Both **prf_$name_print** and **prf_$queue_file** queue a print reqest for a file, and thus initiate a print job to print a file. The difference between them is that **prf_$name_print** copies the file to a print spool, and the resulting print request prints from the spooled copy. Consequently, when the print job terminates and the print server deletes the file indicated by the print request, it is the spooled copy that is deleted — not the original file named in the **prf_$name_print** call. **Prf_$queue_file** doesn't copy the file to a print spool; so, when the print job terminates and the print server deletes the file indicated by the print request, the original file named in the **prf_$queue_file** call is deleted.

Therefore, while setting DELETE OFF for a print request queued by **prf_$queue_file** is a good safety precaution for preserving the original file, setting DELETE OFF for a print request queued by **prf_$name_print** will leave orphaned files on the print spool and is not generally a good idea.

SEE ALSO

prf_$stream_print.

NAME

prf_$queue_file – queue a print request

SYNOPSIS (C)

```
#include <apollo/base.h>
#include <apollo/prf.h>

void prf_$queue_file(
        char *pathname,
        short &pathname_length,
        char *queue,
        short &queue_length,
        status_$t *status)
```

SYNOPSIS (Pascal)

```
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/prf.ins.pas';

procedure prf_$queue_pathname(
        in pathname: univ name_$pname_t;
        in pathname_length: integer;
        out queue: name_$pname_t;
        out queue_length: integer;
        out status: status_$t);
```

SYNOPSIS (FORTRAN)

```
%include '/sys/ins/base.ins.ftn'

      integer*2 pathname_length, queue_length
      integer*4 status
      character pathname*1024, queue*1024

      call prf_$queue_file(pathname, pathname_length, queue,
     &                     queue_length, status)
```

DESCRIPTION

This call queues a print request for the file at *pathname*. It does not copy the file at *pathname* to the print spool, so the print server will delete the original file itself rather than a spooled copy when the print job completes unless the **DELETE** option is set to **OFF**. When **prf_$queue_file** returns, it supplies the pathname of the queued print request in *queue*.

pathname

The pathname of the file to print.

pathname_length

The number of bytes in *pathname*.

queue The pathname of the queued print request.

queue_length
The number of bytes in *queue*.

status The completion status.

NOTES

Both **prf_$name_print** and **prf_$queue_file** queue a print reqest, and thus initiate a print job to print a file. The difference between them is that **prf_$name_print** copies the file to a print spool and the resulting print request prints from the spooled copy. Consequently, when the print job terminates and the print server deletes the file indicated by the print request, it is the spooled copy that is deleted — not the original file named in the **prf_$name_print** call. **Prf_$queue_file** doesn't copy the file to a print spool, so when the print job terminates and the print server deletes the file indicated by the print request, the original file named in the **prf_$queue_file** call is deleted.

Therefore, while setting **DELETE OFF** for a print request queued by **prf_$queue_file** is a good safety precaution for preserving the original file, setting **DELETE OFF** for a print request queued by **prf_$name_print** will leave orphaned files on the print spool and is not generally a good idea.

SEE ALSO

prf_$stream_print.

NAME

prf_$read_queue – get a list of print jobs in the queue

SYNOPSIS (C)

```
#include <apollo/base.h>
#include <apollo/prf.h>

void prf_$read_queue(
        prf_$read_opt_t &printer_flag,
        long *index,
        long &max_jobs,
        long *ret_jobs,
        prf_$job_entry_t *job_list,
        status_$t *status)
```

SYNOPSIS (Pascal)

```
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/prf.ins.pas';

procedure prf_$read_queue(
        in printer_flag: prf_$read_opt_t;
        var index: integer32;
        in max_jobs: integer32;
        out ret_jobs: integer32;
        out job_list: univ prf_$job_array_t;
        out status: status_$t);
```

SYNOPSIS (FORTRAN)

```
%include '/sys/ins/base.ins.ftn'

        integer*2 n_jobs, example
        parameter (n_jobs = 32, example = 3)

        integer*4 printer_flag, index, status
        integer*4 max_jobs, ret_jobs
        integer*2 job_list(39, n_jobs)

        integer*4 job_id, job_priority
        integer*2 print_user_length, job_name_length, job_status
        character print_user*32, job_name*32

        equivalence (job_id, job_list(1, example))
        equivalence (print_user, job_list(3, example))
        equivalence (print_user_length, job_list(19, example))
        equivalence (job_name, job_list(20, example))
        equivalence (job_name_length, job_list(36, example))
```

```
      equivalence (job_priority, job_list(37, example))
      equivalence (job_status, job_list(39, example))

      call prf_$read_queue(printer_flag, index, max_jobs,
     &                     ret_jobs, job_list, status)
```

DESCRIPTION

This call supplies a list of queued jobs at the current site.

printer_flag
: An enumerated value that specifies how jobs will be listed in *job_list*. Currently, the only legal value for *printer_flag* is **prf_$read_by_printer**, which causes **prf_$read_queue** to return only those jobs queued for the printer specified in the current print data structure.

index
: An index into the jobs in the current queue. *Index* specifies the position in the queue of the first job requested when **prf_$read_queue** is called; it contains the position in the queue of the next job to request when **prf_$read_queue** returns.

When calling **prf_$read_queue**, an *index* of 1 will cause **prf_$read_queue** to supply the first *max_jobs* jobs from the queue in *job_list*. Then when **prf_$read_queue** returns, *index* will be the position of the next unread job in the queue.

If the difference between the call and return values of *index* is less than *max_jobs*, then *job_list* contains the last job in the queue. If the difference is equal to *max_jobs*, there may be more jobs in the queue; that is, *job_list* may not contain the last job in the queue. If the job list is exhausted when **prf_$read_queue** returns, *index* contains the position of the last job, or, more simply, just the number of jobs queued at the current site.

max_jobs
: The maximum number of jobs that **prf_$read_queue** should supply. The value of *max_jobs* depends on the size of the buffer allocated to receive the *job_list* argument.

ret_jobs
: The number of jobs supplied in *job_list*.

job_list
: An array of type **prf_$job_entry_t** containing a list of jobs from the current queue.

status
: The completion status.

SEE ALSO

prf_$edit_job, prf_$get_printers, prf_$get_sites.

NAME

prf_$set_option – set an option in a print request

SYNOPSIS (C)

```
#include <apollo/base.h>
#include <apollo/prf.h>

void prf_$set_option(
        char *option,
        short &option_length,
        num_var_t &string_value,
        char *p_str,
        short &string_length,
        boolean &string_encoding,
        status_$t *status)
```

SYNOPSIS (Pascal)

```
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/prf.ins.pas';

procedure prf_$set_option(
        in option: univ string;
        in option_length: integer;
        in number_value:  num_var_t;
        in string_value: univ string;
        in string_length: integer;
        in string_encoding: boolean;
        out status: status_$t);
```

SYNOPSIS (FORTRAN)

```
%include '/sys/ins/base.ins.ftn'

      integer*2 opt_len
      parameter (opt_len = 120)

      integer*2 option_length, string_length
      integer*4 number_value, status
      character option*(opt_len), string_value*256
      logical string_encoding

      call prf_$set_option(option, option_length, number_value, string_value,
     &                     string_length, string_encoding, status)
```

DESCRIPTION

This call sets the print option specified by *option* in the current print data structure.

option The name of the option to set.

option_length
The number of bytes in *option*.

number_value
The numeric value to set *option* to.

string_value
The string value to set *option* to.

string_length
The number of bytes in *string_value*.

string_encoding
A Boolean argument indicating whether the value of *option* is encoded as a string or number. If **string_encoding** is **true**, the value of *option* is set from *string_value*. If **false**, the value of *option* is set from *number_value*.

status The completion status.

SEE ALSO
prf_$config_file, prf_$inq_option.

NAME

prf_$signal_printer – signal a print job

SYNOPSIS (C)

```
#include <apollo/base.h>
#include <apollo/prf.h>

void prf_$signal_printer(
        prf_$pr_sig_t &signal,
        prf_$name_t *printer_name,
        status_$t *status)
```

SYNOPSIS (Pascal)

```
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/prf.ins.pas';

procedure prf_$signal_printer(
        in signal: prf_$pr_sig_t;
        in printer_name: univ prf_$name_t;
        out status: status_$t);
```

SYNOPSIS (FORTRAN)

```
%include '/sys/ins/base.ins.ftn'

        integer*4 status
        integer*2 signal, length
        character printer_name*34, name*32

        equivalence (name, printer_name(1:32))
        equivalence (length, printer_name(33:34))

        call prf_$signal_printer(signal, printer_name, status)
```

DESCRIPTION

This call sends the signal specified in *signal* to the printer named *printer_name* at the current site.

signal Specify one of the following:

prf_$abort_job
Abort the job.

prf_$suspend
Suspend the job.

prf_$continue
Continue the job.

printer_name
The name of the printer.

status The completion status.

SEE ALSO

prf_$delete_entry, prf_$edit_job, prf_$read_queue.

NAME

prf_$stream_print – print from a stream

SYNOPSIS (C)

```
#include <apollo/base.h>
#include <apollo/prf.h>

void prf_$stream_print(
        stream_$id_t &stream_id,
        char *queue,
        short &queue_length,
        status_$t *status)
```

SYNOPSIS (Pascal)

```
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/prf.ins.pas';

procedure prf_$stream_print(
        in stream_id: stream_$id_t;
        out queue: name_$pname_t;
        out queue_length: integer;
        out status: status_$t);
```

SYNOPSIS (FORTRAN)

```
%include '/sys/ins/base.ins.ftn'

        integer*2 stream_id, queue_length
        integer*4 status
        character queue*1024

        call prf_$stream_print(stream_id, queue, queue_length, status)
```

DESCRIPTION

This call copies data from the stream specified by *stream_id* to a file on the print spool, then queues a print request for the spooled file. When **prf_$stream_print** returns, it supplies the pathname of the resulting print request in *queue*.

stream_id
The ID of the stream to print from. Everything on the stream up to an end-of-file markeer (EOF) is spooled.

queue The pathname of the resulting print request.

queue_length
The number of bytes in *queue*.

status The completion status.

SEE ALSO

prf_$name_print, prf_$queue_file.

proc1

Contents

NAME

intro – the Level 1 Process Manager

SYNOPSIS (C)

#include <apollo/base.h>
#include <apollo/proc1.h>

SYNOPSIS (Pascal)

%include '/sys/ins/base.ins.pas';
%include '/sys/ins/proc1.ins.pas';

SYNOPSIS (FORTRAN)

%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/proc1.ins.ftn'

DESCRIPTION

There is one call in the Level 1 Process Manager interface, **proc1_$get_cput**, which reports the total CPU time used by the calling process.

Constants

proc1_$n_user_processes
The maximum number of user processes per node.

SEE ALSO

time_$intro.

NAME

proc1_$get_cput – get elapsed CPU time

SYNOPSIS (C)

```
#include <apollo/base.h>
#include <apollo/proc1.h>

void proc1_$get_cput(
        time_$clock_t *clock_value)
```

SYNOPSIS (Pascal)

```
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/proc1.ins.pas';

procedure proc1_$get_cput(
        out clock_value: time_$clock_t);
```

SYNOPSIS (FORTRAN)

```
%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/proc1.ins.ftn'

        integer*2 clock_value(3), clockh
        integer*4 clockl

        equivalence (clockh, clock_value(1)), (clockl, clock_value(2))

        call proc1_$get_cput(clock_value)
```

DESCRIPTION

Proc1_$get_cput supplies the elapsed CPU time for the calling process in *clock_value*.

clock_value

The amount of CPU time used by the calling process since its creation. *Clock_value* is a 48-bit integer counter of the number of 4-microsecond system clock periods that have elapsed while the calling process was running in the CPU. It includes time spent by the operating system in servicing the process, but not time the process spent waiting for I/O or suspended.

proc2

Contents

NAME

intro – the Level 2 Process Manager

SYNOPSIS (C)

#include <apollo/base.h>
#include <apollo/proc2.h>

SYNOPSIS (Pascal)

%include '/sys/ins/base.ins.pas';
%include '/sys/ins/proc2.ins.pas';

SYNOPSIS (FORTRAN)

%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/proc2.ins.ftn'

DESCRIPTION

The **proc_$** calls supply information about Level 2 context of processes running on the local node.

Data Types

proc2_$info_t

A record type for passing process information. The following diagram illustrates its format:

15	0
stack_uid	
stack_uid	
stack_uid	
stack_uid	
state	
Not Used	
usr	
Not Used	
upc	
upc	
usp	
usp	
usb	
usb	
cpu_total	
cpu_total	
cpu_total	
Not Used	
priority	
15	0

stack_uid
The user stack UID.

stack_base
The base address of the user stack.

state The process state, a value of type **proc2_$state_t**.

usr The user status register.

upc The user program counter.

usp The user stack pointer.

usb The user stack base pointer.

cpu_total
The cumulative CPU time used.

priority
The process priority.

proc2_$uid_list_t
An array of type **uid_$t** for passing up to **proc1_$n_user_processes** process UIDs.

proc2_$state_t
A set type for describing the state of a user process. It can take any combination of the following predefined values:

proc2_$waiting
The process is waiting.

proc2_$suspended
The process is suspended.

proc2_$susp_pending
The process suspension is pending.

proc2_$bound
The process is bound.

Errors

proc2_$bad_stack_base
Bad stack base.

proc2_$is_current
The completion status of **proc2_$get_info** when passed the UID for the calling process. It does not indicate a failure.

proc2_$not_level_2
The completion status of **proc2_$get_info** when passed the UID for a process that is not a user process.

proc2_$uid_not_found
Process not found.

NAME

proc2_$get_info – get level 2 process information

SYNOPSIS (C)

```
#include <apollo/base.h>
#include <apollo/proc2.h>

void proc2_$get_info(
        uid_$t &process_uid,
        proc2_$info_t *process_info,
        pinteger &buffer_length,
        status_$t *status)
```

SYNOPSIS (Pascal)

```
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/proc2.ins.pas';

procedure proc2_$get_info(
        in process_uid: uid_$t;
        out process_info: univ proc2_$info_t;
        in buffer_length: pinteger;
        out status: status_$t);
```

SYNOPSIS (FORTRAN)

```
%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/proc2.ins.ftn'

        integer*4 process_uid(2), status
        integer*2 buffer_length, process_info(18)

        integer*4 stack_uid(2), stack_base, upc, usp, usb, cpu_total_low
        integer*2 state, usr, cpu_total(3), cpu_total_high, priority

        equivalence (stack_uid, process_info(1))
        equivalence (stack_base, process_info(5))
        equivalence (state, process_info(7))
        equivalence (usr, process_info(8))
        equivalence (upc, process_info(9))
        equivalence (usp, process_info(11))
        equivalence (usb, process_info(13))
        equivalence (cpu_total, process_info(15))
        equivalence (priority, process_info(18))
        equivalence (cpu_total_high, cpu_total(1))
        equivalence (cpu_total_low, cpu_total(2))

        call proc2_$get_info(process_uid, process_info, buffer_length, status)
```

DESCRIPTION

Proc2_$get_info supplies information about the local process specified by *process_uid*.

process_uid
: The UID of a process.

process_info
: Information about the process specified in *process_uid*. If *process_uid* is the UID of the calling process, **proc_$get_info** supplies only the stack UID and stack base pointer in *process_info*.

buffer_length
: The number of bytes allocated to receive *process_info*. **Proc2_$get_info** will not write more than *buffer_length* bytes of information into *process_info*.

status
: The completion status. In addition to **status_$ok**, **proc2_$get_info** can return successfully with a *status* of **proc2_$is_current**, indicating that the UID passed in *process_uid* is for the calling process and that, therefore, the only valid information supplied in *process_info* is the stack UID and the stack base pointer. If *process_uid* does not correspond to a process running on the local node, **proc2_$get_info** fails with a *status* of **proc2_$uid_not_found**.

NOTES

Proc2_$who_am_i and **proc2_$list** supply process UIDs.

SEE ALSO

proc1_$cpu_time.

NAME

proc2_$list – list level 2 process UIDs

SYNOPSIS (C)

```
#include <apollo/base.h>
#include <apollo/proc2.h>

void proc2_$list(
        uid_$t *uid_list,
        pinteger &max_num_uids,
        pinteger *num_uids)
```

SYNOPSIS (Pascal)

```
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/proc2.ins.pas';

procedure proc2_$list(
        out uid_list: univ proc2_$uid_list_t;
        in max_num_uids: pinteger;
        out num_uids: pinteger);
```

SYNOPSIS (FORTRAN)

```
%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/proc2.ins.ftn'

      integer*4 uid_list(48)
      integer*2 max_num_uids, num_uids

      call proc2_$list(uid_list, max_num_uids, num_uids)
```

DESCRIPTION

Proc2_$list supplies a list of up to *max_num_uids* level 2 processes (user processes) running on the local node in *uid_list*.

uid_list A list of the up to *max_num_uids* UIDs for level 2 processes running on the local node.

max_num_uids
The maximum number of UIDs that **proc2_$list** should supply. **Proc2_$list** will not write more than *max_num_uids* UIDs into *uid_list*.

num_uids
The number of active level 2 processes on the local node. If *num_uids* is greater than *max_num_uids*, then only a partial list was written into *uid_list*.

SEE ALSO

proc2_$who_am_i.

NAME

proc2_$who_am_i – get the UID of the calling process

SYNOPSIS (C)

```
#include <apollo/base.h>
#include <apollo/proc2.h>

void proc2_$who_am_i(
      uid_$t *my_uid)
```

SYNOPSIS (Pascal)

```
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/proc2.ins.pas';

procedure proc2_$who_am_i(
      out my_uid: uid_$t);
```

SYNOPSIS (FORTRAN)

```
%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/proc2.ins.ftn'

      integer*4 my_uid(2)

      call proc2_$who_am_i(my_uid)
```

DESCRIPTION

Proc2_$who_am_i supplies the UID of the calling process in *my_uid*.

my_uid The UID of the calling process.

SEE ALSO

proc2_$list.

rws

Contents

NAME

intro – dynamic storage allocation

SYNOPSIS (C)

#include <apollo/base.h>
#include <apollo/rws.h>

SYNOPSIS (Pascal)

%include '/sys/ins/base.ins.pas';
%include '/sys/ins/rws.ins.pas';

SYNOPSIS (FORTRAN)

%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/rws.ins.ftn'

DESCRIPTION

The **rws_$** calls allow programs to allocate storage dynamically. There are two types of storage supported by the system. ''Heap storage'' can be allocated and released by user processes, whereas ''read/write storage'' can only be allocated by a user process and is released only when the allocating process terminates. Heap storage allows more control over storage, but requires slightly more system overhead to maintain.

Data Types

rws_$pool_t

An enumerated type for specifying the type of storage to allocate. It can assume one of the following values:

rws_$std_pool

Storage allocated from the standard pool is private to the allocating process and is released during a UNIX exec(2) call. A pointer to storage allocated from the standard pool is not valid in another process.

rws_$stream_tm_pool

Storage allocated from the stream pool is private to the allocating process and is retained across a UNIX exec(2) call. A pointer to storage allocated from the standard pool is not valid in another process.

rws_$global_pool

Storage allocated from the global pool is available to all processes running on a single node. A pointer to storage allocated from the global pool may be used by all local processes to access a common storage area.

Errors

rws_$bad_free

Attempted to free storage allocated from a different pool.

rws_$level_failure

Program level information was corrupted by a user process.

rws_$non_existent_pool
Invalid pool specified.

rws_$no_space
Not enough address space or disk space.

rws_$not_heap_entry
Argument to **rws_$release_heap** did not refer to storage allocated with **rws_$alloc_heap**.

rws_$scribbled_over
Heap process information was corrupted by a user process.

rws_$wrong_level
Attempted to release standard or stream pool storage that was allocated by a program at a lower program level.

NAME

rws_$alloc – allocate storage

SYNOPSIS (C)

```
#include <apollo/base.h>
#include <apollo/rws.h>

void rws_$alloc(
        long &storage_size,
        void **storage_ptr)
```

SYNOPSIS (Pascal)

```
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/rws.ins.pas';

procedure rws_$alloc(
        in storage_size: integer32;
        out storage_ptr: univ_ptr);
```

SYNOPSIS (FORTRAN)

```
%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/rws.ins.ftn'

        integer*4 nwords
        parameter (nwords = 5000)

        integer*4 storage_size, storage_ptr
        integer*2 dummy(nwords)
        pointer /storage_ptr/ dummy

        call rws_$alloc(storage_size, storage_ptr)
```

DESCRIPTION

Rws_$alloc allocates *storage_size* bytes of storage to the calling process and supplies a pointer to it in *storage_ptr*.

All storage allocated with **rws_$alloc** remains allocated until process termination. It cannot be released back to the system.

storage_size
: The number of bytes of storage to allocate.

storage_ptr
: The address of the new storage space. If *storage_ptr* is **NULL**, **nil**, or 0, then **rws_$alloc** could not allocate the requested storage.

NOTES

Rws_$alloc_heap_pool and **rws_$alloc_rw_pool** provide more control over storage allocation than **rws_$alloc**.

NAME

rws_$alloc_heap_pool – allocate heap storage

SYNOPSIS (C)

```
#include <apollo/base.h>
#include <apollo/rws.h>

void *rws_$alloc_heap_pool(
        rws_$pool_t &alloc_pool,
        long &storage_size)
```

SYNOPSIS (Pascal)

```
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/rws.ins.pas';

function rws_$alloc_heap_pool(
        in alloc_pool: rws_$pool_t;
        in storage_size: rws_$word_aligned_long): univ_ptr;
```

SYNOPSIS (FORTRAN)

```
%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/rws.ins.ftn'

        integer*4 nwords
        parameter (nwords = 5000)

        integer*4 storage_size, storage_ptr
        integer*2 alloc_pool, dummy(nwords)
        pointer /storage_ptr/ dummy

        storage_ptr = rws_$alloc_heap_pool(alloc_pool, storage_size)
```

DESCRIPTION

Rws_$alloc_heap_pool allocates *storage_size* bytes of heap storage from the pool specified by *alloc_pool* and returns a pointer to it. It returns **NULL**, **nil**, or 0, when it cannot allocate the requested storage.

alloc_pool

The pool from which the storage will be allocated. It can assume one of the following values:

rws_$std_pool

Storage allocated from the standard pool is private to the allocating process and is released during a UNIX exec(2) call. A pointer to storage allocated from the standard pool is not valid in another process.

rws_$stream_tm_pool

Storage allocated from the stream pool is private to the allocating process and is retained across a UNIX exec(2) call. A pointer to storage allocated from the standard pool is not valid in another process.

rws_$global_pool
Storage allocated from the global pool is available to all processes running on a single node. A pointer to storage allocated from the global pool may be used by all local processes to access a common storage area.

storage_size
The number of bytes of storage needed.

NOTES

Unlike storage allocated with **rws_$alloc_rw_pool**, storage allocated with **rws_$alloc_heap_pool** can be released back to the system with **rws_$release_heap_pool**, though the storage requires slightly more overhead.

SEE ALSO

rws_$alloc.

NAME

rws_$alloc_rw_pool – allocate read/write storage from a pool

SYNOPSIS (C)

```
#include <apollo/base.h>
#include <apollo/rws.h>

void *rws_$alloc_rw_pool(
        rws_$pool_t &alloc_pool,
        long &storage_size)
```

SYNOPSIS (Pascal)

```
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/rws.ins.pas';

function rws_$alloc_rw_pool(
        in alloc_pool: rws_$pool_t;
        in storage_size: rws_$word_aligned_long): univ_ptr;
```

SYNOPSIS (FORTRAN)

```
%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/rws.ins.ftn'

        integer*4 nwords
        parameter (nwords = 5000)

        integer*4 storage_size, storage_ptr
        integer*2 alloc_pool, dummy(nwords)
        pointer /storage_ptr/ dummy

        storage_ptr = rws_$alloc_rw_pool(alloc_pool, storage_size)
```

DESCRIPTION

Rws_$alloc_rw_pool allocates *storage_size* bytes of read/write storage from the pool specified by *alloc_pool* and returns a pointer to it. It returns **NULL**, **nil**, or 0, when it cannot allocate the requested storage.

alloc_pool

The pool from which the storage will be allocated.

rws_$std_pool

Storage allocated from the standard pool is private to the allocating process and is released during a UNIX exec(2) call. A pointer to storage allocated from the standard pool is not valid in another process.

rws_$stream_tm_pool

Storage allocated from the stream pool is private to the allocating process and is retained across a UNIX exec(2) call. A pointer to storage allocated from the standard pool is not valid in another process.

rws_$global_pool
Storage allocated from the global pool is available to all processes running on a single node. A pointer to storage allocated from the global pool may be used by all local processes to access a common storage area.

storage_size
The number of bytes of storage needed.

NOTES
Unlike storage allocated with **rws_$alloc_heap_pool** storage allocated with **rws_$alloc_rw_pool** cannot be released back to the system.

SEE ALSO
rws_$alloc.

NAME

rws_$release_heap_pool – release heap storage

SYNOPSIS (C)

```
#include <apollo/base.h>
#include <apollo/rws.h>

void rws_$release_heap_pool(
        void *&storage_ptr,
        rws_$pool_t &alloc_pool,
        status_$t *status)
```

SYNOPSIS (Pascal)

```
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/rws.ins.pas';

procedure rws_$release_heap_pool(
        in storage_ptr: univ_ptr;
        in alloc_pool: rws_$pool_t;
        out status: status_$t);
```

SYNOPSIS (FORTRAN)

```
%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/rws.ins.ftn'

          integer*4 nwords
          parameter (nwords = 5000)

          integer*4 storage_ptr, status
          integer*2 alloc_pool, dummy(nwords)
          pointer /storage_ptr/ dummy

          call rws_$release_heap_pool(storage_ptr, alloc_pool, status)
```

DESCRIPTION

Rws_$release_heap_pool releases the storage at *storage_ptr* back to the system.

storage_ptr

A pointer to storage allocated from the heap with **rws_$alloc_heap_pool**.

alloc_pool

The pool from which the storage was allocated. Choose one of the following values:

rws_$std_pool

Storage allocated from the standard pool is private to the allocating process and is released during a UNIX exec(2) call. A pointer to storage allocated from the standard pool is not valid in another process.

rws_$stream_tm_pool
Storage allocated from the stream pool is private to the allocating process and is retained across a UNIX exec(2) call. A pointer to storage allocated from the standard pool is not valid in another process.

rws_$global_pool
Storage allocated from the global pool is available to all processes running on a single node. A pointer to storage allocated from the global pool may be used by all local processes to access a common storage area.

status The completion status.

NOTES

Unlike storage allocated with **rws_$alloc_rw_pool**, storage allocated with **rws_$alloc_heap_pool** can be released back to the system with **rws_$release_heap_pool**.

sio

Contents

NAME

intro – controlling serial I/O lines

SYNOPSIS (C)

#include <apollo/base.h>
#include <apollo/sio.h>

SYNOPSIS (Pascal)

%include '/sys/ins/base.ins.pas';
%include '/sys/ins/sio.ins.pas';

SYNOPSIS (FORTRAN)

%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/sio.ins.ftn'

DESCRIPTION

The calls are the program interface to the serial I/O (SIO) lines. They can control and monitor the state of an RS-232 port, and modify the stream connections to it. The file system interface to the SIO lines are named /dev/sio*n* where *n* is some integer.

Constants

sio_$50 A baud rate value for the **sio_$speed** and **sio_$speed_force** options.

sio_$75 A baud rate value for the **sio_$speed** and **sio_$speed_force** options.

sio_$110
A baud rate value for the **sio_$speed** and **sio_$speed_force** options.

sio_$134
A baud rate value for the **sio_$speed** and **sio_$speed_force** options.

sio_$150
A baud rate value for the **sio_$speed** and **sio_$speed_force** options.

sio_$300
A baud rate value for the **sio_$speed** and **sio_$speed_force** options.

sio_$600
A baud rate value for the **sio_$speed** and **sio_$speed_force** options.

sio_$1200
A baud rate value for the **sio_$speed** and **sio_$speed_force** options.

sio_$2000
A baud rate value for the **sio_$speed** and **sio_$speed_force** options.

sio_$2400
A baud rate value for the **sio_$speed** and **sio_$speed_force** options.

sio_$3600
A baud rate value for the **sio_$speed** and **sio_$speed_force** options.

sio_$4800
A baud rate value for the **sio_$speed** and **sio_$speed_force** options.

sio_$7200
A baud rate value for the **sio_$speed** and **sio_$speed_force** options.

sio_$9600
A baud rate value for the **sio_$speed** and **sio_$speed_force** options.

sio_$19200
A baud rate value for the **sio_$speed** and **sio_$speed_force** options.

sio_$even_parity
A possible value for the **sio_$parity** option, denoting even parity.

sio_$max_line
The highest SIO line number supported.

sio_$no_parity
A possible value for the **sio_$parity** option, denoting no parity.

sio_$odd_parity
A possible value for the **sio_$parity** option, denoting odd parity.

sio_$stop_1
A stop bit value for the **sio_$stop_bits** option.

sio_$stop_1_point_5
A stop bit value for the **sio_$stop_bits** option.

sio_$stop_2
A stop bit value for the **sio_$stop_bits** option.

sio_$5bpc
A possible value for the **sio_$bits_per_char** option, denoting five bits per character.

sio_$6bpc
A possible value for the **sio_$bits_per_char** option, denoting six bits per character.

sio_$7bpc
A possible value for the **sio_$bits_per_char** option, denoting seven bits per character.

sio_$8bpc
A possible value for the **sio_$bits_per_char** option, denoting eight bits per character.

Data Types

sio_$err_enables_t
A small set of enabled SIO errors. It can assume a combination of the following values:

sio_$check_parity
Check for received parity errors.

sio_$check_framing
Check for received framing errors.

sio_$check_dcd_change
Check for when the Data Carrier Detect (DCD) line changes state.

sio_$check_cts_change
Check for when the Clear To Send (CTS) line changes state.

sio_$err_possibilities_t
A small set of errors to report during read calls on an SIO line. It can assume a combination of the following values:

sio_$check_parity
Report parity errors.

sio_$check_framing
Report framing errors. Framing errors are reported by default.

sio_$check_dcd_change
Report DCD line state changes.

sio_$check_cts_change
Report CTS line state changes.

sio_$line_t
An SIO line number between 0 and **sio_$max_line**.

sio_$opt_t
An enumerated type for specifying an SIO option. It takes one of the following values:

sio_$bits_per_char
The number of bits per character. This option can have one of the following values:

sio_$5bpc
Five bits per character.

sio_$6bpc
Six bits per character.

sio_$7bpc
Seven bits per character.

sio_$8bpc
Eight bits per character.

sio_$bp_enable
This option is a Boolean value that enables or disables interpretation of control codes from a bit pad. If **true**, bit pad control code interpretation is enabled. If **false**, bit pad control code interpretation is disabled. The default value is **false**.

sio_$cts This option is a Boolean value that reflects the state of the Clear to Send (CTS) line. The value of this option cannot be changed with **sio_$control**.

sio_$cts_enable
This option is a Boolean value that specifies whether the CTS line will be used to inhibit transmission. If **true**, then transmission is inhibited when CTS is **false**. If **false**, the CTS line does not inhibit transmission.

sio_$dcd
This option is a Boolean value that reflects the state of the Data Carrier Detect (DCD) line. The value of this option cannot be changed with **sio_$control**.

sio_$dcd_enable
This option is a Boolean value that specifies whether a DCD line transition will generate a **fault_$stop** fault. If **true**, then a DCD line transition from **true** to **false** generates a fault. If **false**, a DCD line transition does not generate a fault.

sio_$drain_out
This option is a Boolean value that specifies whether write calls on the SIO line should wait until all characters in the output buffer are transmitted before returning. If **true**, the write call waits. If **false**, the write call doesn't wait. The default is **false**.

sio_$dtr
This option is a Boolean value that sets the state of the Data Terminal Ready (DTR) line.

sio_$eofchr
The value of this option is the end-of-file (EOF) character. It takes a character value, and the default is CTRL/Z.

sio_$erase
The value of this option is the erase character. It takes a character value, and the default is CTRL/H.

sio_$err_enable
The value of this option is a small set of type **sio_$err_possibilities_t** that defines the errors that can be reported by read calls on an SIO line. It can be any combination of the following values:

sio_$check_parity
Report parity errors.

sio_$check_framing
Report framing errors. Framing errors are reported by default.

sio_$check_dcd_change
Report DCD line state changes.

sio_$check_cts_change
Report CTS line state changes.

sio_$flush_in
This option is a Boolean value that specifies whether to flush the input buffer of an SIO line. If **true**, the input buffer is flushed. If **false**, the input buffer is not flushed. The default is **false**.

sio_$flush_out
This option is a Boolean value that specifies whether to flush the output buffer of an SIO line. If **true**, the output buffer is flushed. If **false**, the output buffer is not flushed. The default is **false**.

sio_$host_sync
This option is a Boolean value that enables or disables host synchronization. If **true**, the node sends XOFF (CTRL/S) when its input buffer is full, and sends XON (CTRL/Q) when it is again ready to receive data. If **false**, the node does not use XON/XOFF synchronization. The default is **true**.

sio_$hup_close
This option is a Boolean value that specifies whether the SIO line should ''hang up'' when the last stream on it is closed. If **true**, the node drops the Data Terminal Ready (DTR) line for approximately three-fourths of a second when the last stream on it is closed. If **false**, the node does not hang up.

sio_$input_sync
This option is a Boolean value that enables or disables input synchronization. If **true**, the node honors incoming XON and XOFF requests. If **false**, the node ignores incoming XON/XOFF signals. The default is **false**.

sio_$int_enable
This option is a Boolean value that enables or disables interrupts from the calling process. If **true**, the SIO line honors interrupts from the calling process. If **false**, the SIO line ignores interrupts from the calling process.

sio_$intchr
The value of this option is the interrupt character. It takes a character value, and the default is CTRL/C.

sio_$kill
The value of this option is the kill character. It takes a character value, and the default is CTRL/X.

sio_$line
This value of this option is the SIO line number, an integer from 0 to **sio_$max_line**. The value of this option cannot be changed with

sio_$control.

sio_$nlc_delay
The value of this option is the time delay in milliseconds following transmission of a line feed character, to allow for carriage motion or scrolling time, and so on. The default is 0.

sio_$no_NL
This option is a Boolean value that specifies whether a newline character is transmitted as a carriage-return line-feed sequence or as a raw newline character. If **true**, newlines are transmitted as is. If **false**, newlines are transmitted as carriage-return line-feed sequences. The default is **false**.

sio_$no_echo
This option is a Boolean value that specifies whether input characters are echoed back in the output. If **true**, input characters are not echoed. If **false**, input characters are echoed back to the sender. The default is **false**.

sio_$parity
The value of this option is the parity used by the SIO line. It can take any one of the following values:

sio_$even_parity
Even parity.

sio_$no_parity
No parity.

sio_$odd_parity
Odd parity.

If the parity is even or odd, then one bit is added to each character to enforce the chosen parity. If there are fewer than eight bits per character, the parity bit is delivered with the data in raw mode, but is stripped in cooked mode.

sio_$quit_enable
This option is a Boolean value that specifies whether the node will respond to a quit character or break condition by interrupting the calling process. If **true**, a quit or break generates a quit fault. If **false**, a quit or break is ignored. The default is **false**. (In raw mode, break and quit are the only ways to generate a quit fault.)

sio_$quitchr
The value of this option is the quit character. It takes a character value, and the default is CTRL/].

sio_$raw
This option is a Boolean value that determines whether the line is in

raw or cooked mode. If **true**, the line is in raw mode and each read returns all bytes received since the last call without any interpretation. If **false**, the line is in cooked mode and control codes are interpreted. The default value is **false**.

The input buffer is flushed whenever **sio_$raw** changes value.

sio_$raw_nl
This option is a Boolean value that specifies whether a newline character is transmitted as a carriage-return line-feed sequence or as a raw newline character when the line is in raw mode. If **true**, newlines are transmitted as is while in raw mode. If **false**, newlines are transmitted as carriage-return line-feed sequences while in raw mode. The default is **false**.

sio_$rts This option is a Boolean value that determines the state of the Request To Send (RTS) line. The default is **true**.

sio_$rts_enable
This option is a Boolean value that enables or disables the RTS line. If **true**, the RTS line is used with the CTS line for data flow control. For flow control to work, the CTS line must also be enabled. If **false**, no RTS/CTS flow control is provided. The default is **false**.

sio_$send_break
The value of this option is the duration of a break condition on the SIO line, in milliseconds. A reasonable value for the **sio_$send_break** option is 200. A value other than 0 causes a break condition.

sio_$speed
The value of this option is the baud rate of the SIO line. It can have any of the following values: **sio_$50**, **sio_$75**, **sio_$110**, **sio_$134**, **sio_$150**, **sio_$300**, **sio_$600**, **sio_$1200**, **sio_$2000**, **sio_$2400**, **sio_$3600**, **sio_$4800**, **sio_$7200**, **sio_$9600**, or **sio_$19200**. The default is **sio_$9600**.

sio_$speed_force
The value of this option is the baud rate of a partnered SIO line. The baud rate is forced to the value of this option, even if it is incompatible with the rate of the line's partner. If the value of **sio_$speed_force** is incompatible with the speed of the partnered line, the partnered line is forced to the default speed of **sio_$9600**. This option can have any of the following values: **sio_$50**, **sio_$75**, **sio_$110**, **sio_$134**, **sio_$150**, **sio_$300**, **sio_$600**, **sio_$1200**, **sio_$2000**, **sio_$2400**, **sio_$3600**, **sio_$4800**, **sio_$7200**, **sio_$9600**, or **sio_$19200**.

sio_$stop_bits
The value of this option is the number of stop bits used by the SIO line. It can have any of the following values: **sio_$stop_1**,

sio_$stop_1_point_5, or **sio_$stop_2**. The default is **sio_$stop_1**.

sio_$susp_enable

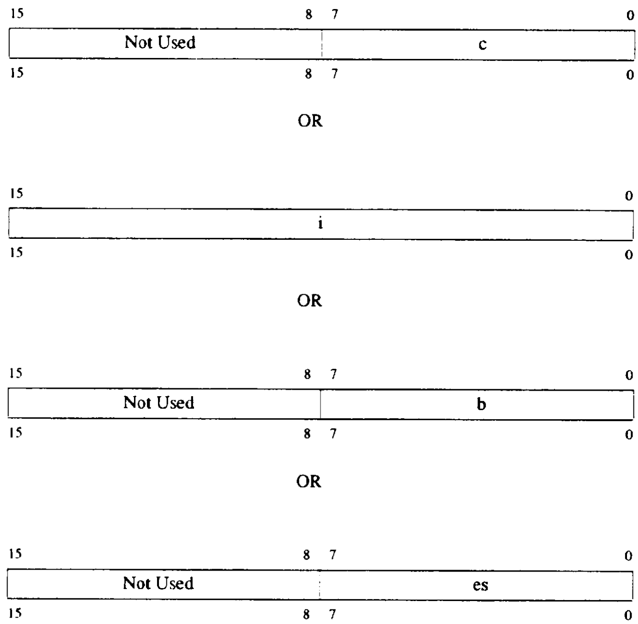
This option is a Boolean value that enables or disables suspend signals from the calling process. If **true**, the SIO line honors suspend signals (faults). If **false**, the SIO line ignores suspend signals (faults).

sio_$suspchr

The value of this option is the suspend character. It takes a character value, and the default is CTRL/P.

sio_$value_t

A variant record type for passing the values of SIO options. The diagram below illustrates the sio_$value_t data type:



c A character value.

i A 2-byte integer value.

b A Boolean value.

es A set of type **sio_$err_enables_t**.

Errors

sio_$bad_option
The option passed is not a valid SIO option.

sio_$illegal_strid
The stream ID passed in the **sio_$** call is not open on an SIO line.

sio_$incompatible_speed
Attempted to set the speed for a partnered line that is incompatible with the speed of the line's partner.

NAME

sio_$control – set serial line options

SYNOPSIS (C)

```
#include <apollo/base.h>
#include <apollo/sio.h>

void sio_$control(
        stream_$id_t &stream_id,
        sio_$opt_t &sio_option,
        sio_$value_t &option_value,
        status_$t *status)
```

SYNOPSIS (Pascal)

```
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/sio.ins.pas';

procedure sio_$control(
        in stream_id: stream_$id_t;
        in sio_option: sio_$opt_t;
        in option_value: univ sio_$value_t;
        out status: status_$t);
```

SYNOPSIS (FORTRAN)

```
%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/sio.ins.ftn'

        integer*2 nchar
        parameter (nchar = 256)

        integer*4 status, option_value
        integer*2 stream_id, sio_option, i2_value
        logical l_value
        character c_value*(nchar)

        equivalence (i2_value, option_value)
        equivalence (l_value, option_value)
        equivalence (c_value, option_value)

        call sio_$control(stream_id, sio_option, option_value, status)
```

DESCRIPTION

Sio_$control sets the option specified by *sio_option* to the value passed in *option_value* for the SIO line open on *stream_id*.

The hardware configuration for some machine types is such that certain SIO lines are "partnered" with each other. The following is a list of machine types and the SIO lines that are partnered on them.

Serial I/O Lines	
Machine Type	**Partnered Lines**
DN300	1,2
DSP80	1,2
DN460 DN660	0,1 2,3
DN550	1,2
DN3000	No Partners
DN4000	No Partners

A characteristic of partnered lines is that some baud rates are incompatible. The following lists show the baud rates that are incompatible for partnered lines.

Incompatible Baud Rates	
A Rates	**B Rates**
sio_$50 **sio_$7200**	**sio_$75** **sio_$150** **sio_$2000** **sio_$19200**

If one partner is set to a baud rate in the A list, an attempt to set the other partner to a baud rate in the B list (using the **sio_$speed** option) will result in the error **sio_$incompatible_speed**. The same is true for the reverse (having a partnered line set to a rate in the B list and attempting to set its partner to a rate in the A list). Speeds that don't appear in the table are compatible with any speed.

Sio_$control can force a line's speed to one that is incompatible with its partner by using the **sio_$speed_force** option; however, it will also change the speed of the partnered line to **sio_$9600**, which is compatible with any speed.

stream_id
: The stream ID of a stream open on a serial line.

sio_option
: An SIO line option. (The value of **sio_$cts**, **sio_$dcd**, and **sio_$line** cannot be changed with **sio_$control**. They are only valid when calling **sio_$inquire**.) Specify only one of the following values:

  sio_$bits_per_char
  : The number of bits per character. This option can have one of the following values:

    sio_$5bpc
    : Five bits per character.

sio_$6bpc
Six bits per character.

sio_$7bpc
Seven bits per character.

sio_$8bpc
Eight bits per character.

sio_$bp_enable
This option is a Boolean value that enables or disables interpretation of control codes from a bit pad. If **true**, bit pad control code interpretation is enabled. If **false**, bit pad control code interpretation is disabled. The default value is **false**.

sio_$cts This option is a Boolean value that reflects the state of the Clear to Send (CTS) line. The value of this option cannot be changed with **sio_$control**.

sio_$cts_enable
This option is a Boolean value that specifies whether the CTS line will be used to inhibit transmission. If **true**, then transmission is inhibited when CTS is **false**. If **false**, the CTS line does not inhibit transmission.

sio_$dcd
This option is a Boolean value that reflects the state of the Data Carrier Detect (DCD) line. The value of this option cannot be changed with **sio_$control**.

sio_$dcd_enable
This option is a Boolean value that specifies whether a DCD line transition will generate a **fault_$stop** fault. If **true**, then a DCD line transition from **true** to **false** generates a fault. If **false**, a DCD line transition does not generate a fault.

sio_$drain_out
This option is a Boolean value that specifies whether write calls on the SIO line should wait until all characters in the output buffer are transmitted before returning. If **true**, the write call waits. If **false**, the write call doesn't wait. The default is **false**.

sio_$dtr
This option is a Boolean value that sets the state of the Data Terminal Ready (DTR) line.

sio_$eofchr
The value of this option is the end-of-file (EOF) character. It takes a character value, and the default is CTRL/Z.

sio_$erase
The value of this option is the erase character. It takes a character

value, and the default is CTRL/H.

sio_$err_enable
The value of this option is a small set of type **sio_$err_possibilities_t** that defines the errors that can be reported by read calls on an SIO line. It can be any combination of the following values:

sio_$check_parity
Report parity errors.

sio_$check_framing
Report framing errors. Framing errors are reported by default.

sio_$check_dcd_change
Report DCD line state changes.

sio_$check_cts_change
Report CTS line state changes.

sio_$flush_in
This option is a Boolean value that specifies whether to flush the input buffer of an SIO line. If **true**, the input buffer is flushed. If **false**, the input buffer is not flushed. The default is **false**.

sio_$flush_out
This option is a Boolean value that specifies whether to flush the output buffer of an SIO line. If **true**, the output buffer is flushed. If **false**, the output buffer is not flushed. The default is false.

sio_$host_sync
This option is a Boolean value that enables or disables host synchronization. If **true**, the node sends XOFF (CTRL/S) when its input buffer is full, and sends XON (CTRL/Q) when it is again ready to receive data. If **false**, the node does not use XON/XOFF synchronization. The default is **true**.

sio_$hup_close
This option is a Boolean value that specifies whether the SIO line should "hang up" when the last stream on it is closed. If **true**, the node drops the Data Terminal Ready (DTR) line for approximately three-fourths of a second when the last stream on it is closed. If **false**, the node does not hang up.

sio_$input_sync
This option is a Boolean value that enables or disables input synchronization. If **true**, the node honors incoming XON and XOFF requests. If **false**, the node ignores incoming XON/XOFF signals. The default is **false**.

sio_$int_enable
This option is a Boolean value that enables or disables interrupts from

the calling process. If **true**, the SIO line honors interrupts from the calling process. If **false**, the SIO line ignores interrupts from the calling process.

sio_$intchr
The value of this option is the interrupt character. It takes a character value, and the default is CTRL/C.

sio_$kill
The value of this option is the kill character. It takes a character value, and the default is CTRL/X.

sio_$line
This value of this option is the SIO line number, an integer from 0 to **sio_$max_line**. The value of this option cannot be changed with **sio_$control**.

sio_$nlc_delay
The value of this option is the time delay in milliseconds following transmission of a line feed character, to allow for carriage motion or scrolling time, and so on. The default is 0.

sio_$no_NL
This option is a Boolean value that specifies whether a newline character is transmitted as a carriage-return line-feed sequence or as a raw newline character. If **true**, newlines are transmitted as is. If **false**, newlines are transmitted as carriage-return line-feed sequences. The default is **false**.

sio_$no_echo
This option is a Boolean value that specifies whether input characters are echoed back in the output. If **true**, input characters are not echoed. If **false**, input characters are echoed back to the sender. The default is **false**.

sio_$parity
The value of this option is the parity used by the SIO line. It can take any one of the following values:

sio_$even_parity
Even parity.

sio_$no_parity
No parity.

sio_$odd_parity
Odd parity.

If the parity is even or odd, then one bit is added to each character to enforce the chosen parity. If there are fewer than eight bits per character, the parity bit is delivered with the data in raw mode, but is stripped in cooked mode.

sio_$quit_enable
This option is a Boolean value that specifies whether the node will respond to a quit character or break condition by interrupting the calling process. If **true**, a quit or break generates a quit fault. If **false**, a quit or break is ignored. The default is **false**. (In raw mode, break and quit are the only ways to generate a quit fault.)

sio_$quitchr
The value of this option is the quit character. It takes a character value, and the default is CTRL/].

sio_$raw
This option is a Boolean value that determines whether the line is in raw or cooked mode. If **true**, the line is in raw mode and each read returns all bytes received since the last call without any interpretation. If **false**, the line is in cooked mode and control codes are interpreted. The default value is **false**

The input buffer is flushed whenever **sio_$raw** changes value.

sio_$raw_nl
This option is a Boolean value that specifies whether a newline character is transmitted as a carriage-return line-feed sequence or as a raw newline character when the line is in raw mode. If **true**, newlines are transmitted as is while in raw mode. If **false**, newlines are transmitted as carriage-return line-feed sequences while in raw mode. The default is **false**.

sio_$rts This option is a Boolean value that determines the state of the Request To Send (RTS) line. The default is **true**.

sio_$rts_enable
This option is a Boolean value that enables or disables the RTS line. If **true**, the RTS line is used with the CTS line for data flow control. For flow control to work, the CTS line must also be enabled. If **false**, no RTS/CTS flow control is provided. The default is **false**.

sio_$send_break
The value of this option is the duration of a break condition on the SIO line, in milliseconds. A reasonable value for the **sio_$send_break** option is 200. A value other than 0 causes a break condition.

sio_$speed
The value of this option is the baud rate of the SIO line. It can have any of the following values: **sio_$50**, **sio_$75**, **sio_$110**, **sio_$134**, **sio_$150**, **sio_$300**, **sio_$600**, **sio_$1200**, **sio_$2000**, **sio_$2400**, **sio_$3600**, **sio_$4800**, **sio_$7200**, **sio_$9600**, or **sio_$19200**. The default is **sio_$9600**.

sio_$speed_force
The value of this option is the baud rate of a partnered SIO line. The baud rate is forced to the value of this option, even if it is incompatible with the rate of the line's partner. If the value of **sio_$speed_force** is incompatible with the speed of the partnered line, the partnered line is forced to the default speed of **sio_$9600**. This option can have any of the following values: **sio_$50**, **sio_$75**, **sio_$110**, **sio_$134**, **sio_$150**, **sio_$300**, **sio_$600**, **sio_$1200**, **sio_$2000**, **sio_$2400**, **sio_$3600**, **sio_$4800**, **sio_$7200**, **sio_$9600**, or **sio_$19200**.

sio_$stop_bits
The value of this option is the number of stop bits used by the SIO line. It can have any of the following values: **sio_$stop_1**, **sio_$stop_1_point_5**, or **sio_$stop_2**. The default is **sio_$stop_1**.

sio_$susp_enable
This option is a Boolean value that enables or disables suspend signals from the calling process. If **true**, the SIO line honors suspend signals (faults). If **false**, the SIO line ignores suspend signals (faults).

sio_$suspchr
The value of this option is the suspend character. It takes a character value, and the default is CTRL/P.

option_value
The new value for the option specified by *sio_option*.

status The completion status. Possible values are:

sio_$bad_option
The option passed is not a valid SIO option.

sio_$illegal_strid
The stream ID passed in the sio_$ call is not open on an SIO line.

sio_$incompatible_speed
Attempted to set the speed for a partnered line that is incompatible with the speed of the line's partner.

NAME

sio_$inquire – get serial line options

SYNOPSIS (C)

```
#include <apollo/base.h>
#include <apollo/sio.h>

void sio_$inquire(
        stream_$id_t &stream_id,
        sio_$opt_t &sio_option,
        sio_$value_t *option_value,
        status_$t *status)
```

SYNOPSIS (Pascal)

```
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/sio.ins.pas';

procedure sio_$inquire(
        in stream_id: stream_$id_t;
        in sio_option: sio_$opt_t;
        out option_value: univ sio_$value_t;
        out status: status_$t);
```

SYNOPSIS (FORTRAN)

```
%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/sio.ins.ftn'

        integer*2 nchar
        parameter (nchar = 256)

        integer*4 status, option_value
        integer*2 stream_id, sio_option, i2_value
        logical l_value
        character c_value*(nchar)

        equivalence (i2_value, option_value)
        equivalence (l_value, option_value)
        equivalence (c_value, option_value)

        call sio_$inquire(stream_id, sio_option, option_value, status)
```

DESCRIPTION

Sio_$inquire supplies the value of the option specified by *sio_option* for the SIO line open on *stream_id*.

stream_id

The stream ID of a stream open on a serial line.

sio_option

An SIO line option. Specify only one of the following values:

sio_$bits_per_char

The number of bits per character. This option can have one of the following values:

sio_$5bpc

Five bits per character.

sio_$6bpc

Six bits per character.

sio_$7bpc

Seven bits per character.

sio_$8bpc

Eight bits per character.

sio_$bp_enable

This option is a Boolean value that enables or disables interpretation of control codes from a bit pad. If **true**, bit pad control code interpretation is enabled. If **false**, bit pad control code interpretation is disabled. The default value is **false**.

sio_$cts This option is a Boolean value that reflects the state of the Clear to Send (CTS) line. The value of this option cannot be changed with **sio_$control**.

sio_$cts_enable

This option is a Boolean value that specifies whether the CTS line will be used to inhibit transmission. If **true**, then transmission is inhibited when CTS is **false**. If **false**, the CTS line does not inhibit transmission.

sio_$dcd

This option is a Boolean value that reflects the state of the Data Carrier Detect (DCD) line. The value of this option cannot be changed with **sio_$control**.

sio_$dcd_enable

This option is a Boolean value that specifies whether a DCD line transition will generate a **fault_$stop** fault. If **true**, then a DCD line transition from **true** to **false** generates a fault. If **false**, a DCD line transition does not generate a fault.

sio_$drain_out

This option is a Boolean value that specifies whether write calls on the SIO line should wait until all characters in the output buffer are transmitted before returning. If **true**, the write call waits. If **false**, the write call doesn't wait. The default is **false**.

sio_$dtr
This option is a Boolean value that sets the state of the Data Terminal Ready (DTR) line.

sio_$eofchr
The value of this option is the end-of-file (EOF) character. It takes a character value, and the default is CTRL/Z.

sio_$erase
The value of this option is the erase character. It takes a character value, and the default is CTRL/H.

sio_$err_enable
The value of this option is a small set of type **sio_$err_possibilities_t** that defines the errors that can be reported by read calls on an SIO line. It can be any combination of the following values:

sio_$check_parity
Report parity errors.

sio_$check_framing
Report framing errors. Framing errors are reported by default.

sio_$check_dcd_change
Report DCD line state changes.

sio_$check_cts_change
Report CTS line state changes.

sio_$flush_in
This option is a Boolean value that specifies whether to flush the input buffer of an SIO line. If **true**, the input buffer is flushed. If **false**, the input buffer is not flushed. The default is **false**.

sio_$flush_out
This option is a Boolean value that specifies whether to flush the output buffer of an SIO line. If **true**, the output buffer is flushed. If **false**, the output buffer is not flushed. The default is **false**.

sio_$host_sync
This option is a Boolean value that enables or disables host synchronization. If **true**, the node sends XOFF (CTRL/S) when its input buffer is full, and sends XON (CTRL/Q) when it is again ready to receive data. If **false**, the node does not use XON/XOFF synchronization. The default is **true**.

sio_$hup_close
This option is a Boolean value that specifies whether the SIO line should ''hang up'' when the last stream on it is closed. If **true**, the node drops the Data Terminal Ready (DTR) line for approximately three-fourths of a second when the last stream on it is closed. If **false**,

the node does not hang up.

sio_$input_sync
This option is a Boolean value that enables or disables input synchronization. If **true**, the node honors incoming XON and XOFF requests. If **false**, the node ignores incoming XON/XOFF signals. The default is **false**.

sio_$int_enable
This option is a Boolean value that enables or disables interrupts from the calling process. If **true**, the SIO line honors interrupts from the calling process. If **false**, the SIO line ignores interrupts from the calling process.

sio_$intchr
The value of this option is the interrupt character. It takes a character value, and the default is CTRL/C.

sio_$kill
The value of this option is the kill character. It takes a character value, and the default is CTRL/X.

sio_$line
This value of this option is the SIO line number, an integer from 0 to **sio_$max_line**. The value of this option cannot be changed with **sio_$control**.

sio_$nlc_delay
The value of this option is the time delay in milliseconds following transmission of a line feed character, to allow for carriage motion or scrolling time, and so on. The default is 0.

sio_$no_NL
This option is a Boolean value that specifies whether a newline character is transmitted as a carriage-return line-feed sequence or as a raw newline character. If **true**, newlines are transmitted as is. If **false**, newlines are transmitted as carriage-return line-feed sequences. The default is **false**.

sio_$no_echo
This option is a Boolean value that specifies whether input characters are echoed back in the output. If **true**, input characters are not echoed. If **false**, input characters are echoed back to the sender. The default is **false**.

sio_$parity
The value of this option is the parity used by the SIO line. It can take any one of the following values:

sio_$even_parity
Even parity.

sio_$no_parity
No parity.

sio_$odd_parity
Odd parity.

If the parity is even or odd, then one bit is added to each character to enforce the chosen parity. If there are fewer than eight bits per character, the parity bit is delivered with the data in raw mode, but is stripped in cooked mode.

sio_$quit_enable
This option is a Boolean value that specifies whether the node will respond to a quit character or break condition by interrupting the calling process. If **true**, a quit or break generates a quit fault. If **false**, a quit or break is ignored. The default is **false**. (In raw mode, break and quit are the only ways to generate a quit fault.)

sio_$quitchr
The value of this option is the quit character. It takes a character value, and the default is CTRL/].

sio_$raw
This option is a Boolean value that determines whether the line is in raw or cooked mode. If **true**, the line is in raw mode and each read returns all bytes received since the last call without any interpretation. If **false**, the line is in cooked mode and control codes are interpreted. The default value is **false**.

The input buffer is flushed whenever **sio_$raw** changes value.

sio_$raw_nl
This option is a Boolean value that specifies whether a newline character is transmitted as a carriage-return line-feed sequence or as a raw newline character when the line is in raw mode. If **true**, newlines are transmitted as is while in raw mode. If **false**, newlines are transmitted as carriage-return line-feed sequences while in raw mode. The default is **false**.

sio_$rts This option is a Boolean value that determines the state of the Request To Send (RTS) line. The default is **true**.

sio_$rts_enable
This option is a Boolean value that enables or disables the RTS line. If **true**, the RTS line is used with the CTS line for data flow control. For flow control to work, the CTS line must also be enabled. If **false**, no RTS/CTS flow control is provided. The default is **false**.

sio_$send_break
The value of this option is the duration of a break condition on the SIO line, in milliseconds. A reasonable value for the **sio_$send_break**

option is 200. A value other than 0 causes a break condition.

sio_$speed
The value of this option is the baud rate of the SIO line. It can have any of the following values: **sio_$50**, **sio_$75**, **sio_$110**, **sio_$134**, **sio_$150**, **sio_$300**, **sio_$600**, **sio_$1200**, **sio_$2000**, **sio_$2400**, **sio_$3600**, **sio_$4800**, **sio_$7200**, **sio_$9600**, or **sio_$19200**. The default is **sio_$9600**.

sio_$speed_force
The value of this option is the baud rate of a partnered SIO line. The baud rate is forced to the value of this option, even if it is incompatible with the rate of the line's partner. If the value of **sio_$speed_force** is incompatible with the speed of the partnered line, the partnered line is forced to the default speed of **sio_$9600**. This option can have any of the following values: **sio_$50**, **sio_$75**, **sio_$110**, **sio_$134**, **sio_$150**, **sio_$300**, **sio_$600**, **sio_$1200**, **sio_$2000**, **sio_$2400**, **sio_$3600**, **sio_$4800**, **sio_$7200**, **sio_$9600**, or **sio_$19200**.

sio_$stop_bits
The value of this option is the number of stop bits used by the SIO line. It can have any of the following values: **sio_$stop_1**, **sio_$stop_1_point_5**, or **sio_$stop_2**. The default is **sio_$stop_1**.

sio_$susp_enable
This option is a Boolean value that enables or disables suspend signals from the calling process. If **true**, the SIO line honors suspend signals (faults). If **false**, the SIO line ignores suspend signals (faults).

sio_$suspchr
The value of this option is the suspend character. It takes a character value, and the default is CTRL/P.

option_value
The value for the option specified by *sio_option*.

status The completion status. Possible values are

sio_$bad_option
The option passed is not a valid SIO option.

sio_$illegal_strid
The stream ID passed in the sio_$ call is not open on an SIO line.

status

Contents

NAME
intro – status reporting types and constants

SYNOPSIS (C)
#include <apollo/base.h>

SYNOPSIS (Pascal)
%include '/sys/ins/base.ins.pas';

SYNOPSIS (FORTRAN)
%include '/sys/ins/base.ins.ftn'

DESCRIPTION
Most system calls supply their completion status in **status_$t** format. The **status_$t** type and the constant **status_$ok** are defined to help evaluate the status information supplied by system calls.

Constants
status_$ok

A constant used to check status. If the **all** instance of a completion status is equal to **status_$ok**, then the system call that supplied it was successful.

Data Types
status_$t

This is a variant record type with two instances. One instance has only one member, named **all**. The other instance has four members, named **fail**, **code**, **mode**, and **subsys**. Variables of the **status_$t** type require four bytes of storage.

all All 32 bits in the status code. If **all** is equal to **status_$ok**, the system call that supplied the status was successful.

fail The field labeled "f" in the diagram is the fail bit. If **fail** is set, the error was not within the scope of the module invoked, but occurred within a lower-level module.

subsys This indicates the subsystem that encountered the error.

mode This indicates the module that encountered the error.

code This is a signed number that identifies the type of error that occurred.

31		16
all		
all		
15		0

OR

31	30 ... 24	23 ... 16
f	subsys	mode
code		
15		0

task

Contents

NAME
intro – the Domain/OS task library

SYNOPSIS (C)
#include <apollo/base.h>
#include <apollo/task.h>

SYNOPSIS (Pascal)
%include "/sys/ins/base.ins.pas"
%include "/sys/ins/task.ins.pas"

DESCRIPTION
The **task_$** calls create and manage a multitasking environment within a single process. In a multitasking environment, multiple threads of execution (tasks) run within a single address space. Because task creation and switching are less expensive than process creation and switching, tasking is a useful way of breaking down any complex operation into separate pieces that run concurrently. It is particularly useful in Remote Procedure Call (RPC) servers that manage multiple remote requests in parallel.

Creating a Task
A process creates a task by calling **task_$create**, which returns a task handle to identify the task. You can obtain the handle of the currently running task by calling **task_$get_handle**. (You can also optionally assign to the task an ASCII-string task name, up to 32 characters, by calling **task_$set_name**.)

The first task in a process, the one that first calls **task_$create**, is known as the Distinguished Task (DT). After the first call to **task_$create**, tasking is enabled in the process, and any task can create a new task.

Time Slicing and Priority Levels
Tasks are able to run concurrently because of a scheduling scheme based on time slicing and priority levels. An initial priority level (from 1 to 9, where 9 is the highest) is assigned to a task when it is created. The Task Scheduler uses the priority level to determine which task (of the tasks that are ready) to run when one of the following conditions occurs:

- A time-slice interval ends.
- A task voluntarily yields the processor by calling **task_$yield**.
- A task blocks in user space by calling one of the following: **ec2_$wait**, **ec2_$wait_svc**, **time_$wait**, **msg_$wait**, or **mutex_$lock**. (When a task blocks in the operating system the entire process is blocked, and another task is not started until the process returns to user space.)

As a task runs without blocking, its priority level is gradually lowered until it reaches the lowest priority level. It then runs at the lowest priority level until it blocks. Each time a task wakes after being blocked, the Task Scheduler reassigns the initial priority level to the task. The Task Scheduler algorithm favors tasks that run a short time between blocking, but ensures that all tasks eventually run: no task can lock out other tasks forever.

A task exists until one of the following events occurs:

- The task routine returns.
- The task routine calls **task_$exit**.
- An unhandled fault occurs.
- The task's lifetime, as defined by **task_$create**, expires.

Completion Eventcount

If a task is created with the **task_$intend_to_wait** option, the system maintains an eventcount that advances when the task completes. A task can wait for the completion of another by calling **task_$get_ec** to get the eventcount, and then using **ec2_$wait** to wait for it to advance. If a task is created with a completion eventcount, then some other task must call **task_$release** to release the task after it completes. The **task_$release** call will also supply the completion status and output value of the completed task to the caller.

Fault Inhibiting

Inhibiting faults in a tasking process inhibits all asynchronous signals, including those used for task switching. To inhibit all signals, including those that cause task switching, use **pfm_$inhibit** and **pfm_$enable**. For example, a tasking program could use **pfm_$inhibit** and **pfm_$enable** pairs to prevent other tasks from running during critical sections of code.

Fault inhibiting is always per task: one task can inhibit faults while another doesn't. If a fault is pending when a non-inhibited task begins to run, the fault is signaled then. Also, **pfm_$inhibit** does not prevent a task switch from occurring when a task blocks or voluntarily yields the processor.

Fault Handlers

Because fault handlers are established on a per-process basis, the same fault handler executes regardless of the task that established it or the task that is running when the fault occurs. A handled fault is not propagated to the DT.

Clean-Up Handlers

Clean-up handlers are established per task; that is, a clean-up handler is invoked only in the task that established it. A synchronous fault is delivered to the clean-up handler in the task in which the synchronous fault occurs. An asynchronous fault is delivered to the DT the next time it runs. The Task Manager sets up a default clean-up handler at the base of a task's stack when the task is created.

A task can exit with cleanup either by returning or by calling **task_$exit**. A task can also cause another task to exit cleanly by calling **task_$signal**. As a last resort, a task can destroy another task without cleanup by calling **task_$blast**.

Programming Restrictions

Because tasks are time-sliced, an application using tasking must ensure that all code it uses is ''re-entrant''; that is, designed so that multiple tasks can execute it concurrently.

For an example of code that is not re-entrant, suppose an application (written in C) uses two tasks, T1 and T2. Each task calls a function **add_to_table** that adds an integer to a table (array) of integers. The procedure uses two global variables, **num_of_entries** and **table**.

```
int num_of_entries = 0;
int table[100];

void
add_to_table(int x)
{
      table[num_of_entries] = x;
      num_of_entries = num_of_entries + 1;
}
```

Task T1 assigns x to **table[num_of_entries]** but is then suspended. Meanwhile, task T2 starts running and calls **add_to_table**. Since task T1 has not yet updated **num_of_entries**, task T2 overwrites the entry that task T1 has made. Presumably, the application intends that the table and entry count should reflect the calls made by both tasks; consequently, **add_to_table** is not re-entrant.

It is often possible to safely use code that is not re-entrant. For example, if a system library that is not re-entrant is called solely from one task, then it does not matter that the library is not re-entrant. Also, some libraries return handles to static data structures, where the handle is the only pathway to that storage. Consequently, if only one task ever uses a particular handle when it makes calls that are not re-entrant, no problems will occur. For example, **ios_$open** is not re-entrant, but a single task may safely use it.

If you create a library and you intend it to run with tasking applications, you should make the manager re-entrant by using Mutual Exclusion (mutex) locks. Mutex locks ensure that only one task at a time has access to the manager's data structures. In order to use a data structure, the task must first request and receive the lock associated with it. For example, to make the **add_to_table** procedure re-entrant, add mutex locks to it as follows:

```
int num_of_entries = 0;
int table[100];
mutex_$lock_rec_t table_lock;

void
add_to_table(int x)
{
      mutex_$lock(table_lock, mutex_$wait_forever);
      table[num_of_entries] = x;
      num_of_entries = num_of_entries + 1;
      mutex_$unlock(table_lock);
}
```

Note that the application must call **mutex_$init** before it makes the first call to the **add_to_table** procedure.

Not all Domain/OS libraries are re-entrant. The following table categorizes Domain/OS calls into those that are probably re-entrant (that is, they are most likely re-entrant but have not been thoroughly tested for this feature), partially re-entrant (they return handles as described above), not re-entrant, and those that are definitely re-entrant.

Re-entrant Properties of Domain/OS Calls			
Probably Re-entrant	**Partially Re-entrant**	**Not Re-entrant**	**Re-entrant**
ec2	cal	error	ctm
ms	fpp	gmf	ev
mutex	pbu	ios	gmr
name	proc2	ipc	gmr3d
pfm	sio	pad	gpr
proc1	smd	stream	mbx
rws	time	vfmt	pbufs
sfcb	tpad		pgm
task	vec		prf
			trait

Domain/OS calls without side effects (**time_$clock** for example) are generally re-entrant. Calls that open, create, or close objects (**ios_$open** and **ipc_$delete** for example) are not re-entrant.

The following additional restrictions apply to tasking:

- UNIX system and library calls are not supported. They may work, but they are not guaranteed to.
- Asynchronous fault handlers must not make any **task_$** calls.
- **Ec2_$wait_svc** behaves exactly like **ec2_$wait** when tasking is enabled. The reason is that the Task Manager does not know which task to notify when an asynchronous fault handler has returned to the point at which the fault occurred.

Constants

task_$max_priority
The highest task priority.

task_$name_max_len
The maximum number of bytes in a task name.

task_$dt_handle
The task handle for the Distinguished Task (DT).

Data Types

task_$handle_t
A task handle.

task_$info_pt
A pointer to type **task_$info_t**.

task_$info_t
A record type for passing task information. The diagram below illustrates the **task_$info_t** data type.

15	0
routine_to_run_in_task	
routine_to_run_in_task	
arg_ptr	
arg_ptr	
arg_len	
arg_len	
stack_size	
stack_size	
initial_priority	
initial_priority	
current_priority	
current_priority	
lifetime	
lifetime	
task_name_ptr	
task_name_ptr	
task_name_len	
task_name_len	
state	
level_created	
15	0

routine_to_run_in_task
A pointer to the task routine.

arg_ptr Pointer to the task's argument vector.

arg_len The number of bytes in the task's argument vector.

stack_size
The number of bytes in the task's stack.

initial_priority
The initial priority of the task.

current_priority
The priority at which the task is currently running.

lifetime The lifetime of the task. It is a value of type **task_$lifetime_t**.

task_name_ptr
A pointer to the name of the task.

task_name_len
The number of bytes in the task name.

state The run state of the task. It is a value of type **task_$run_state_t**.

level_created
The program level at which the task was created.

task_$lifetime_t
An enumerated type for specifying the lifetime of a task. (In Pascal, the values of type **task_$lifetime_t** are constants.) It takes one of the following values:

task_$forever
Create a task that lives independent of the process that created it.

task_$until_level_exit
The task lives until the level at which it was created is released.

task_$until_exec_or_level_exit
The task lives until the level at which it was created is released or overlaid by a UNIX exec(2) call.

task_$name_pt
A pointer to a task name.

task_$option_set_t
A small set type for specifying options to **task_$create**. There is currently only one value:

task_$intend_to_wait
Create a task completion eventcount that advances when the new task terminates. When created with the **task_$intend_to_wait** option, a task is suspended when it completes pending a **task_$release** call.

When created without the **task_$intend_to_wait** option, a task is not suspended when it completes; instead it terminates immediately on completion, and the completion status and output value of the task will not be accessible.

task_$routine_pt

A pointer to a task routine. A task routine must not return a value and must take two arguments: a pointer to an argument vector, followed by the number of bytes in the argument vector. The argument vector may contain anything.

In C, a valid task routine declaration is

```
void task_routine(
    void *arg_pointer,
    int arg_length)
```

In Pascal, a valid task routine declaration is

```
procedure task_routine(
    in arg_ptr: univ_ptr;
    in arg_len: integer32);
    options(val_param);
```

task_$run_state_t

An enumerated type for describing the run state of a task. It takes one of the following values:

task_$ready
The task is ready to run.

task_$waiting
The task is waiting.

Errors

task_$bad_$ec_wait
Internal error: bad status from **ec2_$wait**.

task_$bad_handle
Invalid task handle.

task_$cant_blast_dt
Attempted to blast the distinguished task.

task_$cant_blast_yourself
Task attempted to blast itself.

task_$exit_fault
Task exited.

task_$invalid_lifetime
Invalid task lifetime.

task_$invalid_name
Task name is too long.

task_$invalid_priority
Invalid task priority.

task_$lib_init_failed
Internal error: library initialization failed.

task_$no_backgrnd_from_foregrnd
A foreground task attempted to create a background task.

task_$no_completion_ec
Task does not have a completion eventcount.

task_$no_room
Not enough virtual memory left to create task.

task_$not_completed
Task not yet completed.

task_$not_found
Task does not exist.

task_$not_init
Task Manager has not been initialized.

task_$stack_corrupted
A task stack was corrupted.

task_$stack_overflow
A task stack overflowed.

task_$too_many_tasks
Attempted to exceed the maximum allowable number of tasks in a process.

SEE ALSO

ec2_$intro, fault_$intro, mutex_$intro, pfm_$intro.

NAME

task_$blast – kill a task without cleanup

SYNOPSIS (C)

```
#include <apollo/base.h>
#include <apollo/task.h>

void task_$blast(
        task_$handle_t task_handle,
        status_$t *status)
```

SYNOPSIS (Pascal)

```
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/task.ins.pas';

procedure task_$blast(
        in task_handle: task_$handle_t;
        out status: status_$t);
```

DESCRIPTION

Task_$blast kills the task identified by *task_handle*. Because the task is immediately destroyed without its clean-up handlers being invoked, **task_$blast** should be used only as a last resort.

task_handle
The handle of the task.

status The completion status.

NOTES

Task_$exit and **task_$signal** can terminate a task via a signal.

SEE ALSO

pfm_$intro, task_$create, task_$exit, task_$get_handle, task_$signal.

NAME

task_$create – create a task

SYNOPSIS (C)

```
#include <apollo/base.h>
#include <apollo/task.h>

task_$handle_t task_$create(
        void (*routine_pointer)(),
        char *arg_pointer,
        long arg_length,
        long stack_size,
        long task_priority,
        task_$lifetime_t life_time,
        task_$option_set_t task_options,
        status_$t *status)
```

SYNOPSIS (Pascal)

```
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/task.ins.pas';

function task_$create(
    in routine_pointer: task_$routine_p;
    in arg_pointer: univ_ptr;
    in arg_length: integer32;
    in stack_size: integer32;
    in task_priority: integer32;
    in life_time: task_$lifetime_t;
    in task_options: task_$option_set_t;
    out status: status_$t): task_$handle_t;
```

DESCRIPTION

Task_$create creates a new task and returns a handle for it. **Task_$create** allocates *stack_size* bytes for the task's stack, then creates an initial context by calling the routine that *routine_pointer* points to.

routine_pointer

A pointer to the routine to run in the task. The task routine must not return a value and must take two arguments: a pointer to an argument vector, followed by the number of bytes in the argument vector. The argument vector may contain anything.

In C, a valid task routine declaration is

```
void task_routine(
    void *arg_pointer,
    int arg_length)
```

In Pascal, a valid task routine declaration is

```
procedure task_routine(
    in arg_ptr: univ_ptr;
    in arg_len: integer32);
    options(val_param);
```

arg_pointer
A pointer to the argument vector for the routine at *routine_pointer*. The contents and format of the argument vector is not defined by the task library. If *arg_length* is 0, *arg_pointer* is passed directly to the routine at *routine_pointer*; otherwise, the argument vector is copied from *arg_pointer* onto the stack for the created task, and the task gets a pointer to that copy in its first argument.

arg_length
The number of bytes at *arg_pointer* to pass in the argument vector. The value of *arg_length* is passed to the routine at *routine_pointer* as its second argument when it is called.

stack_size
The number of bytes to allocate for the new task's stack. In general, *stack_size* should be at least 64K bytes so it can accommodate system calls, some of which require a large stack.

Task_$create increases the size of the stack to hold the argument vector, so *stack_size* should not include the size of the argument vector.

task_priority
The initial priority of the task. The priority levels are from 1 to 9, where 9 is the highest.

life_time
The lifetime of the new task. Specify one of the following values:

task_$forever
Create a task that lives independent of the process that created it.

task_$until_level_exit
Create a task that lives until the level at which it was created is released.

task_$until_exec_or_level_exit
Create a task that lives until the level at which it was created is released or overlaid by a UNIX exec(2) call.

The system does not ensure that a task's code is available for the duration of the task's lifetime; that is left up to the application. Thus, if *life_time* is **task_$forever**, the task's code must exist in a global library.

task_options
A small set of options when creating a new task. Currently there is one option:

task_$intend_to_wait
Create a task completion eventcount that advances when the created task terminates. When created with the **task_$intend_to_wait** option, a task is suspended when it completes pending a **task_$release** call.

If **task_$intend_to_wait** is not specified in *task_options* — that is, if *task_options* is 0 (or [] in Pascal) — then **task_$create** does not create a completion eventcount. When created without the **task_$intend_to_wait** option, a task is not suspended when it completes; instead it terminates immediately on completion, and the completion status and output value of the task will not be accessible.

status The completion status.

SEE ALSO

task_$get_ec, task_$release, task_$set_result.

NAME

task_$exit – exit a task

SYNOPSIS (C)

```
#include <apollo/base.h>
#include <apollo/task.h>

void task_$exit();
```

SYNOPSIS (Pascal)

```
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/task.ins.pas';

procedure task_$exit;
```

DESCRIPTION

Task_$exit terminates the calling task by signaling a **task_$exit_fault** fault. Consequently, any clean-up handlers for that fault established by the task will run.

SEE ALSO

pfm_$intro, task_$blast, task_$create, task_$get_handle, task_$signal.

NAME

task_$get_ec – get a completion eventcount

SYNOPSIS (C)

#include <apollo/base.h>
#include <apollo/task.h>

void task_$get_ec(
task_$handle_t *task_handle*,
ec2_$ptr_t **eventcount_pointer*,
status_$t **status*)

SYNOPSIS (Pascal)

%include '/sys/ins/base.ins.pas';
%include '/sys/ins/task.ins.pas';

procedure task_$get_ec(
in *task_handle*: **task_$handle_t;**
out *eventcount_pointer*: **ec2_$ptr_t;**
out *status*: **status_$t);**

DESCRIPTION

Task_$get_ec returns a pointer to the completion eventcount for the task identified by *task_handle*. The task must have been created with a completion eventcount; that is, the **task_$create** call that created it must have specified the **task_$intend_to_wait** option.

task_handle
The handle of the task, in **task_$handle_t** format.

eventcount_pointer
A pointer to an eventcount.

status The completion status.

SEE ALSO

ec2_$intro, task_$get_handle, task_$release, task_$set_result.

NAME

task_$get_handle – return the task handle

SYNOPSIS (C)

#include <apollo/base.h>
#include <apollo/task.h>

task_$handle_t task_$get_handle();

SYNOPSIS (Pascal)

%include '/sys/ins/base.ins.pas';
%include '/sys/ins/task.ins.pas';

function task_$get_handle: task_$handle_t;

DESCRIPTION

Task_$get_handle returns the task handle of the calling task.

SEE ALSO

task_$create.

NAME

task_$get_info – get information about a task

SYNOPSIS (C)

#include <apollo/base.h>
#include <apollo/task.h>

```
void task_$get_info(
        task_$handle_t task_handle,
        task_$info_pt info_pointer,
        task_$handle_t *next_handle,
        status_$t *status)
```

SYNOPSIS (Pascal)

%include '/sys/ins/base.ins.pas';
%include '/sys/ins/task.ins.pas';

```
procedure task_$get_info(
        in task_handle: task_$handle_t;
        in info_pointer: task_$info_pt;
        out next_handle: task_$handle_t;
        out status: status_$t);
```

DESCRIPTION

Task_$get_info gets information about the task identified by **task_handle**.

task_handle
A task handle.

info_pointer
The task information.

next_handle
The handle of the next task for which information can be obtained. The handles for tasks in a process are in a circular list. Repeated calls to **task_$get_info** with *task_handle* set to the *next_handle* supplied by the previous call will eventually return a *next_handle* equivalent to the *task_handle* when **task_$get_info** was first called, indicating that all tasks have been described.

status The completion status.

SEE ALSO

task_$create, task_$get_handle.

NAME

task_$release – release a task and report

SYNOPSIS (C)

```
#include <apollo/base.h>
#include <apollo/task.h>

void task_$release(
        task_$handle_t task_handle,
        status_$t *completion_status,
        long *output_value,
        status_$t *status)
```

SYNOPSIS (Pascal)

```
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/task.ins.pas';

procedure task_$release(
        in task_handle: task_$handle_t;
        out completion_status: status_$t;
        out output_value: integer32;
        out status: status_$t);
```

DESCRIPTION

Task_$release returns the completion status and output value of the task identified by *task_handle*, and then terminates the task. It may be called only if the task was created with a completion eventcount.

task_handle
: A task handle.

completion_status
: The completion status of the task identified by *task_handle*. If the task completed because it received a fault signal it couldn't handle, *completion_status* will be the fault code received. Otherwise, *completion_status* will be **status_$ok** unless the task changed it by calling **task_$set_result**.

output_value
: The output value of the task identified by *task_handle*. The task's output value will be 0 unless the task changed it by calling **task_$set_result**.

status
: The completion status. If **task_$release** is called before the task has completed, it returns with a completion status of **task_$not_completed**, and *completion_status* and *output_value* are undefined. In general, **task_$release** should not be called until the completion eventcount of the task specified by *task_handle* is advanced.

SEE ALSO

task_$create, task_$exit, task_$get_ec, task_$set_result.

NAME

task_$set_name – name a task

SYNOPSIS (C)

#include <apollo/base.h>
#include <apollo/task.h>

void task_$set_name(
task_$handle_t *task_handle*,
task_$name_pt *name_pointer*,
long *name_length*,
status_$t **status*)

SYNOPSIS (Pascal)

%include '/sys/ins/base.ins.pas';
%include '/sys/ins/task.ins.pas';

procedure task_$set_name(
in *task_handle*: **task_$handle_t;**
in *name_pointer*: **task_$name_pt;**
in *name_length*: **integer32;**
out *status*: **status_$t);**

DESCRIPTION

Task_$set_name associates the name pointed to by *name_pointer* with the task identified by *task_handle*. The name is copied into the task's information record, and is accessible via **task_$get_info**.

task_handle
A task handle.

name_pointer
A pointer to the name to be associated with the task.

name_length
The number of significant bytes at *name_pointer*. *Name_length* should be **task_$name_max_len** or less.

status The completion status.

SEE ALSO

task_$create, task_$get_handle.

NAME

task_$set_result – change the completion status and output value

SYNOPSIS (C)

```
#include <apollo/base.h>
#include <apollo/task.h>

void task_$set_result(
        status_$t completion_status,
        long output_value,
        status_$t *status)
```

SYNOPSIS (Pascal)

```
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/task.ins.pas';

procedure task_$set_result(
        in completion_status: status_$t;
        in output_value: integer32;
        out status: status_$t);
```

DESCRIPTION

Task_$set_result sets the completion status and output value of the calling process to *completion_status* and *output_value*, respectively. It may be called only by a task created with a completion eventcount.

completion_status
: The completion status for the calling task.

output_value
: The output value for the calling task.

status
: The completion status of **task_$set_result**.

NOTES

The completion status and output value of a task are reported by **task_$release** when the task is terminated.

SEE ALSO

task_$create, task_$get_handle, task_$release.

NAME

task_$signal – signal a task

SYNOPSIS (C)

```
#include <apollo/base.h>
#include <apollo/task.h>

void task_$signal(
        task_$handle_t task_handle,
        status_$t fault_status,
        status_$t *status)
```

SYNOPSIS (Pascal)

```
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/task.ins.pas';

procedure task_$signal(
        in task_handle: task_$handle_t;
        in fault_status: status_$t;
        out status: status_$t);
```

DESCRIPTION

Task_$signal sends the signal specified by *fault_status* to the task specified by *task_handle*. **Task_$signal** will activate any fault or clean-up handlers established by the task on *task_handle* to handle the fault specified by *fault_status*. If the task is not running or has inhibited faults, it will be signaled the next time it is running with faults enabled.

task_handle
A task handle.

fault_status
The signal to send to the task on *task_handle*. Also, the fault code passed to any fault or clean-up handler established to handle the fault.

status The completion status of **task_$signal**.

SEE ALSO

pfm_$intro, task_$blast, task_$create, task_$exit, task_$get_handle.

NAME

task_$tasking_enabled – determine whether tasking is enabled

SYNOPSIS (C)

```
#include &lt;apollo/base.h&gt;
#include &lt;apollo/task.h&gt;

boolean task_$tasking_enabled(void)
```

SYNOPSIS (Pascal)

```
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/task.ins.pas';

function task_$tasking_enabled: boolean;
```

DESCRIPTION

Task_$tasking_enabled returns **true** if tasking is enabled, or **false** if tasking is not enabled. Tasking is enabled when the Distinguished Task (DT) first makes a call to **task_$create**.

NAME

task_$yield – yield the processor

SYNOPSIS (C)

```
#include <apollo/base.h>
#include <apollo/task.h>

void task_$yield(void);
```

SYNOPSIS (Pascal)

```
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/task.ins.pas';

procedure task_$yield;
```

DESCRIPTION

Task_$yield yields the processor to the next ready task at the same priority level as the calling task. If there are no other ready tasks at that level, the calling task continues to run.

SEE ALSO

task_$exit.

time

Contents

NAME

intro – the Domain/OS time service

SYNOPSIS (C)

#include <apollo/base.h>
#include <apollo/time.h>

SYNOPSIS (Pascal)

%include '/sys/ins/base.ins.pas';
%include '/sys/ins/time.ins.pas';

SYNOPSIS (FORTRAN)

%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/time.ins.ftn'

DESCRIPTION

Domain/OS represents time as the number of 4-microsecond intervals that have elapsed since midnight (00:00) on 1 January 1980 (UTC). ("UTC" is an abbreviation for "Coordinated Universal Time," which is just another, less parochial, term for Greenwich Mean Time or "GMT.") The count of elapsed 4-microsecond intervals is a 48-bit integer value of type **time_$clock_t**. An integer of type **time_$clock_t** is called a "clock value."

The Domain/OS time service reports system time, provides an eventcount that is advanced at regular intervals, and allows the timed suspension of processes. The **time_$** interface consists of the following calls:

time_$clock	get the system clock value
time_$get_ec	get a pointer to the time eventcount
time_$wait	wait for an interval

Data Types

time_$clock_t

A record type for system clock values. A value of type **time_$clock_t** is a 48-bit integer count of 4-microsecond clock periods. A clock value has the following format:

15	0
high	
high	
low	
15	0

high The most significant 32 bits of the clock value.

low The least significant 16 bits of the clock value.

time_$clockh_t
A 32-bit unsigned integer for holding the most significant bits of a clock value.

time_$key_t
An enumerated type for choosing a time eventcount. It currently can take on only one value:

time_$clockh_key
An eventcount that is advanced about every 1/4 second.

time_$rel_abs_t
An enumerated type for specifying the type of clock value passed in an argument. It can take one of the following values:

time_$relative
The clock value specifies an interval.

time_$absolute
The clock value specifies a UTC time.

Errors

time_$bad_key
Bad key to **time_$get_ec**.

time_$wait_quit
Wait interrupted by quit fault.

NAME

time_$clock – get the system clock value

SYNOPSIS (C)

```
#include <apollo/base.h>
#include <apollo/time.h>

void time_$clock(
        time_$clock_t *clock_value)
```

SYNOPSIS (Pascal)

```
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/time.ins.pas';

procedure time_$clock(
        out clock_value: time_$clock_t);
```

SYNOPSIS (FORTRAN)

```
%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/time.ins.ftn'

        integer*2 clock_value(3), clockh
        integer*4 clockl

        equivalence (clockl, clock_value(2)), (clockh, clock_value(1))

        call time_$clock(clock_value)
```

DESCRIPTION

Time_$clock supplies the system clock value in *clock_value*.

clock_value

The value of the system clock.

SEE ALSO

cal_$apply_local_offset, cal_$get_local_time.

NAME

time_$get_ec – get a pointer to the time eventcount

SYNOPSIS (C)

```
#include <apollo/base.h>
#include <apollo/time.h>

void time_$get_ec(
        time_$key_t &time_key,
        ec2_$ptr_t *eventcount_pointer,
        status_$t *status)
```

SYNOPSIS (Pascal)

```
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/time.ins.pas';

procedure time_$get_ec(
        in time_key: time_$key_t;
        out eventcount_pointer: ec2_$ptr_t;
        out status: status_$t);
```

SYNOPSIS (FORTRAN)

```
%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/time.ins.ftn'

        integer*4 status, ec_value
        integer*2 time_key, event(3)

        equivalence (ec_value, event(1))

        integer*4 eventcount_pointer
        pointer /eventcount_pointer/ event

        call time_$get_ec(time_key, eventcount_pointer, status)
```

DESCRIPTION

Time_$get_ec supplies a pointer to an eventcount that is advanced about every 1/4 second. The incrementing interval is nominally 262,144 microseconds, but the exact interval varies with the system load.

time_key

A key specifying which time eventcount the system should return. The only defined value currently is **time_$clockh_key**, because there is only one time eventcount.

eventcount_pointer

A pointer to the eventcount specified by *time_key*.

status The completion status.

SEE ALSO

ec2_$intro.

NAME

time_$wait – wait for an interval

SYNOPSIS (C)

```
#include <apollo/base.h>
#include <apollo/time.h>

void time_$wait(
        time_$rel_abs_t &rel_abs,
        time_$clock_t &clock_value,
        status_$t *status)
```

SYNOPSIS (Pascal)

```
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/time.ins.pas';

procedure time_$wait(
        var rel_abs: time_$rel_abs_t ;
        in clock_value: time_$clock_t ;
        out status: status_$t);
```

SYNOPSIS (FORTRAN)

```
%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/time.ins.ftn'

          integer*2 clock_value(3), clockh, rel_abs
          integer*4 clockl, status

          equivalence (clockl, clock_value(2)), (clockh, clock_value(1))

          call time_$wait(rel_abs, clock_value, status)
```

DESCRIPTION

Time_$wait returns after the interval specified by *rel_abs* and *clock_value*.

rel_abs The type of clock value supplied supplied in *clock_value*. It can have one of the following values:

time_$relative

Clock_value specifies a relative interval to wait before returning. **Time_$wait** will return after the number of 4-microsecond periods specified by *clock_value* have elapsed.

time_$absolute

Clock_value specifies a system clock value at which **time_$wait** should return.

clock_value

A clock value specifying a relative or absolute time to wait for before returning.

status The completion status. If the completion status is **time_$wait_quit**, then **time_$wait** returned because of an asynchronous fault, not because the specified interval elapsed.

NOTES

Note that **time_$wait** expects a Coordinated Universal Time (UTC) time in *clock_value* if *rel_abs* equals **time_$absolute**. To convert local time into UTC time, use **cal_$remove_local_offset**.

SEE ALSO

ec2_$wait, ec2_$wait_slow_io, ec2_$wait_svc.

tone

Contents

NAME

intro – make a noise

SYNOPSIS (C)

#include <apollo/base.h>
#include <apollo/tone.h>

SYNOPSIS (Pascal)

%include '/sys/ins/base.ins.pas';
%include '/sys/ins/tone.ins.pas';

SYNOPSIS (FORTRAN)

%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/tone.ins.ftn'

DESCRIPTION

This section describes the **tone_$time** call, which allows software to command the node it's running on to make a noise. Only Apollo workstations shipped after 19 April 1982 can make a noise on command from software.

NAME

tone_$time – make a noise of a specified duration

SYNOPSIS (C)

```
#include <apollo/base.h>
#include <apollo/tone.h>

void tone_$time(time_$clock_t &time)
```

SYNOPSIS (Pascal)

```
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/tone.ins.pas';

procedure tone_$time(in time: time_$clock_t);
```

SYNOPSIS (FORTRAN)

```
%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/tone.ins.ftn'

        integer*4 clockl
        integer*2 clockh, time(3)

        equivalence (clockl, time(2)), (clockh, time(1))

        call tone_$time(time)
```

DESCRIPTION

This call commands the node to make a noise for the length of time specified in *time*.

time This is the duration of the noise in **time_$clock_t** format.

tpad

Contents

NAME
intro – locator (touchpad) manager calls

SYNOPSIS (C)
#include <apollo/base.h>
#include <apollo/tpad.h>

SYNOPSIS (Pascal)
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/tpad.ins.pas';

SYNOPSIS (FORTRAN)
%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/tpad.ins.ftn'

DESCRIPTION
The **tpad_$** system calls manage locating devices.

Data Types

tpad_$mode_t

This is an enumerated type for specifying how locator input is translated in to cursor movement. Variables of this type can take one of the following values:

tpad_$absolute
Absolute mode maps the pointing space to a part of the screen dictated by the scaling factor and the origin value. Thus, there is a one-to-one mapping between points in the pointing space and a subset of points on the screen.

tpad_$relative
In relative mode, the cursor responds only to device movement relative to the current position. The response of the cursor simulates response to mouse movement. This is the only meaningful mode to use with a mouse.

tpad_$rel_abs
In relative/absolute mode, the cursor responds as in absolute mode on first contact to establish a new current position. After first contact, the cursor responds as in relative mode. Momentary lifting of the finger from a touchpad (for less than about half a second) does not affect the cursor position.

smd_$pos_t

A record type for specifying display positions.

line The vertical coordinate.

column The horizontal coordinate.

31	16
line	
column	
15	0

NAME

tpad_$inq_dtype – return the last locator used

SYNOPSIS (C)

```
#include <apollo/base.h>
#include <apollo/tpad.h>

tpad_$dev_type_t tpad_$inq_dtype(void)
```

SYNOPSIS (Pascal)

```
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/tpad.ins.pas';

function tpad_$inq_dtype : tpad_$dev_type_t;
```

SYNOPSIS (FORTRAN)

```
%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/tpad.ins.ftn'

        integer*2 device
        device = tpad_$inq_dtype
```

DESCRIPTION

This call returns a value indicating the last locator used for input. The returned value can be one of the following:

tpad_$have_bitpad
A bitpad provided the last locator input.

tpad_$have_mouse
A mouse provided the last locator input.

tpad_$have_touchpad
A touchpad provided the last locator input.

tpad_$unknown
No locator input was received since the node was last booted.

SEE ALSO

tpad_$inquire.

NAME

tpad_$inquire – get information about the current locator response

SYNOPSIS (C)

```
#include <apollo/base.h>
#include <apollo/tpad.h>

void tpad_$inquire(
        tpad_$mode_t *mode,
        short *x_scale,
        short *y_scale,
        short *hysteresis,
        smd_$pos_t *origin)
```

SYNOPSIS (Pascal)

```
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/tpad.ins.pas';

procedure tpad_$inquire(
        out mode: tpad_$mode_t;
        out x_scale: integer;
        out y_scale: integer;
        out hysteresis: integer;
        out origin: smd_$pos_t);
```

SYNOPSIS (FORTRAN)

```
%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/tpad.ins.ftn'

      integer*2 mode, x_scale, y_scale, hysteresis
      integer*2 line, column, origin(2)

      equivalence (line, origin(1)), (column, origin(2))

      call tpad_$inquire(mode, x_scale, y_scale, hysteresis, origin)
```

DESCRIPTION

This call supplies information about the response of the current locator device. Use the information from this call to save the locator configuration for later restoration with the **tpad_$set_mode** call, or to change one aspect of the locator response without changing any other aspects.

mode The cursor mode. It can be one of the following predefined values:

tpad_$absolute

Absolute mode maps the locator to a part of the screen dictated by the scaling factor and the origin value. Thus, there is a one-to-one mapping between points in the locator space and a subset of points on the screen.

tpad_$relative
In relative mode, the cursor responds only to locator movement relative to the current position. The response of the cursor simulates response to mouse movement.

tpad_$rel_abs
In relative/absolute mode, the cursor responds as in absolute mode on first contact to establish a new current position. After first contact, the cursor responds as in relative mode. Momentary lifting of the finger from a touchpad (for less than about half a second) does not affect the current cursor position.

x_scale The scale factor in the x-dimension.

y_scale The scale factor in the y-dimension.

hysteresis
The hysteresis factor, in pixels.

The hysteresis factor prevents minor, unintentional movements from changing the cursor location. It effectively defines a box around the current location. The cursor does not move while the current location stays within the hysteresis box.

If the current location moves beyond the box, the distance the cursor moves is the difference between the new location and the past location less the hysteresis factor. The default hysteresis factor is 5.

origin The scale origin for x- and y-dimensions in **smd_$pos_t** format.

SEE ALSO
tpad_$inq_dtype, tpad_$re_range, tpad_$set_cursor, tpad_$set_mode.

NAME

tpad_$re_range – re-establishes the touchpad raw data range

SYNOPSIS (C)

```
#include <apollo/base.h>
#include <apollo/tpad.h>

void tpad_$re_range(void)
```

SYNOPSIS (Pascal)

```
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/tpad.ins.pas';

procedure tpad_$re_range;
```

SYNOPSIS (FORTRAN)

```
%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/tpad.ins.ftn'

        call tpad_$re_range
```

DESCRIPTION

This call re-establishes the touchpad raw data range over the next 1000 data points. This is also done for you at system boot.

NAME

tpad_$set_cursor – re-establish the locator origin in relative mode

SYNOPSIS (C)

```
#include <apollo/base.h>
#include <apollo/tpad.h>

void tpad_$set_cursor(
        smd_$pos_t &origin)
```

SYNOPSIS (Pascal)

```
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/tpad.ins.pas';

procedure tpad_$set_cursor(
        in origin: smd_$pos_t);
```

SYNOPSIS (FORTRAN)

```
%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/tpad.ins.ftn'

        integer*2 line, column, origin(2)

        equivalence (line, origin(1)), (column, origin(2))

        call tpad_$set_cursor(origin)
```

DESCRIPTION

This call is useful for relative and absolute/relative mode only and is primarily used with touchpad and bitpad locating devices.

The system remembers the last cursor position delivered by the locator. When new input comes from a locator in relative mode, a displacement is computed and applied to the previous locator position. The **tpad_$set_cursor** call makes the system forget the previous locator position, and use the value passed in the call instead. The next displacement will be computed from the position specified in the **tpad_$set_cursor** call instead of the locator's previous position. **Tpad_$set_cursor** can be called at any time and affect all subsequent locator input.

When using a touchpad or bitpad in relative mode, the origin is automatically re-established when the user takes his finger from the touchpad for more than one eighth of a second. One effect of relative mode is that the cursor doesn't move the next time the user touches the pad unless **tpad_$set_cursor** was called in the interim.

When using a touchpad in absolute/relative mode, a call to **tpad_$set_cursor** is only useful while the locator is functioning in the relative phase; that is, after the first touch and before the operator lifts his finger for more than one half second.

origin The screen position that will be the origin for subsequent input from the the locator in **smd_$pos_t** format.

SEE ALSO

tpad_$inquire, tpad_$set_mode.

NAME

tpad_$set_mode – set pointing device response characteristics

SYNOPSIS (C)

#include <apollo/base.h>
#include <apollo/tpad.h>

```
void tpad_$set_mode(
        tpad_$mode_t &mode,
        short &x_scale,
        short &y_scale,
        short &hysteresis,
        smd_$pos_t &origin)
```

SYNOPSIS (Pascal)

```
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/tpad.ins.pas';

procedure tpad_$set_mode(
        in mode: tpad_$mode_t;
        in x_scale: integer;
        in y_scale: integer;
        in hysteresis: integer;
        in origin: smd_$pos_t);
```

SYNOPSIS (FORTRAN)

```
%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/tpad.ins.ftn'

        integer*2 mode, x_scale, y_scale, hysteresis
        integer*2 line, column, origin(2)

        equivalence (line, origin(1)), (column, origin(2))

        call tpad_$set_mode(mode, x_scale, y_scale, hysteresis, origin)
```

DESCRIPTION

This call sets the mode, scale factors, and hysteresis factors of pointing devices. It can also change the origin for relative or absolute/relative mode. Use the output from **tpad_$inquire** as the input to this call to change one aspect of pointing device response without changing any other aspects.

A program can use this call with the touchpad, mouse, and bit pad pointing devices, but the mouse uses only the scale and hysteresis factors and ignores the other mode settings, since it is inherently a relative device.

mode The cursor mode to set. Specify one of the following predefined values:

tpad_$absolute
Absolute mode maps the pointing space to a part of the screen dictated by the scaling factor and the origin value. Thus, there is a one-to-one mapping between points in the pointing space and a subset of points on the screen.

tpad_$relative
In relative mode, the cursor responds only to device movement relative to the current position. The response of the cursor simulates response to mouse movement. This is the only meaningful mode to use with a mouse.

tpad_$rel_abs
In relative/absolute mode, the cursor responds as in absolute mode on first contact to establish a new current position. After first contact, the cursor responds as in relative mode. Momentary lifting of the finger from a touchpad (for less than about half a second) does not affect the cursor position.

x_scale The scale factor in the x dimension. Scale factors are not relevent to absolute mode.

y_scale The scale factor in the y dimension. Scale factors are not relevent to absolute mode.

hysteresis
The hysteresis factor, in pixels.

The hysteresis factor prevents minor, unintentional movements from changing the cursor location. It effectively defines a box around the current location. The cursor does not move while the current location stays within the hysteresis box.

If the current location moves beyond the box, the distance the cursor moves is the difference between the new location and the past location less the hysteresis factor. The default hysteresis factor is 5.

origin is scale origin for x- and y-dimensions in **smd_$pos_t** format.

SEE ALSO
tpad_$inq_dtype.

vec

Contents

NAME

intro – the Vector Library

SYNOPSIS (C)

#include <apollo/base.h>
#include <apollo/vec.h>

SYNOPSIS (Pascal)

%include '/sys/ins/base.ins.pas';
%include '/sys/ins/vec.ins.pas';

SYNOPSIS (FORTRAN)

%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/vec.ins.ftn'

DESCRIPTION

The Vector Library performs floating-point and integer vector matrix arithmetic.

Most of the vector calls have four versions: single-precision floating-point, double-precision floating-point, 16-bit integer, and 32-bit integer. The names of all single-precision vector calls begin with the simple prefix **vec_$**. Double-precision calls begin with the prefix **vec_$d**. The 16-bit integer calls begin with the prefix **vec_$i** and add a suffix of **16**. The 32-bit integer calls begin with the prefix **vec_$i**, but lack the **16** suffix. For example, **vec_$dot** and **vec_$ddot** are single- and double-precision versions of dot (scalar) product calls. **Vec_$idot** and **vec_$idot16** are the 32-bit and 16-bit integer versions, respectively.

Each variant of a routine takes similar arguments that differ from the arguments of the other variants only in the types of their operands. For the double-precision calls, all floating-point parameters are double-precision; for the single-precision calls, all floating-point parameters must be single precision; for the integer procedures and functions, the parameters and returned values are integers, etc.

Names that ultimately end in _i denote "incremental" calls, which step through vector arrays at user-specified increments. They are mainly useful for operations on vectors in matrixes that are not stored contiguously in memory.

NOTES

When using any of the vector calls, make sure that the indexes you pass are valid. In the interest of performance, these calls do **not** check index values for validity.

NAME

vec_$add_constant – add a scalar to a single-precision vector

SYNOPSIS (C)

```
#include <apollo/base.h>
#include <apollo/vec.h>

void vec_$add_constant(
        float *start_vec,
        long int &length,
        float &constant,
        float *result_vec)
```

SYNOPSIS (Pascal)

```
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/vec.ins.pas';

procedure vec_$add_constant(
        in start_vec: univ vec_$real_vector;
        in length: integer32;
        in constant: real;
        out result_vec: univ vec_$real_vector);
```

SYNOPSIS (FORTRAN)

```
%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/vec.ins.ftn'

        parameter (nvec = 256)

        real start_vec(nvec), result_vec(nvec), constant
        integer*4 length

        call vec_$add_constant(start_vec, length, constant, result_vec)
```

DESCRIPTION

Vec_$add_constant adds the scalar *constant* to the single-precision vector *start_vec*, and supplies the result in *result_vec*.

In C, the resulting operation is

```
for (i = 0; i < length; ++i)
     result_vec[i] = constant + start_vec[i];
```

In Pascal, the resulting operation is

```
for i := 1 to length do
      begin
      result_vec[i] := constant + start_vec[i];
      end
```

In FORTRAN, the resulting operation is

```
       do 10 i = 1, length
            result_vec(i) = constant + start_vec(i)
  10   continue
```

start_vec
The operand vector that *constant* will be added to.

length The number of elements in *start_vec* that *constant* will be added to.

constant
A scalar constant to be added to *start_vec*.

result_vec
The vector resulting from adding *constant* to *start_vec*.

NOTES

When **vec_$add_constant** is used to operate on matrixes in C and Pascal, *start_vec* and *result_vec* are row vectors; whereas in FORTRAN, they are column vectors.

SEE ALSO

vec_$add_constant_i, vec_$dadd_constant, vec_$iadd_constant, vec_$iadd_constant16.

NAME

vec_$add_constant_i – add a scalar to a vector in a single-precision matrix

SYNOPSIS (C)

```
#include <apollo/base.h>
#include <apollo/vec.h>

void vec_$add_constant_i(
        float *start_vec,
        long int &inc1,
        long int &length,
        float &constant,
        float *result_vec,
        long int &inc2)
```

SYNOPSIS (Pascal)

```
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/vec.ins.pas';

procedure vec_$add_constant_i(
        in start_vec: univ vec_$real_vector;
        in inc1: integer32;
        in length: integer32;
        in constant: real;
        out result_vec: univ vec_$real_vector;
        in inc2: integer32);
```

SYNOPSIS (FORTRAN)

```
%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/vec.ins.ftn'

      parameter (nvec = 10)

      real start_vec(nvec), result_vec(nvec), constant
      integer*4 length, inc1, inc2

      call vec_$add_constant_i(start_vec, inc1, length,
     &                          constant, result_vec, inc2)
```

DESCRIPTION

Vec_$add_constant_i adds the scalar *constant* to elements of the single-precision array *start_vec* selected by *inc1*, and puts the sums in elements of the array *result_vec* selected by *inc2*.

Through appropriate choice of *inc1* and *inc2*, a program can use **vec_$add_constant_i** to operate on individual vectors in a matrix. To add *constant* to the *M*th vector in a matrix, choose *inc1* equal to the number of vectors in the matrix, and place the *M*th element of the matrix array at the beginning of *start_vec*. To place the result of the operation in the

*N*th vector of a matrix, choose *inc2* equal to the number of vectors in the resultant matrix, and place the *N*th element of the matrix array at the beginning of *result_vec*.

In C, the resulting operation is

```
j = 0;
k = 0;
for (i = 0; i < length; ++i) {
      result_vec[k] = constant + start_vec[j];
      k += inc2;
      j += inc1;
}
```

In Pascal, the resulting operation is

```
j := 1;
k := 1;
for i := 1 to length do
      begin
      result_vec[k] := constant + start_vec[j];
      k := k + inc2;
      j := j + inc1;
      end
```

In FORTRAN, the resulting operation is

```
      j = 1
      k = 1
      do 10 i = 1, length
           result_vec(k) = constant + start_vec(j)
           k = k + inc2
           j = j + inc1
  10  continue
```

start_vec
: The operand array whose elements will be summed with *constant*.

inc1
: Increment for the index of *start_vec* used to select the elements of *start_vec* that will be summed with *constant*.

length
: The number of elements in *start_vec* that *constant* will be added to.

constant
: The scalar value to be summed with elements of *start_vec*.

result_vec
: The array resulting from summing *constant* with elements of *start_vec*.

inc2 Increment for the index of *result_vec* used to select the elements of *result_vec* that will receive the sums of *constant* with the elements of *start_vec* selected by *inc1*.

NOTES

In C and Pascal, **vec_$add_constant_i** operates on column vectors; whereas in FORTRAN, it operates on row vectors.

SEE ALSO

vec_$add_constant, vec_$dadd_constant_i, vec_$iadd_constant_i, vec_$iadd_constant16_i.

NAME

vec_$add_vector – add two single-precision vectors

SYNOPSIS (C)

```
#include <apollo/base.h>
#include <apollo/vec.h>

void vec_$add_vector(
        float *start_vec,
        float *add_vec,
        long int &length,
        float *result_vec)
```

SYNOPSIS (Pascal)

```
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/vec.ins.pas';

procedure vec_$add_vector(
        in start_vec: univ vec_$real_vector;
        in add_vec: univ vec_$real_vector;
        in length: integer32;
        out result_vec: univ vec_$real_vector);
```

SYNOPSIS (FORTRAN)

```
%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/vec.ins.ftn'

        parameter (nvec = 10)

        real start_vec(nvec), add_vec(nvec), result_vec(nvec)
        integer*4 length

        call vec_$add_vector(start_vec, add_vec, length, result_vec)
```

DESCRIPTION

Vec_$add_vector adds the single-precision vectors *start_vec* and *add_vec*, and supplies the vector sum in *result_vec*.

In C, the resulting operation is

```
for (i = 0; i < length; ++i)
     result_vec[i] = start_vec[i] + add_vec[i];
```

In Pascal, the resulting operation is

```
for i := 1 to length do
      begin
      result_vec[i] := start_vec[i] + add_vec[i];
      end
```

In FORTRAN, the resulting operation is

```
       do 10 i = 1, length
           result_vec(i) = start_vec(i) + add_vec(i)
  10   continue
```

start_vec
An addend vector.

add_vec
An addend vector.

length Number of elements to sum. *Length* is usually just the order of the addends.

result_vec
The vector that is the sum of *start_vec* and *add_vec*.

NOTES

When **vec_$add_vector** is used to operate on matrixes in C and Pascal, *start_vec* and *result_vec* are row vectors; whereas in FORTRAN, they are column vectors.

SEE ALSO

vec_$add_vector_i, vec_$dadd_vector, vec_$iadd_vector, vec_$iadd_vector16.

NAME

vec_$add_vector_i – add vectors in two single-precision matrixes

SYNOPSIS (C)

```
#include <apollo/base.h>
#include <apollo/vec.h>

void vec_$add_vector_i(
        float *start_vec,
        long int &inc1,
        float *add_vec,
        long int &inc2,
        long int &length,
        float *result_vec,
        long int &inc3)
```

SYNOPSIS (Pascal)

```
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/vec.ins.pas';

procedure vec_$add_vector_i(
        in start_vec: univ vec_$real_vector;
        in inc1: integer32;
        in add_vec: univ vec_$real_vector;
        in inc2: integer32;
        in length: integer32;
        out result_vec: univ vec_$real_vector;
        in inc3: integer32);
```

SYNOPSIS (FORTRAN)

```
%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/vec.ins.ftn'

      parameter (nvec = 10)

      real start_vec(nvec), add_vec(nvec), result_vec(nvec)
      integer*4 length, inc1, inc2, inc3

      call vec_$add_vector_i(start_vec, inc1, add_vec, inc2,
     &                       length, result_vec, inc3)
```

DESCRIPTION

Vec_$add_vector_i adds elements of the array *start_vec*, selected by *inc1*, to elements of the array *add_vec*, selected by *inc1*, and puts the sums in elements of the array *result_vec* selected by *inc3*.

Through appropriate choice of *inc1*, *inc2*, and *inc3*, a program can use **vec_$add_vector_i** to add individual vectors in two matrixes and place the sum in a vector of another matrix. To add the *M*th vector in matrix *X* to the *N*th vector in matrix *Y*, choose *inc1* equal to the number of vectors in matrix *X* and *inc2* equal to the number of vectors in matrix *Y*. Then place the *M*th element of matrix *X* at the beginning of *start_vec*, and place the *N*th element of matrix *Y* at the beginning of *add_vec*. To place the result of the operation in the *P*th vector of a resultant matrix, choose inc2 equal to the number of vectors in the resultant matrix, and place the *P*th element of the matrix array at the beginning of *result_vec*.

In C, the resulting operation is

```
j = 0;
k = 0;
m = 0;
for (i = 0; i < length; ++i) {
      result_vec[j] = start_vec[k] + add_vec[m];
      j += inc3;
      k += inc1;
      m += inc2;
}
```

In Pascal, the resulting operation is

```
j := 1;
k := 1;
m := 1;
for i := 1 to length do
      begin
      result_vec[j] := start_vec[k] + add_vec[m];
      j := j + inc3;
      k := k + inc1;
      m := m + inc2;
      end
```

In FORTRAN, the resulting operation is

```
      j = 1
      k = 1
      m = 1
      do 10 i = 1, length
           result_vec(j) = start_vec(k) + add_vec(m)
           j = j + inc3
           k = k + inc1
           m = m + inc2
 10   continue
```

start_vec
An array whose elements will be summed with elements of *add_vec*.

inc1 Increment for the index of *start_vec* used to select the elements of *start_vec* that will be summed with elements of *add_vec*.

add_vec
An array whose elements will be summed with elements of *start_vec*.

inc2 Increment for the index of *add_vec* used to select the elements of *add_vec* that will be summed with elements of *start_vec*.

length The number of elements in *start_vec* that will be summed with elements of *add_vec*.

result_vec
An array to contain the *length* sums of elements of *start_vec* and *add_vec*.

inc3 Increment for the index of *result_vec* used to select the elements of *result_vec* that will receive the sums.

NOTES

In C and Pascal, **vec_$add_vector_i** operates on column vectors; whereas in FORTRAN, it operates on row vectors.

SEE ALSO

vec_$add_vector, vec_$dadd_vector_i, vec_$iadd_vector16_i, vec_$iadd_vector_i.

NAME

vec_$copy – copy a single-precision vector

SYNOPSIS (C)

#include <apollo/base.h>
#include <apollo/vec.h>

void vec_$copy(
float **start_vec*,
float **result_vec*,
long int *&length*)

SYNOPSIS (Pascal)

%include '/sys/ins/base.ins.pas';
%include '/sys/ins/vec.ins.pas';

procedure vec_$copy(
in *start_vec*: **univ vec_$real_vector;**
out *result_vec*: **univ vec_$real_vector;**
in *length*: **integer32);**

SYNOPSIS (FORTRAN)

%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/vec.ins.ftn'

parameter (*nvec* = **10**)

real *start_vec(nvec)*, *result_vec(nvec)*
integer*4 *length*

call vec_$copy(*start_vec*, *result_vec*, *length*)

DESCRIPTION

Vec_$copy copies *length* elements from *start_vec* to *result_vec*.

In C, the resulting operation is

```
for (i = 0; i < length; ++i)
      result_vec[i] = start_vec[i];
```

In Pascal, the resulting operation is

```
for i := 1 to length do
      begin
      result_vec[i] := start_vec[i];
      end
```

In FORTRAN, the resulting operation is

```
       do 10 i = 1, length
            result_vec(i) = start_vec(i)
  10   continue
```

start_vec
The vector that *result_vec* will be copied from.

result_vec
The vector that *start_vec* will be copied to.

length The number of elements to copy from *start_vec* to *result_vec*.

NOTES

When vec_$copy is used to operate on matrixes in C and Pascal, *start_vec* and *result_vec* are row vectors; whereas in FORTRAN, they are column vectors.

SEE ALSO

vec_$copy_i, vec_$dcopy, vec_$icopy, vec_$icopy16.

NAME

vec_$copy_i – copy a vector from one single-precision matrix to another

SYNOPSIS (C)

```
#include <apollo/base.h>
#include <apollo/vec.h>

void vec_$copy_i(
        float *start_vec,
        long int &inc1,
        float *result_vec,
        long int &inc2,
        long int &length)
```

SYNOPSIS (Pascal)

```
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/vec.ins.pas';

procedure vec_$copy_i(
        in start_vec: univ vec_$real_vector;
        in inc1: integer32;
        out result_vec: univ vec_$real_vector;
        in inc2: integer32;
        in length: integer32);
```

SYNOPSIS (FORTRAN)

```
%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/vec.ins.ftn'

        parameter (nvec = 10)

        real start_vec(nvec), result_vec(nvec)
        integer*4 length, inc1, inc2

        call vec_$copy_i(start_vec, inc1, result_vec, inc2, length)
```

DESCRIPTION

Vec_$copy_i copies *length* elements of the single-precision array *start_vec* selected by *inc1* into elements of *result_vec* selected by *inc2*.

Through appropriate choice of *inc1* and *inc2*, a program can use vec_$copy_i to copy a vector from one matrix to another. To copy the *M*th vector in a matrix, choose *inc1* equal to the number of vectors in the matrix, and place the *M*th element of the matrix array at the beginning of *start_vec*. To place the copy into the *N*th vector of a matrix, choose *inc2* equal to the number of vectors in the resultant matrix, and place the *N*th element of the matrix array at the beginning of *result_vec*.

In C, the resulting operation is

```
j = 0;
k = 0;
for (i = 0; i < length; ++i) {
      result_vec[k] = start_vec[j];
      k += inc2;
      j += inc1;
}
```

In Pascal, the resulting operation is

```
j := 1;
k := 1;
for i := 1 to length do
      begin
      result_vec[k] := start_vec[j];
      k := k + inc2;
      j := j + inc1;
      end
```

In FORTRAN, the resulting operation is

```
      j = 1
      k = 1
      do 10 i = 1, length
           result_vec(j) = start_vec(k)
           j = j + inc2
           k = k + inc1
 10   continue
```

start_vec
The array whose elements will be copied to *result_vec*.

inc1 Increment for the index of *start_vec* used to select the elements of *start_vec* that will be copied to *result_vec*.

result_vec
The array resulting from copying elements of *start_vec* selected by *inc1* into elements of *result_vec* selected by *inc2*.

inc2 Increment for the index of *result_vec* used to select the elements of *result_vec* that will receive the copies of the elements of *start_vec* selected by *inc1*.

length The number of elements in *start_vec* that will be copied to *result_vec*.

NOTES

In C and Pascal, **vec_$copy_i** copies column vectors; whereas in FORTRAN, it copies row vectors.

SEE ALSO

vec_$copy, vec_$dcopy_i, vec_$icopy16_i, vec_$icopy_i.

NAME

vec_$dadd_constant – add a scalar to a double-precision vector

SYNOPSIS (C)

```
#include <apollo/base.h>
#include <apollo/vec.h>

void vec_$dadd_constant(
        double *start_vec,
        long int &length,
        double &constant,
        double *result_vec)
```

SYNOPSIS (Pascal)

```
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/vec.ins.pas';

procedure vec_$dadd_constant(
        in start_vec: univ vec_$double_vector;
        in length: integer32;
        in constant: double;
        out result_vec: univ vec_$double_vector);
```

SYNOPSIS (FORTRAN)

```
%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/vec.ins.ftn'

        parameter (nvec = 256)

        double precision start_vec(nvec)
        double precision result_vec(nvec)
        double precision constant
        integer*4 length

        call vec_$dadd_constant(start_vec, length, constant, result_vec)
```

DESCRIPTION

Vec_$dadd_constant adds the scalar *constant* to the double-precision vector *start_vec*, and supplies the result in *result_vec*.

In C, the resulting operation is

```
for (i = 0; i < length; ++i)
      result_vec[i] = constant + start_vec[i];
```

In Pascal, the resulting operation is

```
for i := 1 to length do
      begin
      result_vec[i] := constant + start_vec[i];
      end
```

In FORTRAN, the resulting operation is

```
       do 10 i = 1, length
            result_vec(i) = constant + start_vec(i)
   10  continue
```

start_vec
The operand vector that *constant* will be added to.

length The number of elements in *start_vec* that *constant* will be added to.

constant
A scalar constant to be added to *start_vec*.

result_vec
The vector resulting from adding *constant* to *start_vec*.

NOTES

When **vec_$dadd_constant** is used to operate on matrixes in C and Pascal, *start_vec* and *result_vec* are row vectors; whereas in FORTRAN, they are column vectors.

SEE ALSO

vec_$add_constant, vec_$add_constant_i, vec_$iadd_constant, vec_$iadd_constant16.

NAME

vec_$dadd_constant_i – add a scalar to a vector in a double-precision matrix

SYNOPSIS (C)

```
#include <apollo/base.h>
#include <apollo/vec.h>

void vec_$dadd_constant_i(
        float *start_vec,
        long int &inc1,
        long int &length,
        float &constant,
        float *result_vec,
        long int &inc2)
```

SYNOPSIS (Pascal)

```
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/vec.ins.pas';

procedure vec_$dadd_constant_i(
        in start_vec: univ vec_$real_vector;
        in inc1: integer32;
        in length: integer32;
        in constant: real;
        out result_vec: univ vec_$real_vector;
        in inc2: integer32);
```

SYNOPSIS (FORTRAN)

```
%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/vec.ins.ftn'

      parameter (nvec = 10)

      real start_vec(nvec), result_vec(nvec), constant
      integer*4 length, inc1, inc2

      call vec_$dadd_constant_i(start_vec, inc1, length,
     &                          constant, result_vec, inc2)
```

DESCRIPTION

Vec_$dadd_constant_i adds the scalar *constant* to elements of the double-precision array *start_vec* selected by *inc1*, and puts the sums in elements of the array *result_vec* selected by *inc2*.

Through appropriate choice of *inc1* and *inc2*, a program can use **vec_$dadd_constant_i** to operate on individual vectors in a matrix. To add *constant* to the *M*th vector in a matrix, choose *inc1* equal to the number of vectors in the matrix, and place the *M*th element of the matrix array at the beginning of *start_vec*. To place the result of the operation in the *N*th vector of a matrix, choose *inc2* equal to the number of vectors in the resultant matrix, and place the *N*th element of the matrix array at the beginning of *result_vec*.

In C, the resulting operation is

```
j = 0;
k = 0;
for (i = 0; i < length; ++i) {
      result_vec[k] = constant + start_vec[j];
      k += inc2;
      j += inc1;
}
```

In Pascal, the resulting operation is

```
j := 1;
k := 1;
for i := 1 to length do
      begin
      result_vec[k] := constant + start_vec[j];
      k := k + inc2;
      j := j + inc1;
      end
```

In FORTRAN, the resulting operation is

```
       j = 1
       k = 1
       do 10 i = 1, length
            result_vec(k) = constant + start_vec(j)
            k = k + inc2
            j = j + inc1
 10    continue
```

start_vec
The operand array whose elements will be summed with *constant*.

inc1 Increment for the index of *start_vec* used to select the elements of *start_vec* that will be summed with *constant*.

length The number of elements in *start_vec* that *constant* will be added to.

constant
The scalar value to be summed with elements of *start_vec*.

result_vec
The array resulting from summing *constant* with elements of *start_vec*.

inc2 Increment for the index of *result_vec* used to select the elements of *result_vec* that will receive the sums of *constant* with the elements of *start_vec* selected by *inc1*.

NOTES

In C and Pascal, **vec_$dadd_constant_i** operates on column vectors; whereas in FORTRAN, it operates on row vectors.

SEE ALSO

vec_$add_constant, vec_$add_constant_i, vec_$iadd_constant_i, vec_$iadd_constant16_i.

NAME

vec_$dadd_vector – add two double-precision vectors

SYNOPSIS (C)

```
#include <apollo/base.h>
#include <apollo/vec.h>

void vec_$dadd_vector(
        double *start_vec,
        double *add_vec,
        long int &length,
        double *result_vec)
```

SYNOPSIS (Pascal)

```
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/vec.ins.pas';

procedure vec_$dadd_vector(
        in start_vec: univ vec_$double_vector;
        in add_vec: univ vec_$double_vector;
        in length: integer32;
        out result_vec: univ vec_$double_vector);
```

SYNOPSIS (FORTRAN)

```
%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/vec.ins.ftn'

      parameter (nvec = 10)

      double-precision start_vec(nvec), add_vec(nvec), result_vec(nvec)
      integer*4 length

      call vec_$dadd_vector(start_vec, add_vec, length, result_vec)
```

DESCRIPTION

Vec_$dadd_vector adds the double-precision vectors *start_vec* and *add_vec* and supplies the vector sum in *result_vec*.

In C, the resulting operation is

```
for (i = 0; i < length; ++i)
     result_vec[i] = start_vec[i] + add_vec[i];
```

In Pascal, the resulting operation is

```
for i := 1 to length do
      begin
      result_vec[i] := start_vec[i] + add_vec[i];
      end
```

In FORTRAN, the resulting operation is

```
       do 10 i = 1, length
            result_vec(i) = start_vec(i) + add_vec(i)
   10  continue
```

start_vec
An addend vector.

add_vec
An addend vector.

length Number of elements to sum. *Length* is usually just the order of the addends.

result_vec
The vector that is the sum of *start_vec* and *add_vec*.

NOTES

When **vec_$dadd_vector** is used to operate on matrixes in C and Pascal, *start_vec* and *result_vec* are row vectors; whereas in FORTRAN, they are column vectors.

SEE ALSO

vec_$add_vector, vec_$iadd_vector, vec_$iadd_vector16.

NAME

vec_$dadd_vector_i – add vectors in two double-precision matrixes

SYNOPSIS (C)

```
#include <apollo/base.h>
#include <apollo/vec.h>

void vec_$dadd_vector_i(
        double *start_vec,
        long int &inc1,
        double *add_vec,
        long int &inc2,
        long int &length,
        double *result_vec,
        long int &inc3)
```

SYNOPSIS (Pascal)

```
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/vec.ins.pas';

procedure vec_$dadd_vector_i(
        in start_vec: univ vec_$double_vector;
        in inc1: integer32;
        in add_vec: univ vec_$double_vector;
        in inc2: integer32;
        in length: integer32;
        out result_vec: univ vec_$double_vector;
        in inc3: integer32);
```

SYNOPSIS (FORTRAN)

```
%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/vec.ins.ftn'

      parameter (nvec = 10)

      double precision start_vec(nvec), add_vec(nvec), result_vec(nvec)
      integer*4 length, inc1, inc2, inc3

      call vec_$dadd_vector_i(start_vec, inc1, add_vec, inc2,
     &                        length, result_vec, inc3)
```

DESCRIPTION

Vec_$dadd_vector_i adds elements of the array *start_vec*, selected by *inc1*, to elements of the array *add_vec*, selected by *inc1*, and puts the sums in elements of the array *result_vec* selected by *inc3*.

Through appropriate choice of *inc1*, *inc2*, and *inc3*, a program can use **vec_$dadd_vector_i** to add individual vectors in two matrixes and place the sum in a

vector of another matrix. To add the *M*th vector in matrix *X* to the *N*th vector in matrix *Y*, choose *inc1* equal to the number of vectors in matrix *X* and *inc2* equal to the number of vectors in matrix *Y*. Then place the *M*th element of matrix *X* at the beginning of *start_vec*, and place the *N*th element of matrix *Y* at the beginning of *add_vec*. To place the result of the operation in the *P*th vector of a resultant matrix, choose *inc2* equal to the number of vectors in the resultant matrix, and place the *P*th element of the matrix array at the beginning of *result_vec*.

In C, the resulting operation is

```
j = 0;
k = 0;
m = 0;
for (i = 0; i < length; ++i) {
      result_vec[j] = start_vec[k] + add_vec[m];
      j += inc3;
      k += inc1;
      m += inc2;
}
```

In Pascal, the resulting operation is

```
j := 1;
k := 1;
m := 1;
for i := 1 to length do
      begin
      result_vec[j] := start_vec[k] + add_vec[m];
      j := j + inc3;
      k := k + inc1;
      m := m + inc2;
      end
```

In FORTRAN, the resulting operation is

```
      j = 1
      k = 1
      m = 1
      do 10 i = 1, length
           result_vec(j) = start_vec(k) + add_vec(m)
           j = j + inc3
           k = k + inc1
           m = m + inc2
 10   continue
```

start_vec
An array whose elements will be summed with elements of *add_vec*.

inc1 Increment for the index of *start_vec* used to select the elements of *start_vec* that will be summed with elements of *add_vec*.

add_vec
An array whose elements will be summed with elements of *start_vec*.

inc2 Increment for the index of *add_vec* used to select the elements of *add_vec* that will be summed with elements of *start_vec*.

length The number of elements in *start_vec* that will be summed with elements of *add_vec*.

result_vec
An array to contain the *length* sums of elements of *start_vec* and *add_vec*.

inc3 Increment for the index of *result_vec* used to select the elements of *result_vec* that will receive the sums.

NOTES

In C and Pascal, **vec_$dadd_vector_i** operates on column vectors; whereas in FORTRAN, it operates on row vectors.

SEE ALSO

vec_$add_vector_i, vec_$iadd_vector16_i, vec_$iadd_vector_i.

NAME

vec_$dcopy – copy a double-precision vector

SYNOPSIS (C)

#include <apollo/base.h>
#include <apollo/vec.h>

void vec_$dcopy(
**double ***start_vec*,
**double ***result_vec*,
long int &*length***)**

SYNOPSIS (Pascal)

%include '/sys/ins/base.ins.pas';
%include '/sys/ins/vec.ins.pas';

procedure vec_$dcopy(
in *start_vec*: **univ vec_$double_vector;**
out *result_vec*: **univ vec_$double_vector;**
in *length*: **integer32);**

SYNOPSIS (FORTRAN)

%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/vec.ins.ftn'

parameter (*nvec* **= 10)**

double precision *start_vec(nvec)*, *result_vec(nvec)*
integer*4 *length*

call vec_$dcopy(*start_vec*, *result_vec*, *length***)**

DESCRIPTION

Vec_$dcopy copies *length* elements from *start_vec* to *result_vec*.

In C, the resulting operation is

```
for (i = 0; i < length; ++i)
      result_vec[i] = start_vec[i];
```

In Pascal, the resulting operation is

```
for i := 1 to length do
      begin
      result_vec[i] := start_vec[i];
      end
```

In FORTRAN, the resulting operation is

```
      do 10 i = 1, length
           result_vec(i) = start_vec(i)
 10   continue
```

start_vec
The vector that *result_vec* will be copied from.

result_vec
The vector that *start_vec* will be copied to.

length The number of elements to copy from *start_vec* to *result_vec*.

NOTES
When **vec_$dcopy** is used to operate on matrixes in C and Pascal, *start_vec* and *result_vec* are row vectors; whereas in FORTRAN, they are column vectors.

SEE ALSO
vec_$copy, vec_$dcopy_i, vec_$icopy, vec_$icopy16.

NAME

vec_$dcopy_i – copy a vector from one double-precision matrix to another

SYNOPSIS (C)

```
#include &lt;apollo/base.h&gt;
#include &lt;apollo/vec.h&gt;

void vec_$dcopy_i(
        double *start_vec,
        long int &incl,
        double *result_vec,
        long int &inc2,
        long int &length)
```

SYNOPSIS (Pascal)

```
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/vec.ins.pas';

procedure vec_$dcopy_i(
        in start_vec: univ vec_$double_vector;
        in incl: integer32;
        out result_vec: univ vec_$double_vector;
        in inc2: integer32;
        in length: integer32);
```

SYNOPSIS (FORTRAN)

```
%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/vec.ins.ftn'

        parameter (nvec = 10)

        double precision start_vec(nvec), result_vec(nvec)
        integer*4 length, incl, inc2

        call vec_$dcopy_i(start_vec, incl, result_vec, inc2, length)
```

DESCRIPTION

Vec_$dcopy_i copies *length* elements of the double-precision array *start_vec* selected by *incl* into elements of *result_vec* selected by *inc2*.

Through appropriate choice of *incl* and *inc2*, a program can use **vec_$dcopy_i** to copy a vector from one matrix to another. To copy the *M*th vector in a matrix, choose *incl* equal to the number of vectors in the matrix, and place the *M*th element of the matrix array at the beginning of *start_vec*. To place the copy into the *N*th vector of a matrix, choose *inc2* equal to the number of vectors in the resultant matrix, and place the *N*th element of the matrix array at the beginning of *result_vec*.

In C, the resulting operation is

```
j = 0;
k = 0;
for (i = 0; i < length; ++i) {
      result_vec[k] = start_vec[j];
      k += inc2;
      j += inc1;
}
```

In Pascal, the resulting operation is

```
j := 1;
k := 1;
for i := 1 to length do
      begin
      result_vec[k] := start_vec[j];
      k := k + inc2;
      j := j + inc1;
      end
```

In FORTRAN, the resulting operation is

```
      j = 1
      k = 1
      do 10 i = 1, length
           result_vec(j) = start_vec(k)
           j = j + inc2
           k = k + inc1
   10 continue
```

start_vec
: The array whose elements will be copied to *result_vec*.

inc1
: Increment for the index of *start_vec* used to select the elements of *start_vec* that will be copied to *result_vec*.

result_vec
: The array resulting from copying elements of *start_vec* selected by *inc1* into elements of *result_vec* selected by *inc2*.

inc2
: Increment for the index of *result_vec* used to select the elements of *result_vec* that will receive the copies of the elements of *start_vec* selected by *inc1*.

length
: The number of elements in *start_vec* that will be copied to *result_vec*.

NOTES

In C and Pascal, **vec_$dcopy_i** copies column vectors; whereas in FORTRAN, it copies row vectors.

SEE ALSO

vec_$copy_i, vec_$dcopy, vec_$icopy16_i, vec_$icopy_i.

NAME

vec_$ddot – return the dot product of two double-precision vectors

SYNOPSIS (C)

```
#include <apollo/base.h>
#include <apollo/vec.h>

double vec_$ddot(
        double *vector1,
        double *vector2,
        long int &length)
```

SYNOPSIS (Pascal)

```
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/vec.ins.pas';

function vec_$ddot(
        in vector1: univ vec_$double_vector;
        in vector2: univ vec_$double_vector;
        in length: integer32): double;
```

SYNOPSIS (FORTRAN)

```
%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/vec.ins.ftn'

        parameter (nvec = 10)

        double-precision vector1(nvec), vector2(nvec), result
        integer*4 length

        result = vec_$ddot(vector1, vector2, length)
```

DESCRIPTION

Vec_$ddot returns the dot (scalar) product of two single-precision vectors, *vector1* and *vector1*.

In C, the resulting operation is

```
return_value = 0.0;
for (i = 0; i < length; ++i)
      return_value += vector1[i] * vector2[i];
```

In Pascal, the resulting operation is

```
return_value := 0.0;
for i := 1 to length do
      begin
      return_value := return_value
                   + vector1[i] * vector2[i];
      end
```

In FORTRAN, the resulting operation is

```
       vec_$ddot = 0.0
       do 10 i = 1,length
            vec_$ddot = vec_$ddot + vector1(i) * vector2(i)
  10   continue
```

vector1 A vector.

vector2 Another vector.

length The number of elements to use in calculating the dot (scalar) product.

NOTES

When **vec_$ddot** is used on matrixes in C or Pascal, *vector1* and *vector2* are row vectors; whereas in FORTRAN, they are column vectors.

SEE ALSO

vec_$ddot_i, vec_$dot, vec_$idot, vec_$idot16.

NAME

vec_$ddot_i – return the dot product of two vectors in double-precision matrixes

SYNOPSIS (C)

```
#include <apollo/base.h>
#include <apollo/vec.h>

double vec_$ddot_i(
        double *vector1,
        long int &inc1,
        double *vector2,
        long int &inc2,
        long int &length)
```

SYNOPSIS (Pascal)

```
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/vec.ins.pas';

function vec_$ddot_i(
        in vector1: univ vec_$double_vector;
        in inc1: integer32;
        in vector2: univ vec_$double_vector;
        in inc2: integer32;
        in length: integer32): double;
```

SYNOPSIS (FORTRAN)

```
%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/vec.ins.ftn'

        parameter (nvec = 10)

        double precision vector1(nvec), vector2(nvec), result
        integer*4 length, inc1, inc2

        result = vec_$ddot_i(vector1, inc1, vector2, inc2, length)
```

DESCRIPTION

Vec_$ddot_i returns the dot (scalar) product of two vectors from the double-precision arrays *vector1* and *vector2*.

In C, the resulting operation is

```
j = 0;
k = 0;
return_value = 0.0;
for (i = 0; i < length; ++i) {
      return_value += vector1[j] * vector2[k];
      j += inc1;
      k += inc2;
}
```

In Pascal, the resulting operation is

```
j := 1;
k := 1;
return_value := 0.0;
for i := 1 to length do
      begin
      return_value := return_value
                   + vector1[j] * vector2[k];
      j := j + inc1;
      k := k + inc2;
      end;
```

In FORTRAN, the resulting operation is

```
       j = 1
       k = 1
       vec_$ddot_i = 0.0
       do 10 i = 1,length
            vec_$ddot_i = vec_$ddot_i
      &                 + vector1(j) * vector2(k)
                  j = j + inc1
                  k = k + inc2
  10   continue
```

vector1 An array containing one of the vectors to use in calculating the dot product.

inc1 An increment for *vector1* that chooses which elements will be used to calculate the product.

vector2 An array containing the other vector to use in calculating the dot product.

inc2 An increment for *vector2* that chooses which elements will be used to calculate the product.

length The number of elements from *vector1* or *vector2* to use in calculating the dot product.

NOTES

When **vec_$ddot_i** is used on matrixes in C or Pascal, *vector1* and *vector2* are column vectors; whereas in FORTRAN, they are row vectors.

SEE ALSO

vec_$ddot, vec_$dot_i, vec_$idot16_i, vec_$idot_i.

NAME

vec_$dinit – initialize a double-precision vector

SYNOPSIS (C)

```
#include <apollo/base.h>
#include <apollo/vec.h>

void vec_$dinit(
        double *vector,
        long int &length,
        double &constant)
```

SYNOPSIS (Pascal)

```
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/vec.ins.pas';

procedure vec_$dinit(
        var vector: univ vec_$double_vector;
        in length: integer32;
        in constant: double);
```

SYNOPSIS (FORTRAN)

```
%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/vec.ins.ftn'

        parameter (nvec = 10)

        double precision vector(nvec), constant
        integer*4 length

        call vec_$dinit(vector, length, constant)
```

DESCRIPTION

Vec_$dinit sets *length* elements of *vector* to the double-precision value *constant*.

In C, the resulting operation is

```
for (i = 0; i < length; ++i)
      vector[i] = constant;
```

In Pascal, the resulting operation is

```
for i := 1 to length do
      begin
      vector[i] := constant;
      end
```

In FORTRAN, the resulting operation is

```
      do 10 i = 1, length
          vector(i) = constant
  10  continue
```

vector The vector to initialize.

length The number of elements in *vector* to initialize.

constant
The value that the elements of *vector* should be set to.

NOTES

In C and Pascal, vec_**$dinit** initializes a row vector; whereas in FORTRAN, it initializes a column vector.

SEE ALSO

vec_$dinit_i, vec_$iinit, vec_$iinit16, vec_$init.

NAME

vec_$dmat_mult – multiply two 4×4 double-precision matrixes

SYNOPSIS (C)

```
#include <apollo/base.h>
#include <apollo/vec.h>

void vec_$dmat_mult(
        double *matrix1,
        double *matrix2,
        double *out_matrix)
```

SYNOPSIS (Pascal)

```
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/vec.ins.pas';

procedure vec_$dmat_mult(
        in matrix1: univ vec_$double_matrix;
        in matrix2: univ vec_$double_matrix;
        out out_matrix: univ vec_$double_matrix);
```

SYNOPSIS (FORTRAN)

```
%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/vec.ins.ftn'

        double precision matrix1(4, 4), matrix2(4, 4), out_matrix(4, 4)

        call vec_$dmat_mult(matrix1, matrix2, out_matrix)
```

DESCRIPTION

Vec_$dmat_mult multiplies two 4×4 matrixes, *matrix1* and *matrix2*, and supplies the result in *out_matrix*.

In C, **vec_$dmat_mult** calculates the product of *matrix2* on the left and *matrix1* on the right, and the resulting operation is

```
for (i = 0; i < 4; ++i)
        for (j = 0; j < 4; ++j) {
                out_mat[j,i] = 0.0;
                for (k = 0; k < 4; ++k)
                        out_mat[j][i] += matrix1[k][i]
                                       * matrix2[j][k];
        }
```

In Pascal, **vec_$dmat_mult** calculates the product of *matrix2* on the left and *matrix1* on the right, and the resulting operation is

```
for i := 1 to 4 do
      for j := 1 to 4 do
            begin
            out_mat[j,i] := 0.0;
            for k := 1 to 4 do
                  out_mat[j,i] := out_mat[j,i]
                               + matrix1[k,i]
                               * matrix2[j,k];
            end;
```

In FORTRAN, **vec_$dmat_mult** calculates the product of *matrix1* on the left and *matrix2* on the right, and the resulting operation is

```
         do 10 i = 1, 4
              do 10 j = 1, 4
                   out_mat(i,j) = 0.0
                   do 10 k = 1, 4
                        out_mat(i,j) = out_mat(i,j)
     &                               + matrix1(i,k)
     &                               * matrix2(k,j)
  10     continue
```

matrix1 A 4×4 matrix.

matrix2 Another 4×4 matrix.

out_matrix
The product of *matrix1* and *matrix2*.

NOTES

Vec_$dmat_multn performs the same operations for variably dimensioned matrixes.

SEE ALSO

vec_$imat_mult, vec_$imat_mult16, vec_$mat_mult.

NAME

vec_$dmat_multn – multiply two double-precision matrixes

SYNOPSIS (C)

```
#include <apollo/base.h>
#include <apollo/vec.h>

void vec_$dmat_multn(
        double *matrix1,
        double *matrix2,
        long int &m,
        long int &n,
        long int &s,
        double *out_matrix)
```

SYNOPSIS (Pascal)

```
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/vec.ins.pas';

procedure vec_$dmat_multn(
        in matrix1: univ vec_$double_matrix;
        in matrix2: univ vec_$double_matrix;
        in m: integer32;
        in n: integer32;
        in s: integer32;
        out out_matrix: univ vec_$double_matrix);
```

SYNOPSIS (FORTRAN)

```
%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/vec.ins.ftn'

        integer*4 m, n, s
        parameter (m = 3, n = 4, s = 5)

        double precision matrix1(m, n), matrix2(n, s), out_matrix(m, s)

        call vec_$dmat_multn(matrix1, matrix2, m, n, s, out_matrix)
```

DESCRIPTION

Vec_$dmat_multn multiplies two variably dimensioned matrixes, *matrix1* and *matrix2*, and supplies the result in *out_matrix*.

In C, **vec_$dmat_multn** calculates the product of *matrix2* on the left and *matrix1* on the right, and the resulting operation is

```
for (i = 0; i < m; ++i)
      for (j = 0; j < s; ++j) {
            out_matrix[j][i] = 0.0;
            for (k = 0; k < n; ++k)
                  out_matrix[j][i] += matrix1[k][i]
                                   * matrix2[j][k];
      }
```

In Pascal, **vec_$dmat_multn** calculates the product of *matrix2* on the left and *matrix1* on the right, and the resulting operation is

```
for i := 1 to m do
      for j := 1 to s do
            begin
            out_matrix[j,i] = 0.0;
            for k := 1 to n do
                  out_matrix[j,i] := out_matrix[j,i]
                                   + matrix1[k,i]
                                   * matrix2[j,k];
            end;
```

In FORTRAN, **vec_$dmat_multn** calculates the product of *matrix1* on the left and *matrix2* on the right, and the resulting operation is

```
        do 10 i = 1, m
             do 10 j = 1, s
                   out_matrix(i,j) = 0.0
                   do 10 k = 1, n
                         out_matrix(i,j) = out_matrix(i,j)
      &                                  + matrix1(i,k)
      &                                  * matrix2(k,j)
   10   continue
```

matrix1 A matrix to be multiplied.

matrix2 Another matrix to be multiplied.

m, n, s The various matrix dimensions.

out_matrix
The product of *matrix1* and *matrix2*.

SEE ALSO
vec_$imat_multn, vec_$imat_multn16, vec_$mat_multn.

NAME

vec_$dmax – find the maximum absolute value in a double-precision vector

SYNOPSIS (C)

```
#include <apollo/base.h>
#include <apollo/vec.h>

void vec_$dmax(
        double *vector,
        long int &length,
        double *result,
        long int *location)
```

SYNOPSIS (Pascal)

```
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/vec.ins.pas';

procedure vec_$dmax(
        in vector: univ vec_$double_vector;
        in length: integer32;
        out result: double;
        out location: integer32);
```

SYNOPSIS (FORTRAN)

```
%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/vec.ins.ftn'

         parameter (nvec = 10)

         double precision vector(nvec), result
         integer*4 length, location

         call vec_$dmax(vector, length, result, location)
```

DESCRIPTION

Vec_$dmax searches through *length* elements of *vector* and supplies the value and location of the element with the greatest absolute value.

In C, the resulting operation is

```
result = fabs(vector[0]);
location = 1;
for (i = 1; i < length; ++i)
      if (fabs(vector[i]) > result) {
            location = i + 1;
            result = fabs(vector[i]);
      }
```

In Pascal, the resulting operation is

```
result := abs(vector[1]);
location = 1;
for 10 i := 2 to length do
      if (abs(vector[i]) > result) then
            begin
            location := i;
            result := abs(vector[i]);
            end
```

In FORTRAN, the resulting operation is

```
       result = dabs(vector(1))
       location = 1
       do 10 i = 2, length
            if (dabs(vector(i)) .gt. result) then
                  location = i
                  result = dabs(vector(i))
            endif
  10   continue
```

vector The vector to search.

length The number of elements to search.

result The maximum absolute value of all the elements searched.

location The location of the element with the greatest absolute value. The location supplied in *location* is just the index of the element with the greatest absolute value in FORTRAN or Pascal (if *vector* is declared to begin with index 1). In C, *location* - 1 is the index of the element.

NOTES

In C and Pascal, **vec_$dmax** searches a row vector; whereas in FORTRAN, it searches a column vector.

SEE ALSO

vec_$dmax_i, vec_$imax, vec_$imax16, vec_$max.

NAME

vec_$dmax_i – find the maximum absolute value in a vector from a double-precision matrix

SYNOPSIS (C)

```
#include <apollo/base.h>
#include <apollo/vec.h>

void vec_$dmax_i(
        double *vector,
        long int &inc,
        long int &length,
        double *result,
        long int *location)
```

SYNOPSIS (Pascal)

```
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/vec.ins.pas';

procedure vec_$dmax_i(
        in vector: univ vec_$double_vector;
        in inc: integer32;
        in length: integer32;
        out result: double;
        out location: integer32);
```

SYNOPSIS (FORTRAN)

```
%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/vec.ins.ftn'

        parameter (nvec = 10)

        double precision vector(nvec), result
        integer*4 length, inc, location

        call vec_$dmax_i(vector, inc, length, result, location)
```

DESCRIPTION

Vec_$dmax_i searches through the *length* elements of *vector* selected by *inc*, and supplies the value and location of the element with the greatest absolute value.

Through appropriate choice of *inc*, a program can use **vec_$dmax_i** to search a vector within a matrix. To search the *M*th vector in a matrix, choose *inc* equal to the number of vectors in the matrix, and place the *M*th element of the matrix array at the beginning of *vector*.

In C, the resulting operation is

```
result = fabs(vector[0]);
location = 1;
j = inc;
for (i = 1; i < length, ++i) {
      if (fabs(vector[j]) > result) {
            location = i + 1;
            result = fabs(vector[j]);
      }
      j += inc;
}
```

In Pascal, the resulting operation is

```
result := abs(vector[1]);
location := 1;
j := 1 + inc;
for 10 i := 2 to length do
      begin
      if (abs(vector[j]) > result) then
            begin
            location := i;
            result := abs(vector[j]);
            end
      j := j + inc;
      end
```

In FORTRAN, the resulting operation is

```
      result = dabs(vector(1))
      location = 1
      j = 1 + inc
      do 10 i = 2, length
           if (dabs(vector(j)) .gt. result) then
                location = i
                result = dabs(vector(j))
           endif
      j = j + inc
 10   continue
```

vector The array to search.

inc An increment for the index of *vector* that selects the elements to search.

length The number of elements to search.

result The maximum absolute value of all the elements searched.

location The location of the element with the greatest absolute value. The location supplied in *location* is just the index of the element with the greatest absolute value in FORTRAN or Pascal (if *vector* is declared to begin with index 1). In C, *location* - 1 is the index of the element.

NOTES

In C and Pascal, **vec_$dmax_i** searches a column vector; whereas in FORTRAN, it searches a row vector.

SEE ALSO

vec_$dmax, vec_$imax16_i, vec_$imax_i, vec_$max_i.

NAME

vec_$dmult_add – scale and add one double-precision vector to another

SYNOPSIS (C)

#include <apollo/base.h>
#include <apollo/vec.h>

void vec_$dmult_add(
  double **add_vec*,
  double **mult_vec*,
  long int *&length*,
  double *&constant*,
  double **result_vec*)

SYNOPSIS (Pascal)

%include '/sys/ins/base.ins.pas';
%include '/sys/ins/vec.ins.pas';

procedure vec_$dmult_add(
  in *add_vec*: **univ vec_$double_vector;**
  in *mult_vec*: **univ vec_$double_vector;**
  in *length*: **integer32;**
  in *constant*: **double;**
  out *result_vec*: **univ vec_$double_vector);**

SYNOPSIS (FORTRAN)

%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/vec.ins.ftn'

  parameter (*nvec* = **10**)

  double precision *add_vec(nvec), mult_vec(nvec), result_vec(nvec), constant*
  integer*4 *length*

  call vec_$dmult_add(*add_vec, mult_vec, length,*
 &   *constant, result_vec*)

DESCRIPTION

Vec_$dmult_add multiplies the vector *mult_vec* by the scalar *constant*, adds the product to the vector *add_vec*, and supplies the resulting vector in *result_vec*.

In C, the resulting operation is

```
for (i = 0; i< length; ++i)
      result_vec[i] = add_vec[i]
                    + constant * mult_vec[i];
```

In Pascal, the resulting operation is

```
for i := 1 to length do
      result_vec[i] := add_vec[i]
                     + constant * mult_vec[i];
```

In FORTRAN, the resulting operation is

```
       do 10 i = 1, length
            result_vec(i) = add_vec(i)
      &                     + constant * mult_vec(i)
  10   continue
```

add_vec
: The vector to add to the product of *mult_vec* and *constant*.

mult_vec
: The vector to scale by *constant* and add to *add_vec*.

length The number of elements to use in the calculation.

constant
: The scalar value used to scale *mult_vec*.

result_vec
: The vector resulting from multiplying *mult_vec* by *constant* and adding the product to *add_vec*.

NOTES

When **vec_$dmult_add** is used to operate on matrixes in C and Pascal, *add_vec*, *mult_vec*, and *result_vec* are row vectors; whereas in FORTRAN, they are column vectors.

SEE ALSO

vec_$dmult_add_i, vec_$imult_add, vec_$imult_add16, vec_$mult_add.

NAME

vec_$dmult_add_i – scale and add double-precision vectors in matrixes

SYNOPSIS (C)

```
#include <apollo/base.h>
#include <apollo/vec.h>

void vec_$dmult_add_i(
        double *add_vec,
        long int &inc1,
        double *mult_vec,
        long int &inc2,
        long int &length,
        double &constant,
        double *result_vec,
        long int &inc3)
```

SYNOPSIS (Pascal)

```
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/vec.ins.pas';

procedure vec_$dmult_add_i(
        in add_vec: univ vec_$double_vector;
        in inc1: integer32;
        in mult_vec: univ vec_$double_vector;
        in inc2: integer32;
        in length: integer32;
        in constant: double;
        out result_vec: univ vec_$double_vector;
        in inc3: integer32);
```

SYNOPSIS (FORTRAN)

```
%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/vec.ins.ftn'

      parameter (nvec = 10)

      double precision add_vec(nvec), mult_vec(nvec), result_vec(nvec), constant
      integer*4 length, inc1, inc2, inc3

      call vec_$dmult_add_i(add_vec, inc1, mult_vec, inc2, length,
     &                      constant, result_vec, inc3)
```

DESCRIPTION

Vec_$dmult_add_i multiplies *length* elements of *mult_vec* selected by *inc2* by the scalar *constant*, adds the resulting products to elements of *add_vec* selected by *inc1*, and supplies the results in elements of *result_vec* selected by *inc3*.

Through appropriate choice of *inc1*, *inc2*, and *inc3*, a program can use **vec_$dmult_add_i** to scale and sum individual vectors in two matrixes and place the result in a vector of another matrix. To scale and add the *N*th vector in matrix *Y* to the *M*th vector in matrix *X*, choose *inc2* equal to the number of vectors in matrix *Y* and *inc1* equal to the number of vectors in matrix *X*. Then place the *M*th element of matrix *X* at the beginning of *add_vec*, and place the *N*th element of matrix *Y* at the beginning of *mult_vec*. To place the result of the operation in the *P*th vector of a resultant matrix, choose *inc3* equal to the number of vectors in the resultant matrix, and place the *P*th element of the matrix array at the beginning of *result_vec*.

In C, the resulting operation is

```
j = 0;
k = 0;
l = 0;
for (i = 0; i< length; ++i) {
      result_vec[l] = add_vec[j]
                    + constant * mult_vec[k];
      j += inc1;
      k += inc2;
      l += inc3;
}
```

In Pascal, the resulting operation is

```
j := 1;
k := 1;
l := 1;
for i := 1 to length do
      begin
      result_vec[l] := add_vec[j]
                     + constant * mult_vec[k];
      j := j + inc1;
      k := k + inc2;
      l := l + inc3;
      end
```

In FORTRAN, the resulting operation is

```
          j = 1
          k = 1
          l = 1
          do 10 i = 1, length
                result(l) = add_vec(j)
         &                 + constant * mult_vec(k)
                j = j + inc1
                k = k + inc2
                l = l + inc3
   10     continue
```

add_vec
: The vector to add to the product of *mult_vec* and *constant*.

inc1
: An increment for the index of *add_vec* that selects the elements to add to the product of *mult_vec* and *constant*.

mult_vec
: The vector to scale by *constant* and add to *add_vec*.

inc2
: An increment for the index of *mult_vec* that selects the elements to multiply by *constant* and add to *add_vec*.

length
: The number of elements to use in the calculation.

constant
: The scalar value used to scale *mult_vec*.

result_vec
: The vector resulting from multiplying *mult_vec* by *constant* and adding the product to *add_vec*.

inc3
: An increment for the index of *result_vec* that selects the elements to receive the results of multiplying *mult_vec* by *constant* and adding the product to *add_vec*.

NOTES

In C and Pascal, **vec_$dmult_add_i** operates on column vectors; whereas in FORTRAN, it operates on row vectors.

SEE ALSO

vec_$dmult_add, vec_$imult_add16_i, vec_$imult_add_i, vec_$mult_add_i.

NAME

vec_$dmult_constant – multiply a double-precision vector by a scalar

SYNOPSIS (C)

#include <apollo/base.h>
#include <apollo/vec.h>

void vec_$dmult_constant(
    double ***mult_vec*,**
    long int &*length*,
    double &*constant*,
    double ***result_vec*)**

SYNOPSIS (Pascal)

%include '/sys/ins/base.ins.pas';
%include '/sys/ins/vec.ins.pas';

procedure vec_$dmult_constant(
    in *mult_vec*: **univ vec_$double_vector;**
    in *length*: **integer32;**
    in *constant*: **double;**
    out *result_vec*: **univ vec_$double_vector);**

SYNOPSIS (FORTRAN)

%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/vec.ins.ftn'

parameter (*nvec* = **10**)

double precision *mult_vec*(*nvec*), *result_vec*(*nvec*), *constant*
integer*4 *length*

call vec_$dmult_constant(*mult_vec*, *length*, *constant*, *result_vec*)

DESCRIPTION

Vec_$dmult_constant multiplies the double-precision array *mult_vec* by the scalar value *constant* and supplies the result in the array *result_vec*.

In C, the resulting operation is

```
for (i = 0; i < length; ++i)
     result_vec[i] = constant * start_vec[i];
```

In Pascal, the resulting operation is

```
for i := 1 to length do
      begin
      result_vec[i] := constant * start_vec[i];
      end
```

In FORTRAN, the resulting operation is

```
       do 10 i = 1, length
            result_vec(i) = constant * mult_vec(i)
  10   continue
```

mult_vec
The vector to multiply by *constant*.

length The number of elements to multiply.

constant
The scalar constant to multiply *mult_vec* by.

result_vec
The vector resulting from multiplying *mult_vec* by *constant*.

NOTES

When **vec_$dmult_constant** is used to operate on matrixes in C and Pascal, *mult_vec* and *result_vec* are row vectors; whereas in FORTRAN, they are column vectors.

SEE ALSO

vec_$dmult_constant_i, vec_$imult_constant, vec_$imult_constant16, vec_$mult_constant.

NAME

vec_$dmult_constant_i – multiply a vector in a double-precision matrix by a scalar

SYNOPSIS (C)

```
#include <apollo/base.h>
#include <apollo/vec.h>

void vec_$dmult_constant_i(
        double *mult_vec,
        long int &inc1,
        long int &length,
        double &constant,
        double *result_vec,
        long int &inc2)
```

SYNOPSIS (Pascal)

```
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/vec.ins.pas';

procedure vec_$dmult_constant_i(
        in mult_vec: univ vec_$double_vector;
        in inc1: integer32;
        in length: integer32;
        in constant: double;
        out result_vec: univ vec_$double_vector;
        in inc2: integer32);
```

SYNOPSIS (FORTRAN)

```
%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/vec.ins.ftn'

      parameter (nvec = 10)

      double precision mult_vec(nvec), result_vec(nvec), constant
      integer*4 length, inc1, inc2

      call vec_$dmult_constant_i(mult_vec, inc1, length, constant,
     &                           result_vec, inc2)
```

DESCRIPTION

Vec_$dmult_constant_i multiplies *length* elements of the double-precision array *mult_vec* selected by *inc1* by the scalar value *constant*, and supplies the result in elements of the array *result_vec* selected by *inc2*.

Through appropriate choice of *inc1* and *inc2*, a program can use **vec_$dmult_constant_i** to operate on a vector in a matrix. To multiply *M*th vector in a matrix by *constant*, choose *inc1* equal to the number of vectors in the matrix, and place the *M*th element of the matrix array at the beginning of *mult_vec*. To place the result of the operation in the

*N*th vector of a matrix, choose *inc2* equal to the number of vectors in the resultant matrix, and place the *N*th element of the matrix array at the beginning of *result_vec*.

In C, the resulting operation is

```
j = 0;
k = 0;
for (i = 0; i < length; ++i) {
      result_vec[k] = constant * mult_vec[j];
      k += inc2;
      j += inc1;
}
```

In Pascal, the resulting operation is

```
j := 1;
k := 1;
for i := 1 to length do
      begin
      result_vec[k] := constant * mult_vec[j];
      k := k + inc2;
      j := j + inc1;
      end
```

In FORTRAN, the resulting operation is

```
      j = 1
      k = 1
      do 10 i = 1, length
           result(j) = constant * mult_vec(k)
           j = j + inc2
           k = k + inc1
 10   continue
```

mult_vec
: The vector to multiply by *constant*.

inc1
: An increment for the index of the array *mult_vec* that selects elements to multiply by *constant*.

length
: The number of products to calculate.

constant
: The scalar constant to multiply elements of *mult_vec* by.

result_vec
: An array whose elements receive the product of *constant* and *mult_vec*.

inc2 An increment for the index of the array *result_vec* that selects elements to receive the product of *constant* and *mult_vec*.

NOTES

In C and Pascal, **vec_$dmult_constant_i** operates on column vectors; whereas in FORTRAN, it operates on row vectors.

SEE ALSO

vec_$dmult_constant, vec_$imult_constant16_i, vec_$imult_constant_i, vec_$mult_constant_i.

NAME

vec_$dot – return the dot product of two single-precision vectors

SYNOPSIS (C)

```
#include <apollo/base.h>
#include <apollo/vec.h>

float vec_$dot(
        float *vector1,
        float *vector2,
        long int &length)
```

SYNOPSIS (Pascal)

```
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/vec.ins.pas';

function vec_$dot(
        in vector1: univ vec_$real_vector;
        in vector2: univ vec_$real_vector;
        in length: integer32): real;
```

SYNOPSIS (FORTRAN)

```
%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/vec.ins.ftn'

          parameter (nvec = 10)

          real vector1(nvec), vector2(nvec), result
          integer*4 length

          result = vec_$dot(vector1, vector2, length)
```

DESCRIPTION

Vec_$dot returns the dot (scalar) product of two single-precision vectors, *vector1* and *vector2*.

In C, the resulting operation is

```
return_value = 0.0;
for (i = 0; i < length; ++i)
     return_value += vector1[i] * vector2[i];
```

In Pascal, the resulting operation is

```
return_value := 0.0;
for i := 1 to length do
      begin
      return_value := return_value
                   + vector1[i] * vector2[i];
      end
```

In FORTRAN, the resulting operation is

```
       vec_$dot = 0.0
       do 10 i = 1,length
            vec_$dot = vec_$dot + vector1(i) * vector2(i)
  10   continue
```

vector1 A vector.

vector2 Another vector.

length The number of elements to use in calculating the dot (scalar) product.

NOTES

When **vec_$dot** is used on matrixes in C or Pascal, *vector1* and *vector2* are row vectors; whereas in FORTRAN they are column vectors.

SEE ALSO

vec_$ddot, vec_$dot_i, vec_$idot, vec_$idot16.

NAME

vec_$dot_i – return the dot product of two vectors in single-precision matrixes

SYNOPSIS (C)

```
#include <apollo/base.h>
#include <apollo/vec.h>

float vec_$dot_i(
        float *vector1,
        long int &inc1,
        float *vector2,
        long int &inc2,
        long int &length)
```

SYNOPSIS (Pascal)

```
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/vec.ins.pas';

function vec_$dot_i(
        in vector1: univ vec_$real_vector;
        in inc1: integer32;
        in vector2: univ vec_$real_vector;
        in inc2: integer32;
        in length: integer32): real;
```

SYNOPSIS (FORTRAN)

```
%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/vec.ins.ftn'

         parameter (nvec = 10)

         real vector1(nvec), vector2(nvec), result
         integer*4 length, inc1, inc2

         result = vec_$dot_i(vector1, inc1, vector2, inc2, length)
```

DESCRIPTION

Vec_$dot_i returns the dot (scalar) product of two vectors from the single-precision arrays *vector1* and *vector2*.

In C, the resulting operation is

```
j = 0;
k = 0;
return_value = 0.0;
for (i = 0; i < length; ++i) {
      return_value += vector1[j] * vector2[k];
      j += inc1;
      k += inc2;
}
```

In Pascal, the resulting operation is

```
j := 1;
k := 1;
return_value := 0.0;
for i := 1 to length do
      begin
      return_value := return_value
                   + vector1[j] * vector2[k];
      j := j + inc1;
      k := k + inc2;
      end;
```

In FORTRAN, the resulting operation is

```
      j = 1
      k = 1
      vec_$dot_i = 0.0
      do 10 i = 1,length
           vec_$dot_i = vec_$dot_i
     &                + vector1(j) * vector2(k)
                j = j + inc1
                k = k + inc2
 10   continue
```

vector1 An array containing one of the vectors to use in calculating the dot product.

inc1 An increment for *vector1* that chooses which elements will be used to calculate the product.

vector2 An array containing the other vector to use in calculating the dot product.

inc2 An increment for *vector2* that chooses which elements will be used to calculate the product.

length The number of elements from *vector1* or *vector2* to use in calculating the dot product.

NOTES

When **vec_$dot_i** is used on matrixes in C or Pascal, *vector1* and *vector2* are column vectors; whereas in FORTRAN, they are row vectors.

SEE ALSO

vec_$ddot_i, vec_$dot, vec_$idot16_i, vec_$idot_i.

NAME

vec_$dp_sp – copy a double-precision vector to a single-precision vector

SYNOPSIS (C)

```
#include <apollo/base.h>
#include <apollo/vec.h>

void vec_$dp_sp(
        double *dp_vec,
        float *sp_vec,
        long int &length)
```

SYNOPSIS (Pascal)

```
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/vec.ins.pas';

procedure vec_$dp_sp(
        in dp_vec: univ vec_$double_vector;
        in sp_vec: univ vec_$real_vector;
        in length: integer32);
```

SYNOPSIS (FORTRAN)

```
%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/vec.ins.ftn'

      parameter (nvec = 10)

      real sp_vec(nvec)
      double precision dp_vec(nvec)
      integer*4 length

      call vec_$dp_sp(dp_vec, sp_vec, length)
```

DESCRIPTION

Vec_$dp_sp copies *length* elements from the double-precision vector *dp_vec* to the single-precision vector *sp_vec*.

In C, the resulting operation is

```
for (i = 0; i < length; ++i)
      sp_vec[i] = (float)dp_vec[i];
```

In Pascal, the resulting operation is

```
for i := 1 to length do
      begin
      sp_vec[i] := dp_vec[i];
      end
```

In FORTRAN, the resulting operation is

```
      do 10 i=1, length
            sp_vec(i) = sngl(dp_vec(i))
 10   continue
```

dp_vec The double-precision vector to copy from.

sp_vec The single-precision vector to copy to.

length The number of elements to copy.

NOTES

In C and Pascal, **vec_$dp_sp** copies a row vector; whereas in FORTRAN, it copies a column vector.

SEE ALSO

vec_$dp_sp_i, vec_$sp_dp.

NAME

vec_$dp_sp_i – copy a vector from a double-precision matrix into a single-precision matrix

SYNOPSIS (C)

```
#include <apollo/base.h>
#include <apollo/vec.h>

void vec_$dp_sp_i(
        double *dp_vec,
        long int &inc1,
        float *sp_vec,
        long int &inc2,
        long int &length)
```

SYNOPSIS (Pascal)

```
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/vec.ins.pas';

procedure vec_$dp_sp_i(
        in dp_vec: univ vec_$double_vector;
        in inc1: integer32;
        in sp_vec: univ vec_$real_vector;
        in inc2: integer32;
        in length: integer32);
```

SYNOPSIS (FORTRAN)

```
%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/vec.ins.ftn'

        parameter (nvec = 10)

        real sp_vec(nvec)
        double precision dp_vec(nvec)
        integer*4 length, inc1, inc2

        call vec_$dp_sp_i(dp_vec, inc1, sp_vec, inc2, length)
```

DESCRIPTION

Vec_$dp_sp_i copies elements from a double-precision array *dp_vec* selected by *inc1* to elements of a single-precision array *sp_vec* selected by *inc2*.

Through appropriate choice of *inc1* and *inc2*, a program can use **vec_$dp_sp_i** to copy a vector from one matrix to another. To copy the *M*th vector in a matrix, choose *inc1* equal to the number of vectors in the matrix, and place the *M*th element of the matrix array at the beginning of *dp_vec*. To place the copy into the *N*th vector of a matrix, choose *inc2* equal to the number of vectors in the resultant matrix, and place the *N*th element of the matrix array at the beginning of *sp_vec*.

In C, the resulting operation is

```
j = 0;
k = 0;
for (i = 0; i < length; ++i) {
      sp_vec[i] = (float)dp_vec[i];
      j += inc1;
      k += inc2;
}
```

In Pascal, the resulting operation is

```
j := 1;
k := 1;
for i := 1 to length do
      begin
      sp_vec[i] := dp_vec[i];
      j := j + inc1;
      k := k + inc2;
      end
```

In FORTRAN, the resulting operation is

```
      j = 1
      k = 1
      do 10 i = 1, length
           sp_vec(k) = sngl(dp_vec(j))
           j = j + inc1
           k = k + inc2
 10   continue
```

dp_vec The double-precision array to copy from.

inc1 An increment for the index of *dp_vec* that selects the elements to copy from.

sp_vec The single-precision array to copy to.

inc2 An increment for the index of *sp_vec* that selects the elements to copy to.

length The number of elements to copy from *dp_vec* to *sp_vec*.

NOTES

In C and Pascal, **vec_$dp_sp_i** copies a column vector; whereas in FORTRAN, it copies a row vector.

SEE ALSO

vec_$dp_sp, vec_$sp_dp_i.

NAME

vec_$dpostmult – multiply a double-precision vector by a 4×4 matrix

SYNOPSIS (C)

```
#include <apollo/base.h>
#include <apollo/vec.h>

void vec_$dpostmult(
        double *matrix,
        double *start_vec,
        double *result_vec)
```

SYNOPSIS (Pascal)

```
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/vec.ins.pas';

procedure vec_$dpostmult(
        in matrix: univ vec_$double_matrix;
        in start_vec: univ vec_$double_vector;
        out result_vec: univ vec_$double_vector);
```

SYNOPSIS (FORTRAN)

```
%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/vec.ins.ftn'

        double precision matrix(4, 4), start_vec(4), result_vec(4)

        call vec_$dpostmult(matrix, start_vec, result_vec)
```

DESCRIPTION

Vec_$dpostmult multiplies the 4-element vector *start_vec* by the 4×4 matrix *matrix*.

In C, **vec_$dpostmult** applies *matrix* as a right transform to a row vector *start_vec*, and the resulting operation is

```
for (j = 0; j < 4; ++j) {
      result_vec[j] = 0.0;
      for (i = 0; i < 4; ++i)
            result_vec[j] += start_vec[i]
                           * matrix[i][j];
}
```

In Pascal, **vec_$dpostmult** applies *matrix* as a right transform to a row vector *start_vec*, and the resulting operation is

```
for j := 1 to 4 do
      begin
      result_vec[j] := 0.0;
      for i := 1 to 4 do
            result_vec[j] := result_vec[j]
                           + start_vec[i]
                           * matrix[i,j];
      end
```

In FORTRAN, **vec_$dpostmult** applies *matrix* as a left transform to a column vector *start_vec*, and the resulting operation is

```
       do 10 j = 1, 4
            result_vec(j) = 0.0
            do 10 i = 1, 4
                  result_vec(j) = result_vec(j)
     &                          + start_vec(i)
     &                          * matrix(j,i)
  10   continue
```

matrix The matrix to multiply by *start_vec*.

start_vec
The vector to multiply by *matrix*.

result_vec
The product of *start_vec* and *matrix*.

NOTES

Vec_$dpremult transforms double-precision vectors from the other side.

SEE ALSO

vec_$ipostmult, vec_$ipostmult16, vec_$postmult.

NAME

vec_$dpostmultn – multiply a double-precision vector by a matrix

SYNOPSIS (C)

```
#include <apollo/base.h>
#include <apollo/vec.h>

void vec_$dpostmultn(
        double *matrix,
        double *start_vec,
        long int &m,
        long int &n,
        double *result_vec)
```

SYNOPSIS (Pascal)

```
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/vec.ins.pas';

procedure vec_$dpostmultn(
        in matrix: univ vec_$double_matrix;
        in start_vec: univ vec_$double_vector;
        in m: integer32;
        in n: integer32;
        out result_vec: univ vec_$double_vector);
```

SYNOPSIS (FORTRAN)

```
%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/vec.ins.ftn'

        integer*4 m, n
        parameter (m = 3, n = 4)

        double precision matrix(m, n), start_vec(n), result_vec(m)

        call vec_$dpostmultn(matrix, start_vec, m, n, result_vec)
```

DESCRIPTION

Vec_$dpostmultn multiplies the *n*-element vector *start_vec* by the variably dimenstioned matrix *matrix*, and supplies the resulting *m*-element vector in *result_vec*.

In C, **vec_$dpostmultn** applies the $n \times m$ matrix *matrix* as a right transform to the *m*-element row vector *start_vec*, and supplies the transformed *n*-element result in *result_vec*:

```
for (i = 0; i < m; ++i) {
      result_vec[i] = 0.0;
      for (j = 0; j < n; ++j)
            result_vec[i] += start_vec[j]
                          * matrix[j][i];
}
```

In Pascal, **vec_$dpostmultn** applies the $n \times m$ matrix *matrix* as a right transform to the m-element row vector *start_vec*, and supplies the transformed n-element result in *result_vec*:

```
for i := 1 to m do
      begin
      result_vec[i] := 0.0;
      for j := 1 to n do
            result_vec[i] := result_vec[i]
                          + start_vec[j]
                          * matrix[j,i];
      end;
```

In FORTRAN, **vec_$dpostmultn** applies the $m \times n$ matrix *matrix* as a left transform to the m-element column vector *start_vec*, and supplies the transformed n-element result in *result_vec*:

```
       do 10 i = 1, m
            result_vec(i) = 0.0
            do 10 j = 1, n
                  result_vec(i) = result_vec(i)
      &                         + start_vec(j)
      &                         * matrix(i,j)
   10  continue
```

matrix A matrix to multiply *start_vec* by.

start_vec
An n-element vector to multiply by *matrix*.

m The number of elements in *start_vec*.

n The number of elements in *result_vec*.

result_vec
An m-element vector that is the product of *matrix* and *start_vec*.

NOTES

Vec_$dpremultn transforms double-precision vectors from the other side.

SEE ALSO

vec_$ipostmultn, vec_$ipostmultn16, vec_$postmultn.

NAME

vec_$dpremult – multiply a double-precision vector by a 4×4 matrix

SYNOPSIS (C)

#include <apollo/base.h>
#include <apollo/vec.h>

void vec_$dpremult(
double **start_vec*,
double **matrix*,
double **result_vec***)**

SYNOPSIS (Pascal)

%include '/sys/ins/base.ins.pas';
%include '/sys/ins/vec.ins.pas';

procedure vec_$dpremult(
in *start_vec*: **univ vec_$double_vector;**
in *matrix*: **univ vec_$double_matrix;**
out *result_vec*: **univ vec_$double_vector);**

SYNOPSIS (FORTRAN)

%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/vec.ins.ftn'

double precision *start_vec*(**4**), *matrix*(**4,4**), *result_vec*(**4**)

call vec_$dpremult(*start_vec*, *matrix*, *result_vec***)**

DESCRIPTION

Vec_$dpremult multiplies the 4-element vector *start_vec* by the 4×4 matrix *matrix*.

In C, **vec_$dpremult** applies *matrix* as a left transform to a column vector *start_vec*, and the resulting operation is

```
for (i = 0; i < 4; ++i) {
      result_vec[i] = 0.0;
      for (j = 0; i < 4; ++j)
            result_vec[i] += start_vec[i]
                           * matrix[i][j];
}
```

In Pascal, **vec_$dpremult** applies *matrix* as a left transform to a column vector *start_vec*, and the resulting operation is

```
for i := 1 to 4 do
      begin
      result_vec[i] := 0.0;
      for j := 1 to 4 do
            result_vec[i] := result_vec[i]
                           + start_vec[i]
                           * matrix[i,j];
      end
```

In FORTRAN, **vec_$dpremult** applies *matrix* as a right transform to a row vector *start_vec*, and the resulting operation is

```
     do 10 i = 1, 4
          result_vec(i) = 0.0
          do 10 j = 1, 4
               result_vec(i) = result_vec(i)
                             + start_vec(j)
                             * matrix(j,i)
10   continue
```

start_vec
The vector to multiply by *matrix*.

matrix The matrix to multiply by *start_vec*.

result_vec
The product of *start_vec* and *matrix*.

NOTES

Vec_$dpostmult transforms double-precision vectors from the other side.

SEE ALSO

vec_$ipremult, vec_$ipremult16, vec_$premult.

NAME

vec_$dpremultn – multiply a double-precision vector by a matrix

SYNOPSIS (C)

```
#include <apollo/base.h>
#include <apollo/vec.h>

void vec_$dpremultn(
        double *start_vec,
        double *matrix,
        long int &m,
        long int &n,
        double *result_vec)
```

SYNOPSIS (Pascal)

```
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/vec.ins.pas';

procedure vec_$dpremultn(
        in start_vec: univ vec_$double_vector;
        in matrix: univ vec_$double_matrix;
        in m: integer32;
        in n: integer32;
        out result_vec: univ vec_$double_vector);
```

SYNOPSIS (FORTRAN)

```
%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/vec.ins.ftn'

        integer*4 m, n
        parameter (m = 3, n = 4)

        double precision start_vec(m), matrix(m, n), result_vec(n)

        call vec_$dpremultn(start_vec, matrix, m, n, result_vec)
```

DESCRIPTION

Vec_$dpremultn multiplies the *m*-element vector *start_vec* by the variably dimenstioned matrix *matrix*, and supplies the resulting *n*-element vector in *result_vec*.

In C, **vec_$dpremultn** applies the *n*×*m* matrix *matrix* as a left transform to the *m*-element column vector *start_vec*, and supplies the transformed *n*-element result in *result_vec*:

```
for (i = 0; i < n; ++i) {
      result_vec[i] = 0.0;
      for (j = 0; i < m; ++i)
            result_vec[i] += start_vec[j]
                           * matrix[i][j];
}
```

In Pascal, **vec_$dpremultn** applies the $n \times m$ matrix *matrix* as a left transform to the m-element column vector *start_vec*, and supplies the transformed n-element result in *result_vec*:

```
for i := 1 to n do
      begin
      result_vec[i] := 0.0;
      for j := 1 to m do
            result_vec[i] = result_vec[i]
                           + start_vec[j]
                           * matrix[i,j];
      end
```

In FORTRAN, **vec_$dpremultn** applies the $m \times n$ matrix *matrix* as a right transform to the m-element row vector *start_vec*, and supplies the transformed n-element result in *result_vec*:

```
       do 10 i = 1, n
            result_vec(i) = 0.0
            do 10 j = 1, m
                  result_vec(i) = result_vec(i)
      &                         + start_vec(j)
      &                         * matrix(j,i)
  10   continue
```

start_vec
: An m-element vector to multiply by *matrix*.

matrix A matrix to multiply *start_vec* by.

m The number of elements in *start_vec*.

n The number of elements in *result_vec*.

result_vec
: An n-element vector that is the product of *matrix* and *start_vec*.

NOTES

Vec_$dpostmultn transforms double-precision vectors from the other side.

SEE ALSO

vec_$ipremultn, vec_$ipremultn16, vec_$premultn.

NAME

vec_$dsub – subtract double-precision vectors

SYNOPSIS (C)

```
#include <apollo/base.h>
#include <apollo/vec.h>

void vec_$dsub(
        double *start_vec,
        double *sub_vec,
        long int &length,
        double *result_vec)
```

SYNOPSIS (Pascal)

```
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/vec.ins.pas';

procedure vec_$dsub(
        in start_vec: univ vec_$double_vector;
        in sub_vec: univ vec_$double_vector;
        in length: integer32;
        out result_vec: univ vec_$double_vector);
```

SYNOPSIS (FORTRAN)

```
%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/vec.ins.ftn'

        parameter (nvec = 10)

        double precision start_vec(nvec), sub_vec(nvec), result_vec(nvec)
        integer*4 length

        call vec_$dsub(start_vec, sub_vec, length, result_vec)
```

DESCRIPTION

Vec_$dsub subtracts *length* elements of *sub_vec* from *start_vec* and supplies the difference in *result_vec*.

In C, the resulting operation is

```
for (i = 0; i < length; ++i)
     result_vec[i] = start_vec[i] - sub_vec[i];
```

In Pascal, the resulting operation is

```
for i := 1 to length do
     result_vec[i] := start_vec[i] - sub_vec[i];
```

In FORTRAN, the resulting operation is

```
     do 10 i = 1, length
          result_vec(i) = start_vec(i) - sub_vec(i)
10   continue
```

start_vec
The vector to subtract *sub_vec* from.

sub_vec The vector to subtract from *start_vec*.

length The number of differences to calculate.

result_vec
The difference of *start_vec* and *sub_vec*.

NOTES
When vec_**$dsub** is used to operate on matrixes in C and Pascal, *start_vec*, *sub_vec*, and *result_vec* are row vectors; whereas in FORTRAN, they are column vectors.

SEE ALSO
vec_$dsub_i, vec_$isub, vec_$isub16, vec_$sub.

NAME

vec_$dsub_i – subtract double-precision vectors in matrixes

SYNOPSIS (C)

```
#include <apollo/base.h>
#include <apollo/vec.h>

void vec_$dsub_i(
        double *start_vec,
        long int &inc1,
        double *sub_vec,
        long int &inc2,
        long int &length,
        double *result_vec,
        long int &inc3)
```

SYNOPSIS (Pascal)

```
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/vec.ins.pas';

procedure vec_$dsub_i(
        in start_vec: univ vec_$double_vector;
        in inc1: integer32;
        in sub_vec: univ vec_$double_vector;
        in inc2: integer32;
        in length: integer32;
        out result_vec: univ vec_$double_vector;
        in inc3: integer32);
```

SYNOPSIS (FORTRAN)

```
%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/vec.ins.ftn'

      parameter (nvec = 10)

      double precision start_vec(nvec), sub_vec(nvec), result_vec(nvec)
      integer*4 length, inc1, inc2, inc3

      call vec_$dsub_i(start_vec, inc1, sub_vec, inc2,
     &                 length, result_vec, inc3)
```

DESCRIPTION

Vec_$dsub_i subtracts *length* elements of *sub_vec* selected by *inc2* from *length* elements of *start_vec* selected by *inc1*, and supplies the result in the elements of *result_vec* selected by *inc3*.

Through appropriate choice of *inc1*, *inc2*, and *inc3*, a program can use **vec_$dsub_i** to subtract individual vectors in two matrixes and place the difference in a vector of another

matrix. To subtract the *N*th vector in matrix *Y* from the *M*th vector in matrix *X*, choose *inc2* equal to the number of vectors in matrix *Y* and *inc1* equal to the number of vectors in matrix *X*. Then place the *M*th element of matrix *X* at the beginning of *start_vec*, and place the *N*th element of matrix *Y* at the beginning of *sub_vec*. To place the result of the operation in the *P*th vector of a resultant matrix, choose *inc3* equal to the number of vectors in the resultant matrix, and place the *P*th element of the matrix array at the beginning of *result_vec*.

In C, the resulting operation is

```
j = 0;
k = 0;
l = 0;
for (i = 0; i < length, ++i) {
      result_vec[j] = start_vec[k] - sub_vec[l];
      j += inc3;
      k += inc1;
      l += inc2;
}
```

In Pascal, the resulting operation is

```
j := 1;
k := 1;
l := 1;
for i := 1 to length do
      begin
      result_vec[j] := start_vec[k] - sub_vec[l];
      j := j + inc3;
      k := k + inc1;
      l := l + inc2;
      end
```

In FORTRAN, the resulting operation is

```
     j = 1
     k = 1
     l = 1
     do 10 i = 1, length
          result_vec(j) = start_vec(k) - sub_vec(l)
          j = j + inc3
          k = k + inc1
          l = l + inc2
10   continue
```

start_vec
The vector to subtract *sub_vec* from.

inc1 An increment for the index of *start_vec* that chooses which elements the elements of *sub_vec* will be subtracted from.

sub_vec The vector to subtract from *start_vec*.

inc2 An increment for the index of *sub_vec* that chooses which elements will be subtracted from the elements of *start_vec*.

length The number of scalar differences to calculate.

result_vec
The difference of *start_vec* and *sub_vec*.

inc3 An increment for the index of *result_vec* that chooses which elements will received the differences.

NOTES

In C and Pascal, **vec_$dsub_i** operates on column vectors; whereas in FORTRAN, it operates on row vectors.

SEE ALSO

vec_$dsub, vec_$isub16_i, vec_$isub_i, vec_$sub_i.

NAME

vec_$dsum – sum the elements of a double-precision vector

SYNOPSIS (C)

#include <apollo/base.h>
#include <apollo/vec.h>

double vec_$dsum(
double **vec*,
long int &*length***)**

SYNOPSIS (Pascal)

%include '/sys/ins/base.ins.pas';
%include '/sys/ins/vec.ins.pas';

function vec_$dsum(
in *vec*: **univ vec_$double_vector;**
in *length*: **integer32): double;**

SYNOPSIS (FORTRAN)

%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/vec.ins.ftn'

parameter (*nvec* = **10**)

double precision *vec*(*nvec*), *sum*
integer*4 *length*

sum = **vec_$dsum**(*vec*, *length*)

DESCRIPTION

Vec_$dsum adds together *length* elements of the double-precision array *vector* and returns the sum.

In C, the resulting operation is

```
return_value = 0.0;
for (i = 0; i < length; ++i)
      return_value += vec[i];
```

In Pascal, the resulting operation is

```
return_value := 0.0;
for i := 1 to length do
      return_value := return_value + vec[i];
```

In FORTRAN, the resulting operation is

```
       vec_$dsum = 0.0
       do 10 i = 1, length
           vec_$dsum = vec_$dsum + vec(i)
  10   continue
```

vec The vector to sum.

length The number of elements in *vec* to sum.

NOTES

When **vec_$dsum** is used to operate on matrixes in C and Pascal, *vec* is a row vector; whereas in FORTRAN, it is a column vector.

SEE ALSO

vec_$dsum_i, vec_$isum, vec_$isum16, vec_$sum.

NAME

vec_$dsum_i – sum the elements of a vector in a double-precision matrix

SYNOPSIS (C)

```
#include <apollo/base.h>
#include <apollo/vec.h>

double vec_$dsum_i(
        double *vec,
        long int &inc,
        long int &length)
```

SYNOPSIS (Pascal)

```
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/vec.ins.pas';

function vec_$dsum_i(
        in vec: univ vec_$double_vector;
        in inc: integer32;
        in length: integer32): double;
```

SYNOPSIS (FORTRAN)

```
%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/vec.ins.ftn'

        parameter (nvec = 10)

        double precision vec(nvec), sum
        integer*4 length, inc

        sum = vec_$sum_i(vec, inc, length)
```

DESCRIPTION

Vec_$dsum_i adds *length* elements from the double-precision array *vector* selected by *inc* and returns the sum.

Through appropriate choice of *inc*, a program can use **vec_$dsum_i** to sum an individual vector in a matrix. To sum the *M*th vector in a matrix choose *inc* equal to the number of vectors in the matrix.

In C, the resulting operation is

```
return_value = 0.0;
j = 0;
for (i = 0; i < length; ++i) {
      return_value += vec[j];
      j += inc;
}
```

In Pascal, the resulting operation is

```
return_value := 0.0;
j := 1;
for i := 1 to length do
      begin
      return_value := return_value + vec[j];
      j := j + inc;
      end
```

In FORTRAN, the resulting operation is

```
    vec_$dsum_i = 0.0
    j = 1
    do 10 i = 1, length
         vec_$dsum_i = vec_$dsum_i + vec(j)
         j = j + inc
10  continue
```

vec An array that contains the elements to sum.

inc An increment for the index of *vec* that selects which elements of *vec* are summed.

length The number of elements in *vec* to sum.

NOTES

In C and Pascal, **vec_$dsum_i** sums a column vector; whereas in FORTRAN, it sums a row vector.

SEE ALSO

vec_$dsum, vec_$isum16_i, vec_$isum_i, vec_$sum_i.

NAME

vec_$dswap – swap two double-precision vectors

SYNOPSIS (C)

```
#include <apollo/base.h>
#include <apollo/vec.h>

void vec_$dswap(
        double *vec1,
        double *vec2,
        long int &length)
```

SYNOPSIS (Pascal)

```
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/vec.ins.pas';

procedure vec_$dswap(
        var vec1: univ vec_$double_vector;
        var vec2: univ vec_$double_vector;
        in length: integer32);
```

SYNOPSIS (FORTRAN)

```
%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/vec.ins.ftn'

        parameter (nvec = 10)

        double precision vec1(nvec), vec2(nvec)
        integer*4 length

        call vec_$dswap(vec1, vec2, length)
```

DESCRIPTION

Vec_$dswap swaps *length* elements between the double-precision vectors *vec1* and *vec2*.

In C, the resulting operation is

```
for (i = 0; i < length, ++i) {
      temp = vec1[i];
      vec1[i] = vec2[i];
      vec2[i] = temp;
}
```

In Pascal, the resulting operation is

```
for i := 1 to length do
      begin
      temp := vec1[i];
      vec1[i] := vec2[i];
      vec2[i] := temp;
      end
```

In FORTRAN, the resulting operation is

```
      do 10 i = 1, length
            temp = vec1(i)
            vec1(i) = vec2(i)
            vec2(i) = temp
 10   continue
```

vec1 The vector to be swapped with *vec2*.

vec2 The vector to be swapped with *vec1*.

length The number of elements to swap.

NOTES

When **vec_$dswap** is used to operate on matrixes in C and Pascal, *vec1* and *vec2* are row vectors; whereas in FORTRAN, they are column vectors.

SEE ALSO

vec_$dswap_i, vec_$iswap, vec_$iswap16, vec_$swap.

NAME

vec_$dswap_i – swap two vectors in a double-precision matrix

SYNOPSIS (C)

```
#include <apollo/base.h>
#include <apollo/vec.h>

void vec_$dswap_i(
        double *vec1,
        long int &inc1,
        double *vec2,
        long int &inc2,
        long int &length)
```

SYNOPSIS (Pascal)

```
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/vec.ins.pas';

procedure vec_$dswap_i(
        var vec1: univ vec_$double_vector;
        in inc1: integer32;
        var vec1: univ vec_$double_vector;
        in inc2: integer32;
        in length: integer32);
```

SYNOPSIS (FORTRAN)

```
%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/vec.ins.ftn'

      parameter (nvec = 10)

      double precision vec1(nvec), vec2(nvec)
      integer*4 length, inc1, inc2

      call vec_$dswap_i(vec1, inc1, vec2, inc2, length)
```

DESCRIPTION

Vec_$dswap_i swaps the *length* elements of *vec1* selected by *inc1* with the *length* elements of *vec2* selected by *inc2*.

Through appropriate choice of *inc1* and *inc2*, a program can use **vec_$dswap_i** to swap vectors between two matrixes. To select the *M*th vector in a matrix, choose *inc1* equal to the number of vectors in the matrix, and place the *M*th element of the matrix array at the beginning of *vec1*. To swap the selected vector with the *N*th vector of the same or another matrix, choose *inc2* equal to the number of vectors in the same or other matrix, and place the *N*th element of the matrix array at the beginning of *vec2*.

In C, the resulting operation is

```
j = 0;
k = 0;
for (i = 0; i < length, ++i) {
      temp = vec1[i];
      vec1[i] = vec2[i];
      vec2[i] = temp;
      j += inc1;
      k += inc2;
}
```

In Pascal, the resulting operation is

```
j := 1;
k := 1;
for i := 1 to length do
      begin
      temp := vec1[i];
      vec1[i] := vec2[i];
      vec2[i] := temp;
      j := j + inc1;
      k := k + inc2;
      end
```

In FORTRAN, the resulting operation is

```
      j = 1
      k = 1
      do 10 i = 1, length
            temp = vec1(j)
            vec1(j) = vec2(k)
            vec2(k) = temp
            j = j + inc1
            k = k + inc2
 10   continue
```

vec1 The vector to be swapped with *vec2*.

inc1 The increment for the index of *vec1* that selects the elements to be swapped with *vec2*.

vec2 The vector to be swapped with *vec1*.

inc2 The increment for the index of *vec2* that selects the elements to be swapped with *vec1*.

length The number of elements to swap.

NOTES

In C and Pascal, vec_$dswap_i swaps column vectors; whereas in FORTRAN, it swaps row vectors.

SEE ALSO

vec_$dswap, vec_$iswap16_i, vec_$iswap_i, vec_$swap_i.

NAME

vec_$dzero – zero a double-precision vector

SYNOPSIS (C)

```
#include <apollo/base.h>
#include <apollo/vec.h>

void vec_$dzero(
        double *vector,
        long int &length)
```

SYNOPSIS (Pascal)

```
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/vec.ins.pas';

procedure vec_$dzero(
        var vector: univ vec_$double_vector;
        in length: integer32);
```

SYNOPSIS (FORTRAN)

```
%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/vec.ins.ftn'

        parameter (nvec = 10)

        double precision vector(nvec)
        integer*4 length

        call vec_$dzero(vector, length)
```

DESCRIPTION

Vec_$dzero zeros the first *length* elements of the double-precision vector *vector*.

In C, the resulting operation is

```
for (i = 0; i < length; ++i)
      vector[i] = 0.0;
```

In Pascal, the resulting operation is

```
for i := 1 to length do
      vector[i] := 0.0;
```

In FORTRAN, the resulting operation is

```
      do 10 i = 1, length
           vec(i) = 0.0
 10   continue
```

vector The vector to be zeroed.

length The number of elements in *vector* to zero.

NOTES

In C and Pascal, **vec_$dzero** zeros row vectors; whereas in FORTRAN, it zeros column vectors.

SEE ALSO

vec_$dzero_i, vec_$izero, vec_$izero16, vec_$zero.

NAME

vec_$dzero_i – zero a vector in a double-precision matrix

SYNOPSIS (C)

```
#include <apollo/base.h>
#include <apollo/vec.h>

void vec_$dzero_i(
        double *vector,
        long int &inc,
        long int &length)
```

SYNOPSIS (Pascal)

```
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/vec.ins.pas';

procedure vec_$dzero_i(
        var vector: univ vec_$double_vector;
        in inc: integer32;
        in length: integer32)
```

SYNOPSIS (FORTRAN)

```
%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/vec.ins.ftn'

         parameter (nvec = 10)

         double precision vector(nvec)
         integer*4 length, inc

         call vec_$dzero_i(vector, inc, length)
```

DESCRIPTION

Vec_$dzero_i zeros the *length* elements of the double-precision array *vector* selected by *inc*.

Through appropriate choice of *inc*, a program can use **vec_$dzero_i** to zero a vector within a matrix. To search the *M*th vector in a matrix, choose *inc* equal to the number of vectors in the matrix, and place the *M*th element of the matrix array at the beginning of *vector*.

In C, the resulting operation is

```
j = 0;
for (i = 0; i < length; ++i) {
      vector[i] = 0.0;
      j += inc;
}
```

In Pascal, the resulting operation is

```
j := 1;
for i := 1 to length do
      begin
      vector[i] := 0.0;
      j := j + inc;
      end
```

In FORTRAN, the resulting operation is

```
      j = 1
      do 10 i = 1, length
            vec(j) = 0.0
            j = j + inc
 10   continue
```

vector The vector to be zeroed.

inc An increment for the index of *vector* that chooses the elements to be zeroed.

length The number of elements in *vector* to zero.

NOTES

In C and Pascal, **vec_$dzero_i** zeros a column vector; whereas in FORTRAN, it zeros a row vector.

SEE ALSO

vec_$dzero, vec_$izero16_i, vec_$izero_i, vec_$zero_i.

NAME

vec_$iadd_constant – add a scalar constant to a 32-bit integer vector

SYNOPSIS (C)

#include <apollo/base.h>
#include <apollo/vec.h>

void vec_$iadd_constant(
long **start_vec*,
long int *&length*,
long *&constant*,
long **result_vec*)

SYNOPSIS (Pascal)

%include '/sys/ins/base.ins.pas';
%include '/sys/ins/vec.ins.pas';

procedure vec_$iadd_constant(
in *start_vec*: **univ vec_$integer32_vector;**
in *length*: **integer32;**
in *constant*: **integer32;**
out *result_vec*: **univ vec_$integer32_vector);**

SYNOPSIS (FORTRAN)

%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/vec.ins.ftn'

parameter (*nvec* = **256**)

integer*4 *start_vec(nvec)*, *result_vec(nvec)*
integer*4 *constant*
integer*4 *length*

call vec_$iadd_constant(*start_vec*, *length*, *constant*, *result_vec*)

DESCRIPTION

Vec_$iadd_constant adds the scalar *constant* to the 32-bit integer vector *start_vec*, and supplies the result in *result_vec*.

In C, the resulting operation is

```
for (i = 0; i < length; ++i)
     result_vec[i] = constant + start_vec[i];
```

In Pascal, the resulting operation is

```
for i := 1 to length do
      begin
      result_vec[i] := constant + start_vec[i];
      end
```

In FORTRAN, the resulting operation is

```
       do 10 i = 1, length
            result_vec(i) = constant + start_vec(i)
   10  continue
```

start_vec
The operand vector that *constant* will be added to.

length The number of elements in *start_vec* that *constant* will be added to.

constant
A scalar constant to be added to *start_vec*.

result_vec
The vector resulting from adding *constant* to *start_vec*.

NOTES

When **vec_$iadd_constant** is used to operate on matrixes in C and Pascal, *start_vec* and *result_vec* are row vectors; whereas in FORTRAN, they are column vectors.

SEE ALSO

vec_$add_constant, vec_$add_constant_i, vec_$dadd_constant, vec_$iadd_constant16.

NAME

vec_$iadd_constant16 – add a scalar constant to a 16-bit integer vector

SYNOPSIS (C)

#include <apollo/base.h>
#include <apollo/vec.h>

void vec_$iadd_constant16(
short **start_vec*,
long int *&length*,
short *&constant*,
short **result_vec*)

SYNOPSIS (Pascal)

%include '/sys/ins/base.ins.pas';
%include '/sys/ins/vec.ins.pas';

procedure vec_$iadd_constant16(
in *start_vec*: **univ vec_$integer16_vector;**
in *length*: **integer32;**
in *constant*: **integer16;**
out *result_vec*: **univ vec_$integer16_vector);**

SYNOPSIS (FORTRAN)

%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/vec.ins.ftn'

parameter (*nvec* = **256**)

integer*4 *start_vec(nvec), result_vec(nvec)*
integer*4 *constant*
integer*4 *length*

call vec_$iadd_constant16(*start_vec, length, constant, result_vec***)**

DESCRIPTION

Vec_$iadd_constant16 adds the scalar *constant* to the 16-bit integer vector *start_vec*, and supplies the result in *result_vec*.

In C, the resulting operation is

```
for (i = 0; i < length; ++i)
     result_vec[i] = constant + start_vec[i];
```

In Pascal, the resulting operation is

```
for i := 1 to length do
      begin
      result_vec[i] := constant + start_vec[i];
      end
```

In FORTRAN, the resulting operation is

```
       do 10 i = 1, length
            result_vec(i) = constant + start_vec(i)
   10  continue
```

start_vec
The operand vector that *constant* will be added to.

length The number of elements in *start_vec* that *constant* will be added to.

constant
A scalar constant to be added to *start_vec*.

result_vec
The vector resulting from adding *constant* to *start_vec*.

NOTES

When **vec_$iadd_constant16** is used to operate on matrixes in C and Pascal, *start_vec* and *result_vec* are row vectors; whereas in FORTRAN, they are column vectors.

SEE ALSO

vec_$add_constant, vec_$add_constant_i, vec_$dadd_constant, vec_$iadd_constant.

NAME

vec_$iadd_constant16_i – add a scalar to a vector in a 16-bit integer matrix

SYNOPSIS (C)

```
#include <apollo/base.h>
#include <apollo/vec.h>

void vec_$iadd_constant16_i(
        float *start_vec,
        long int &inc1,
        long int &length,
        float &constant,
        float *result_vec,
        long int &inc2)
```

SYNOPSIS (Pascal)

```
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/vec.ins.pas';

procedure vec_$iadd_constant16_i(
        in start_vec: univ vec_$real_vector;
        in inc1: integer32;
        in length: integer32;
        in constant: real;
        out result_vec: univ vec_$real_vector;
        in inc2: integer32);
```

SYNOPSIS (FORTRAN)

```
%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/vec.ins.ftn'

      parameter (nvec = 10)

      real start_vec(nvec), result_vec(nvec), constant
      integer*4 length, inc1, inc2

      call vec_$iadd_constant16_i(start_vec, inc1, length,
     &                           constant, result_vec, inc2)
```

DESCRIPTION

Vec_$iadd_constant16_i adds the scalar *constant* to elements of the 16-bit integer array *start_vec* selected by *inc1*, and puts the sums in elements of the array *result_vec* selected by *inc2*.

Through appropriate choice of *inc1* and *inc2*, a program can use **vec_$iadd_constant16_i** to operate on individual vectors in a matrix. To add *constant* to the *M*th vector in a matrix, choose *inc1* equal to the number of vectors in the matrix, and place the *M*th element of the matrix array at the beginning of *start_vec*. To place the result of the operation in the *N*th vector of a matrix, choose *inc2* equal to the number of vectors in the resultant matrix, and place the *N*th element of the matrix array at the beginning of *result_vec*.

In C, the resulting operation is

```
j = 0;
k = 0;
for (i = 0; i < length; ++i) {
      result_vec[k] = constant + start_vec[j];
      k += inc2;
      j += inc1;
}
```

In Pascal, the resulting operation is

```
j := 1;
k := 1;
for i := 1 to length do
      begin
      result_vec[k] := constant + start_vec[j];
      k := k + inc2;
      j := j + inc1;
      end
```

In FORTRAN, the resulting operation is

```
       j = 1
       k = 1
       do 10 i = 1, length
            result_vec(k) = constant + start_vec(j)
            k = k + inc2
            j = j + inc1
  10   continue
```

start_vec
The operand array whose elements will be summed with *constant*.

inc1 Increment for the index of *start_vec* used to select the elements of *start_vec* that will be summed with *constant*.

length The number of elements in *start_vec* that *constant* will be added to.

constant
The scalar value to be summed with elements of *start_vec*.

result_vec
The array resulting from summing *constant* with elements of *start_vec*.

inc2 Increment for the index of *result_vec* used to select the elements of *result_vec* that will receive the sums of *constant* with the elements of *start_vec* selected by *inc1*.

NOTES

In C and Pascal, **vec_$iadd_constant16_i** operates on column vectors; whereas in FORTRAN, it operates on row vectors.

SEE ALSO

vec_$add_constant, vec_$add_constant_i, vec_$dadd_constant_i, vec_$iadd_constant_i.

NAME

vec_$iadd_constant_i – add a scalar to a vector in a 32-bit integer matrix

SYNOPSIS (C)

```
#include &lt;apollo/base.h&gt;
#include &lt;apollo/vec.h&gt;

void vec_$iadd_constant_i(
        float *start_vec,
        long int &incl,
        long int &length,
        float &constant,
        float *result_vec,
        long int &inc2)
```

SYNOPSIS (Pascal)

```
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/vec.ins.pas';

procedure vec_$iadd_constant_i(
        in start_vec: univ vec_$real_vector;
        in incl: integer32;
        in length: integer32;
        in constant: real;
        out result_vec: univ vec_$real_vector;
        in inc2: integer32);
```

SYNOPSIS (FORTRAN)

```
%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/vec.ins.ftn'

      parameter (nvec = 10)

      real start_vec(nvec), result_vec(nvec), constant
      integer*4 length, incl, inc2

      call vec_$iadd_constant_i(start_vec, incl, length,
     &                    constant, result_vec, inc2)
```

DESCRIPTION

Vec_$iadd_constant_i adds the scalar *constant* to elements of the 32-bit integer array *start_vec* selected by *incl*, and puts the sums in elements of the array *result_vec* selected by *inc2*.

Through appropriate choice of *incl* and *inc2*, a program can use **vec_$iadd_constant_i** to operate on individual vectors in a matrix. To add *constant* to the *M*th vector in a matrix, choose *incl* equal to the number of vectors in the matrix, and place the *M*th element of the matrix array at the beginning of *start_vec*. To place the result of the operation in the *N*th vector of a matrix, choose *inc2* equal to the number of vectors in the resultant matrix, and place the *N*th element of the matrix array at the beginning of *result_vec*.

In C, the resulting operation is

```
j = 0;
k = 0;
for (i = 0; i < length; ++i) {
      result_vec[k] = constant + start_vec[j];
      k += inc2;
      j += inc1;
}
```

In Pascal, the resulting operation is

```
j := 1;
k := 1;
for i := 1 to length do
      begin
      result_vec[k] := constant + start_vec[j];
      k := k + inc2;
      j := j + inc1;
      end
```

In FORTRAN, the resulting operation is

```
      j = 1
      k = 1
      do 10 i = 1, length
            result_vec(k) = constant + start_vec(j)
            k = k + inc2
            j = j + inc1
   10 continue
```

start_vec
The operand array whose elements will be summed with *constant*.

inc1 Increment for the index of *start_vec* used to select the elements of *start_vec* that will be summed with *constant*.

length The number of elements in *start_vec* that *constant* will be added to.

constant
The scalar value to be summed with elements of *start_vec*.

result_vec
The array resulting from summing *constant* with elements of *start_vec*.

inc2 Increment for the index of *result_vec* used to select the elements of *result_vec* that will receive the sums of *constant* with the elements of *start_vec* selected by *inc1*.

NOTES

In C and Pascal, **vec_$iadd_constant_i** operates on column vectors; whereas in FORTRAN, it operates on row vectors.

SEE ALSO

vec_$add_constant, vec_$add_constant_i, vec_$dadd_constant_i, vec_$iadd_constant16_i.

NAME

vec_$iadd_vector – add two 32-bit integer vectors

SYNOPSIS (C)

```
#include <apollo/base.h>
#include <apollo/vec.h>

void vec_$iadd_vector(
        long *start_vec,
        long *add_vec,
        long int &length,
        long *result_vec)
```

SYNOPSIS (Pascal)

```
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/vec.ins.pas';

procedure vec_$iadd_vector(
        in start_vec: univ vec_$integer32_vector;
        in add_vec: univ vec_$integer32_vector;
        in length: integer32;
        out result_vec: univ vec_$integer32_vector);
```

SYNOPSIS (FORTRAN)

```
%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/vec.ins.ftn'

      parameter (nvec = 10)

      integer*4 start_vec(nvec), add_vec(nvec), result_vec(nvec)
      integer*4 length

      call vec_$iadd_vector(start_vec, add_vec, length, result_vec)
```

DESCRIPTION

Vec_$iadd_vector adds the 32-bit integer vectors *start_vec* and *add_vec*, and supplies the vector sum in *result_vec*.

In C, the resulting operation is

```
for (i = 0; i < length; ++i)
     result_vec[i] = start_vec[i] + add_vec[i];
```

In Pascal, the resulting operation is

```
for i := 1 to length do
      begin
      result_vec[i] := start_vec[i] + add_vec[i];
      end
```

In FORTRAN, the resulting operation is

```
       do 10 i = 1, length
            result_vec(i) = start_vec(i) + add_vec(i)
  10   continue
```

start_vec
An addend vector.

add_vec
An addend vector.

length Number of elements to sum. *Length* is usually just the order of the addends.

result_vec
The vector that is the sum of *start_vec* and *add_vec*.

NOTES

When **vec_$iadd_vector** is used to operate on matrixes in C and Pascal, *start_vec* and *result_vec* are row vectors; whereas in FORTRAN, they are column vectors.

SEE ALSO

vec_$add_vector, vec_$dadd_vector, vec_$iadd_vector16.

NAME

vec_$iadd_vector16 – add two 16-bit integer vectors

SYNOPSIS (C)

```
#include <apollo/base.h>
#include <apollo/vec.h>

void vec_$iadd_vector16(
        short *start_vec,
        short *add_vec,
        long int &length,
        short *result_vec)
```

SYNOPSIS (Pascal)

```
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/vec.ins.pas';

procedure vec_$iadd_vector16(
        in start_vec: univ vec_$integer16_vector;
        in add_vec: univ vec_$integer16_vector;
        in length: integer32;
        out result_vec: univ vec_$integer16_vector);
```

SYNOPSIS (FORTRAN)

```
%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/vec.ins.ftn'

        parameter (nvec = 10)

        integer*2 start_vec(nvec), add_vec(nvec), result_vec(nvec)
        integer*4 length

        call vec_$iadd_vector16(start_vec, add_vec, length, result_vec)
```

DESCRIPTION

Vec_$iadd_vector16 adds the single-precision vectors *start_vec* and *add_vec*, and supplies the vector sum in *result_vec*.

In C, the resulting operation is

```
for (i = 0; i < length; ++i)
     result_vec[i] = start_vec[i] + add_vec[i];
```

In Pascal, the resulting operation is

```
for i := 1 to length do
      begin
      result_vec[i] := start_vec[i] + add_vec[i];
      end
```

In FORTRAN, the resulting operation is

```
       do 10 i = 1, length
            result_vec(i) = start_vec(i) + add_vec(i)
  10   continue
```

start_vec
An addend vector.

add_vec
An addend vector.

length Number of elements to sum. *Length* is usually just the order of the addends.

result_vec
The vector that is the sum of *start_vec* and *add_vec*.

NOTES

When **vec_$iadd_vector16** is used to operate on matrixes in C and Pascal, *start_vec* and *result_vec* are row vectors; whereas in FORTRAN, they are column vectors.

SEE ALSO

vec_$add_vector, vec_$dadd_vector, vec_$iadd_vector.

NAME

vec_$iadd_vector16_i – add vectors in two 16-bit integer matrixes

SYNOPSIS (C)

```
#include <apollo/base.h>
#include <apollo/vec.h>

void vec_$iadd_vector16_i(
        short *start_vec,
        long int &inc1,
        short *add_vec,
        long int &inc2,
        long int &length,
        short *result_vec,
        long int &inc3)
```

SYNOPSIS (Pascal)

```
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/vec.ins.pas';

procedure vec_$iadd_vector16_i(
        in start_vec: univ vec_$integer16_vector;
        in inc1: integer32;
        in add_vec: univ vec_$integer16_vector;
        in inc2: integer32;
        in length: integer32;
        out result_vec: univ vec_$integer16_vector;
        in inc3: integer32);
```

SYNOPSIS (FORTRAN)

```
%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/vec.ins.ftn'

      parameter (nvec = 10)

      integer*2 start_vec(nvec), add_vec(nvec), result_vec(nvec)
      integer*4 length, inc1, inc2, inc3

      call vec_$iadd_vector16_i(start_vec, inc1, add_vec, inc2,
     &                          length, result_vec, inc3)
```

DESCRIPTION

Vec_$iadd_vector16_i adds elements of the array *start_vec*, selected by *inc1*, to elements of the array *add_vec*, selected by *inc1*, and puts the sums in elements of the array *result_vec* selected by *inc3*.

Through appropriate choice of *inc1*, *inc2*, and *inc3*, a program can use **vec_$iadd_vector16_i** to add individual vectors in two matrixes and place the sum in a

vector of another matrix. To add the *M*th vector in matrix *X* to the *N*th vector in matrix *Y*, choose *inc1* equal to the number of vectors in matrix *X* and *inc2* equal to the number of vectors in matrix *Y*. Then place the *M*th element of matrix *X* at the beginning of *start_vec*, and place the *N*th element of matrix *Y* at the beginning of *add_vec*. To place the result of the operation in the *P*th vector of a resultant matrix, choose *inc2* equal to the number of vectors in the resultant matrix, and place the *P*th element of the matrix array at the beginning of *result_vec*.

In C, the resulting operation is

```
j = 0;
k = 0;
m = 0;
for (i = 0; i < length; ++i) {
      result_vec[j] = start_vec[k] + add_vec[m];
      j += inc3;
      k += inc1;
      m += inc2;
}
```

In Pascal, the resulting operation is

```
j := 1;
k := 1;
m := 1;
for i := 1 to length do
      begin
      result_vec[j] := start_vec[k] + add_vec[m];
      j := j + inc3;
      k := k + inc1;
      m := m + inc2;
      end
```

In FORTRAN, the resulting operation is

```
      j = 1
      k = 1
      m = 1
      do 10 i = 1, length
           result_vec(j) = start_vec(k) + add_vec(m)
           j = j + inc3
           k = k + inc1
           m = m + inc2
 10   continue
```

start_vec
An array whose elements will be summed with elements of *add_vec*.

inc1 Increment for the index of *start_vec* used to select the elements of *start_vec* that will be summed with elements of *add_vec*.

add_vec
An array whose elements will be summed with elements of *start_vec*.

inc2 Increment for the index of *add_vec* used to select the elements of *add_vec* that will be summed with elements of *start_vec*.

length The number of elements in *start_vec* that will be summed with elements of *add_vec*.

result_vec
An array to contain the *length* sums of elements of *start_vec* and *add_vec*.

inc3 Increment for the index of *result_vec* used to select the elements of *result_vec* that will receive the sums.

NOTES

In C and Pascal, **vec_$iadd_vector16_i** operates on column vectors; whereas in FORTRAN, it operates on row vectors.

SEE ALSO

vec_$add_vector, vec_$dadd_vector_i, vec_$iadd_vector_i.

NAME

vec_$iadd_vector_i – add vectors in two 32-bit integer matrixes

SYNOPSIS (C)

```
#include <apollo/base.h>
#include <apollo/vec.h>

void vec_$iadd_vector_i(
        long *start_vec,
        long int &inc1,
        long *add_vec,
        long int &inc2,
        long int &length,
        long *result_vec,
        long int &inc3)
```

SYNOPSIS (Pascal)

```
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/vec.ins.pas';

procedure vec_$iadd_vector_i(
        in start_vec: univ vec_$integer32_vector;
        in inc1: integer32;
        in add_vec: univ vec_$integer32_vector;
        in inc2: integer32;
        in length: integer32;
        out result_vec: univ vec_$integer32_vector;
        in inc3: integer32);
```

SYNOPSIS (FORTRAN)

```
%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/vec.ins.ftn'

      parameter (nvec = 10)

      integer*4 start_vec(nvec), add_vec(nvec), result_vec(nvec)
      integer*4 length, inc1, inc2, inc3

      call vec_$iadd_vector_i(start_vec, inc1, add_vec, inc2,
     &                        length, result_vec, inc3)
```

DESCRIPTION

Vec_$iadd_vector_i adds elements of the array *start_vec*, selected by *inc1*, to elements of the array *add_vec*, selected by *inc1*, and puts the sums in elements of the array *result_vec* selected by *inc3*.

Through appropriate choice of *inc1*, *inc2*, and *inc3*, a program can use **vec_$iadd_vector_i** to add individual vectors in two matrixes and place the sum in a

vector of another matrix. To add the *M*th vector in matrix *X* to the *N*th vector in matrix *Y*, choose *inc1* equal to the number of vectors in matrix *X* and *inc2* equal to the number of vectors in matrix *Y*. Then place the *M*th element of matrix *X* at the beginning of *start_vec*, and place the *N*th element of matrix *Y* at the beginning of *add_vec*. To place the result of the operation in the *P*th vector of a resultant matrix, choose *inc2* equal to the number of vectors in the resultant matrix, and place the *P*th element of the matrix array at the beginning of *result_vec*.

In C, the resulting operation is

```
j = 0;
k = 0;
m = 0;
for (i = 0; i < length; ++i) {
      result_vec[j] = start_vec[k] + add_vec[m];
      j += inc3;
      k += inc1;
      m += inc2;
}
```

In Pascal, the resulting operation is

```
j := 1;
k := 1;
m := 1;
for i := 1 to length do
      begin
      result_vec[j] := start_vec[k] + add_vec[m];
      j := j + inc3;
      k := k + inc1;
      m := m + inc2;
      end
```

In FORTRAN, the resulting operation is

```
      j = 1
      k = 1
      m = 1
      do 10 i = 1, length
           result_vec(j) = start_vec(k) + add_vec(m)
           j = j + inc3
           k = k + inc1
           m = m + inc2
 10   continue
```

start_vec
An array whose elements will be summed with elements of *add_vec*.

inc1 Increment for the index of *start_vec* used to select the elements of *start_vec* that will be summed with elements of *add_vec*.

add_vec
An array whose elements will be summed with elements of *start_vec*.

inc2 Increment for the index of *add_vec* used to select the elements of *add_vec* that will be summed with elements of *start_vec*.

length The number of elements in *start_vec* that will be summed with elements of *add_vec*.

result_vec
An array to contain the *length* sums of elements of *start_vec* and *add_vec*.

inc3 Increment for the index of *result_vec* used to select the elements of *result_vec* that will receive the sums.

NOTES

In C and Pascal, **vec_$iadd_vector_i** operates on column vectors; whereas in FORTRAN, it operates on row vectors.

SEE ALSO

vec_$add_vector_i, vec_$dadd_vector_i, vec_$iadd_vector16_i.

NAME

vec_$icopy – copy a 32-bit integer vector

SYNOPSIS (C)

```
#include <apollo/base.h>
#include <apollo/vec.h>

void vec_$icopy(
        long *start_vec,
        long *result_vec,
        long int &length)
```

SYNOPSIS (Pascal)

```
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/vec.ins.pas';

procedure vec_$icopy(
        in start_vec: univ vec_$integer32_vector;
        out result_vec: univ vec_$integer32_vector;
        in length: integer32);
```

SYNOPSIS (FORTRAN)

```
%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/vec.ins.ftn'

        parameter (nvec = 10)

        integer*4 start_vec(nvec), result_vec(nvec)
        integer*4 length

        call vec_$icopy(start_vec, result_vec, length)
```

DESCRIPTION

Vec_$icopy copies *length* elements from *start_vec* to *result_vec*.

In C, the resulting operation is

```
for (i = 0; i < length; ++i)
      result_vec[i] = start_vec[i];
```

In Pascal, the resulting operation is

```
for i := 1 to length do
      begin
      result_vec[i] := start_vec[i];
      end
```

In FORTRAN, the resulting operation is

```
        do 10 i = 1, length
              result_vec(i) = start_vec(i)
  10    continue
```

start_vec
The vector that *result_vec* will be copied from.

result_vec
The vector that *start_vec* will be copied to.

length The number of elements to copy from *start_vec* to *result_vec*.

NOTES

When **vec_$icopy** is used to operate on matrixes in C and Pascal, *start_vec* and *result_vec* are row vectors; whereas in FORTRAN, they are column vectors.

SEE ALSO

vec_$copy, vec_$dcopy, vec_$icopy_i, vec_$icopy16.

NAME

vec_$icopy16 – copy a 16-bit integer vector

SYNOPSIS (C)

```
#include &lt;apollo/base.h>
#include &lt;apollo/vec.h>

void vec_$icopy16(
        short *start_vec,
        short *result_vec,
        long int &length)
```

SYNOPSIS (Pascal)

```
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/vec.ins.pas';

procedure vec_$icopy16(
        in start_vec: univ vec_$integer16_vector;
        out result_vec: univ vec_$integer16_vector;
        in length: integer32);
```

SYNOPSIS (FORTRAN)

```
%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/vec.ins.ftn'

          parameter (nvec = 10)

          integer*2 start_vec(nvec), result_vec(nvec)
          integer*4 length

          call vec_$icopy16(start_vec, result_vec, length)
```

DESCRIPTION

Vec_$icopy16 copies *length* elements from *start_vec* to *result_vec*.

In C, the resulting operation is

```
for (i = 0; i < length; ++i)
      result_vec[i] = start_vec[i];
```

In Pascal, the resulting operation is

```
for i := 1 to length do
      begin
      result_vec[i] := start_vec[i];
      end
```

In FORTRAN, the resulting operation is

```
      do 10 i = 1, length
           result_vec(i) = start_vec(i)
 10   continue
```

start_vec
The vector that *result_vec* will be copied from.

result_vec
The vector that *start_vec* will be copied to.

length The number of elements to copy from *start_vec* to *result_vec*.

NOTES

When **vec_$icopy16** is used to operate on matrixes in C and Pascal, *start_vec* and *result_vec* are row vectors; whereas in FORTRAN, they are column vectors.

SEE ALSO

vec_$copy, vec_$dcopy, vec_$icopy, vec_$icopy16_i.

NAME

vec_$icopy16_i – copy a vector from one 16-bit integer matrix to another

SYNOPSIS (C)

```
#include <apollo/base.h>
#include <apollo/vec.h>

void vec_$icopy16_i(
        short *start_vec,
        long int &inc1,
        short *result_vec,
        long int &inc2,
        long int &length)
```

SYNOPSIS (Pascal)

```
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/vec.ins.pas';

procedure vec_$icopy16_i(
        in start_vec: univ vec_$integer16_vector;
        in inc1: integer32;
        out result_vec: univ vec_$integer16_vector;
        in inc2: integer32;
        in length: integer32);
```

SYNOPSIS (FORTRAN)

```
%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/vec.ins.ftn'

      parameter (nvec = 10)

      integer*2 start_vec(nvec), result_vec(nvec)
      integer*4 length, inc1, inc2

      call vec_$icopy16_i(start_vec, inc1, result_vec, inc2, length)
```

DESCRIPTION

Vec_$icopy16_i copies *length* elements of the 16-bit integer array *start_vec* selected by *inc1* into elements of *result_vec* selected by *inc2*.

Through appropriate choice of *inc1* and *inc2*, a program can use **vec_$icopy16_i** to copy a vector from one matrix to another. To copy the *M*th vector in a matrix, choose *inc1* equal to the number of vectors in the matrix, and place the *M*th element of the matrix array at the beginning of *start_vec*. To place the copy into the *N*th vector of a matrix, choose *inc2* equal to the number of vectors in the resultant matrix, and place the *N*th element of the matrix array at the beginning of *result_vec*.

In C, the resulting operation is

```
j = 0;
k = 0;
for (i = 0; i < length; ++i) {
      result_vec[k] = start_vec[j];
      k += inc2;
      j += inc1;
}
```

In Pascal, the resulting operation is

```
j := 1;
k := 1;
for i := 1 to length do
      begin
      result_vec[k] := start_vec[j];
      k := k + inc2;
      j := j + inc1;
      end
```

In FORTRAN, the resulting operation is

```
      j = 1
      k = 1
      do 10 i = 1, length
           result_vec(j) = start_vec(k)
           j = j + inc2
           k = k + inc1
 10   continue
```

start_vec
: The array whose elements will be copied to *result_vec*.

inc1
: Increment for the index of *start_vec* used to select the elements of *start_vec* that will be copied to *result_vec*.

result_vec
: The array resulting from copying elements of *start_vec* selected by *inc1* into elements of *result_vec* selected by *inc2*.

inc2
: Increment for the index of *result_vec* used to select the elements of *result_vec* that will receive the copies of the elements of *start_vec* selected by *inc1*.

length
: The number of elements in *start_vec* that will be copied to *result_vec*.

NOTES

In C and Pascal, **vec_$icopy16_i** copies column vectors; whereas in FORTRAN, it copies row vectors.

SEE ALSO

vec_$copy_i, vec_$dcopy_i, vec_$icopy16, vec_$icopy_i.

NAME

vec_$icopy_i – copy a vector from one 32-bit integer matrix to another

SYNOPSIS (C)

```
#include <apollo/base.h>
#include <apollo/vec.h>

void vec_$icopy_i(
        long *start_vec,
        long int &inc1,
        long *result_vec,
        long int &inc2,
        long int &length)
```

SYNOPSIS (Pascal)

```
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/vec.ins.pas';

procedure vec_$icopy_i(
        in start_vec: univ vec_$integer32_vector;
        in inc1: integer32;
        out result_vec: univ vec_$integer32_vector;
        in inc2: integer32;
        in length: integer32);
```

SYNOPSIS (FORTRAN)

```
%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/vec.ins.ftn'

        parameter (nvec = 10)

        integer*4 start_vec(nvec), result_vec(nvec)
        integer*4 length, inc1, inc2

        call vec_$icopy_i(start_vec, inc1, result_vec, inc2, length)
```

DESCRIPTION

Vec_$icopy_i copies *length* elements of the 32-bit integer array *start_vec* selected by *inc1* into elements of *result_vec* selected by *inc2*.

Through appropriate choice of *inc1* and *inc2*, a program can use **vec_$icopy_i** to copy a vector from one matrix to another. To copy the *M*th vector in a matrix, choose *inc1* equal to the number of vectors in the matrix, and place the *M*th element of the matrix array at the beginning of *start_vec*. To place the copy into the *N*th vector of a matrix, choose *inc2* equal to the number of vectors in the resultant matrix, and place the *N*th element of the matrix array at the beginning of *result_vec*.

In C, the resulting operation is

```
j = 0;
k = 0;
for (i = 0; i < length; ++i) {
      result_vec[k] = start_vec[j];
      k += inc2;
      j += inc1;
}
```

In Pascal, the resulting operation is

```
j := 1;
k := 1;
for i := 1 to length do
      begin
      result_vec[k] := start_vec[j];
      k := k + inc2;
      j := j + inc1;
      end
```

In FORTRAN, the resulting operation is

```
      j = 1
      k = 1
      do 10 i = 1, length
           result_vec(j) = start_vec(k)
           j = j + inc2
           k = k + inc1
  10  continue
```

start_vec
The array whose elements will be copied to *result_vec*.

inc1 Increment for the index of *start_vec* used to select the elements of *start_vec* that will be copied to *result_vec*.

result_vec
The array resulting from copying elements of *start_vec* selected by *inc1* into elements of *result_vec* selected by *inc2*.

inc2 Increment for the index of *result_vec* used to select the elements of *result_vec* that will receive the copies of the elements of *start_vec* selected by *inc1*.

length The number of elements in *start_vec* that will be copied to *result_vec*.

NOTES

In C and Pascal, **vec_$icopy_i** copies column vectors; whereas in FORTRAN, it copies row vectors.

SEE ALSO

vec_$copy_i, vec_$dcopy_i, vec_$icopy, vec_$icopy16_i.

NAME

vec_$idot – return the dot product of two 32-bit integer vectors

SYNOPSIS (C)

```
#include <apollo/base.h>
#include <apollo/vec.h>

long vec_$idot(
        long *vector1,
        long *vector2,
        long int &length)
```

SYNOPSIS (Pascal)

```
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/vec.ins.pas';

function vec_$idot(
        in vector1: univ vec_$integer32_vector;
        in vector2: univ vec_$integer32_vector;
        in length: integer32): integer32;
```

SYNOPSIS (FORTRAN)

```
%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/vec.ins.ftn'

        parameter (nvec = 10)

        integer*4 vector1(nvec), vector2(nvec), result
        integer*4 length

        result = vec_$idot(vector1, vector2, length)
```

DESCRIPTION

Vec_$idot returns the dot (scalar) product of two single-precision vectors, *vector1* and *vector2*.

In C, the resulting operation is

```
return_value = 0;
for (i = 0; i < length; ++i)
      return_value += vector1[i] * vector2[i];
```

In Pascal, the resulting operation is

```
return_value := 0;
for i := 1 to length do
      begin
      return_value := return_value
                   + vector1[i] * vector2[i];
      end
```

In FORTRAN, the resulting operation is

```
       vec_$idot = 0
       do 10 i = 1,length
            vec_$idot = vec_$idot
      &                + vector1(i) * vector2(i)
  10   continue
```

vector1 A vector.

vector2 Another vector.

length The number of elements to use in calculating the dot (scalar) product.

NOTES

When vec_$idot is used on matrixes in C or Pascal, *vector1* and *vector2* are row vectors; whereas in FORTRAN they are column vectors.

SEE ALSO

vec_$ddot, vec_$dot, vec_$idot16, vec_$idot_i.

NAME

vec_$idot16 – return the dot product of two 16-bit integer vectors

SYNOPSIS (C)

```
#include <apollo/base.h>
#include <apollo/vec.h>

short vec_$idot16(
        short *vector1,
        short *vector2,
        long int &length)
```

SYNOPSIS (Pascal)

```
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/vec.ins.pas';

function vec_$idot16(
        in vector1: univ vec_$integer16_vector;
        in vector2: univ vec_$integer16_vector;
        in length: integer32): integer16;
```

SYNOPSIS (FORTRAN)

```
%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/vec.ins.ftn'

        parameter (nvec = 10)

        integer*2 vector1(nvec), vector2(nvec), result
        integer*4 length

        result = vec_$idot16(vector1, vector2, length)
```

DESCRIPTION

Vec_$idot16 returns the dot (scalar) product of two single-precision vectors, *vector1* and *vector2*.

In C, the resulting operation is

```
return_value = 0;
for (i = 0; i < length; ++i)
     return_value += vector1[i] * vector2[i];
```

In Pascal, the resulting operation is

```
return_value := 0;
for i := 1 to length do
      begin
      return_value := return_value
                    + vector1[i] * vector2[i];
      end
```

In FORTRAN, the resulting operation is

```
       vec_$idot16 = 0
       do 10 i = 1,length
            vec_$idot16 = vec_$idot16
     &                  + vector1(i) * vector2(i)
  10   continue
```

vector1 A vector.

vector2 Another vector.

length The number of elements to use in calculating the dot (scalar) product.

NOTES

When **vec_$idot16** is used on matrixes in C or Pascal, *vector1* and *vector2* are row vectors; whereas in FORTRAN they are column vectors.

SEE ALSO

vec_$ddot, vec_$dot, vec_$idot, vec_$idot16_i.

NAME

vec_$idot16_i – return the dot product of two vectors in 16-bit integer matrixes

SYNOPSIS (C)

```
#include <apollo/base.h>
#include <apollo/vec.h>

short vec_$idot16_i(
        short *vector1,
        long int &inc1,
        short *vector2,
        long int &inc2,
        long int &length)
```

SYNOPSIS (Pascal)

```
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/vec.ins.pas';

function vec_$idot16_i(
        in vector1: univ vec_$integer16_vector;
        in inc1: integer32;
        in vector2: univ vec_$integer16_vector;
        in inc2: integer32;
        in length: integer32): integer16;
```

SYNOPSIS (FORTRAN)

```
%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/vec.ins.ftn'

        parameter (nvec = 10)

        integer*2 vector1(nvec), vector2(nvec), result
        integer*4 length, inc1, inc2

        result = vec_$idot16_i(vector1, inc1, vector2, inc2, length)
```

DESCRIPTION

Vec_$idot16_i returns the dot (scalar) product of two vectors from the 16-bit integer arrays *vector1* and *vector2*.

In C, the resulting operation is

```
j = 0;
k = 0;
return_value = 0;
for (i = 0; i < length; ++i) {
      return_value += vector1[j] * vector2[k];
      j += inc1;
      k += inc2;
}
```

In Pascal, the resulting operation is

```
j := 1;
k := 1;
return_value := 0;
for i := 1 to length do
      begin
      return_value := return_value
                   + vector1[j] * vector2[k];
      j := j + inc1;
      k := k + inc2;
      end;
```

In FORTRAN, the resulting operation is

```
      j = 1
      k = 1
      vec_$idot16_i = 0
      do 10 i = 1,length
           vec_$idot16_i = vec_$idot16_i
     &                     + vector1(j) * vector2(k)
                 j = j + inc1
                 k = k + inc2
   10 continue
```

vector1 An array containing one of the vectors to use in calculating the dot product.

inc1 An increment for *vector1* that chooses which elements will be used to calculate the product.

vector2 An array containing the other vector to use in calculating the dot product.

inc2 An increment for *vector2* that chooses which elements will be used to calculate the product.

length The number of elements from *vector1* or *vector2* to use in calculating the dot product.

NOTES

When **vec_$idot16_i** is used on matrixes in C or Pascal, *vector1* and *vector2* are column vectors; whereas in FORTRAN they are row vectors.

SEE ALSO

vec_$ddot_i, vec_$dot_i, vec_$idot16, vec_$idot_i.

NAME

vec_$idot_i – return the dot product of two vectors in 32-bit integer matrixes

SYNOPSIS (C)

```
#include <apollo/base.h>
#include <apollo/vec.h>

long vec_$idot_i(
        long *vector1,
        long int &inc1,
        long *vector2,
        long int &inc2,
        long int &length)
```

SYNOPSIS (Pascal)

```
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/vec.ins.pas';

function vec_$idot_i(
        in vector1: univ vec_$integer32_vector;
        in inc1: integer32;
        in vector2: univ vec_$integer32_vector;
        in inc2: integer32;
        in length: integer32): integer32;
```

SYNOPSIS (FORTRAN)

```
%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/vec.ins.ftn'

        parameter (nvec = 10)

        integer*4 vector1(nvec), vector2(nvec), result
        integer*4 length, inc1, inc2

        result = vec_$idot_i(vector1, inc1, vector2, inc2, length)
```

DESCRIPTION

Vec_$idot_i returns the dot (scalar) product of two vectors from the 32-bit integer arrays *vector1* and *vector2*.

In C, the resulting operation is

```
j = 0;
k = 0;
return_value = 0;
for (i = 0; i < length; ++i) {
      return_value += vector1[j] * vector2[k];
      j += inc1;
      k += inc2;
}
```

In Pascal, the resulting operation is

```
j := 1;
k := 1;
return_value := 0;
for i := 1 to length do
      begin
      return_value := return_value
                   + vector1[j] * vector2[k];
      j := j + inc1;
      k := k + inc2;
      end;
```

In FORTRAN,
the resulting operation is

```
      j = 1
      k = 1
      vec_$idot_i = 0
      do 10 i = 1,length
           vec_$idot_i = vec_$idot_i
     &                 + vector1(j) * vector2(k)
                 j = j + inc1
                 k = k + inc2
  10  continue
```

vector1 An array containing one of the vectors to use in calculating the dot product.

inc1 An increment for *vector1* that chooses which elements will be used to calculate the product.

vector2 An array containing the other vector to use in calculating the dot product.

inc2 An increment for *vector2* that chooses which elements will be used to calculate the product.

length The number of elements from *vector1* or *vector2* to use in calculating the dot product.

NOTES

When vec_$idot_i is used on matrixes in C or Pascal, *vector1* and *vector2* are column vectors; whereas in FORTRAN they are row vectors.

SEE ALSO

vec_$ddot_i, vec_$dot_i, vec_$idot, vec_$idot16_i.

NAME

vec_$iinit – initialize a 32-bit integer vector

SYNOPSIS (C)

```
#include <apollo/base.h>
#include <apollo/vec.h>

void vec_$iinit(
        long *vector,
        long int &length,
        long &constant)
```

SYNOPSIS (Pascal)

```
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/vec.ins.pas';

procedure vec_$iinit(
        var vector: univ vec_$integer32_vector;
        in length: integer32;
        in constant: integer32);
```

SYNOPSIS (FORTRAN)

```
%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/vec.ins.ftn'

        parameter (nvec = 10)

        integer*4 vector(nvec), constant
        integer*4 length

        call vec_$iinit(vector, length, constant)
```

DESCRIPTION

Vec_$iinit sets *length* elements of *vector* to the 32-bit integer value *constant*.

In C, the resulting operation is

```
for (i = 0; i < length; ++i)
      vector[i] = constant;
```

In Pascal, the resulting operation is

```
for i := 1 to length do
      begin
      vector[i] := constant;
      end
```

In FORTRAN, the resulting operation is

```
      do 10 i = 1, length
          vector(i) = constant
 10   continue
```

vector The vector to initialize.

length The number of elements in *vector* to initialize.

constant
The value that the elements of *vector* should be set to.

NOTES

In C and Pascal, vec_**$iinit** initializes a row vector; whereas in FORTRAN, it initializes a column vector.

SEE ALSO

vec_$dinit, vec_$iinit16, vec_$iinit_i, vec_$init.

NAME

vec_$iinit16 – initialize a 16-bit integer vector

SYNOPSIS (C)

```
#include <apollo/base.h>
#include <apollo/vec.h>

void vec_$iinit16(
        short *vector,
        long int &length,
        short &constant)
```

SYNOPSIS (Pascal)

```
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/vec.ins.pas';

procedure vec_$iinit16(
        var vector: univ vec_$integer16_vector;
        in length: integer32;
        in constant: integer16);
```

SYNOPSIS (FORTRAN)

```
%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/vec.ins.ftn'

        parameter (nvec = 10)

        integer*2 vector(nvec), constant
        integer*4 length

        call vec_$iinit16(vector, length, constant)
```

DESCRIPTION

Vec_$iinit16 sets *length* elements of *vector* to the 16-bit integer value *constant*.

In C, the resulting operation is

```
for (i = 0; i < length; ++i)
      vector[i] = constant;
```

In Pascal, the resulting operation is

```
for i := 1 to length do
      begin
      vector[i] := constant;
      end
```

In FORTRAN, the resulting operation is

```
      do 10 i = 1, length
          vector(i) = constant
 10   continue
```

vector The vector to initialize.

length The number of elements in *vector* to initialize.

constant
The value that the elements of *vector* should be set to.

NOTES

In C and Pascal, **vec_$iinit16** initializes a row vector; whereas in FORTRAN, it initializes a column vector.

SEE ALSO

vec_$dinit, vec_$iinit16, vec_$iinit_i, vec_$init.

NAME

vec_$imat_mult – multiply two 4×4 32-bit integer matrixes

SYNOPSIS (C)

```
#include <apollo/base.h>
#include <apollo/vec.h>

void vec_$imat_mult(
        long *matrix1,
        long *matrix2,
        long *out_matrix)
```

SYNOPSIS (Pascal)

```
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/vec.ins.pas';

procedure vec_$imat_mult(
        in matrix1: univ vec_$integer32_matrix;
        in matrix2: univ vec_$integer32_matrix;
        out out_matrix: univ vec_$integer32_matrix);
```

SYNOPSIS (FORTRAN)

```
%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/vec.ins.ftn'

        integer*4 matrix1(4, 4), matrix2(4, 4), out_matrix(4, 4)

        call vec_$imat_mult(matrix1, matrix2, out_matrix)
```

DESCRIPTION

Vec_$imat_mult multiplies two 4×4 matrixes, *matrix1* and *matrix2*, and supplies the result in *out_matrix.*

In C, vec_**$imat_mult** calculates the product of *matrix2* on the left and *matrix1* on the right, and the resulting operation is

```
for (i = 0; i < 4; ++i)
      for (j = 0; j < 4; ++j) {
            out_mat[j,i] = 0;
            for (k = 0; k < 4; ++k)
                  out_mat[j][i] += matrix1[k][i]
                                 * matrix2[j][k];
      }
```

In Pascal, vec_**$imat_mult** calculates the product of *matrix2* on the left and *matrix1* on the right, and the resulting operation is

```
for i := 1 to 4 do
      for j := 1 to 4 do
            begin
            out_mat[j,i] := 0;
            for k := 1 to 4 do
                  out_mat[j,i] := out_mat[j,i]
                               + matrix1[k,i]
                               * matrix2[j,k];
            end;
```

In FORTRAN, **vec_$imat_mult** calculates the product of *matrix1* on the left and *matrix2* on the right, and the resulting operation is

```
      do 10 i = 1, 4
            do 10 j = 1, 4
                  out_mat(i,j) = 0
                  do 10 k = 1, 4
                        out_mat(i,j) = out_mat(i,j)
     &                               + matrix1(i,k)
     &                               * matrix2(k,j)
 10   continue
```

matrix1 A 4×4 matrix.

matrix2 Another 4×4 matrix.

out_matrix
The product of *matrix1* and *matrix2*.

NOTES

Vec_$imat_multn performs the same operations for variably dimensioned matrixes.

SEE ALSO

vec_$dmat_mult, vec_$imat_mult16, vec_$mat_mult.

NAME

vec_$imat_mult16 – multiply two 4×4 16-bit integer matrixes

SYNOPSIS (C)

#include <apollo/base.h>
#include <apollo/vec.h>

void vec_$imat_mult16(
short **matrix1*,
short **matrix2*,
short **out_matrix*)

SYNOPSIS (Pascal)

%include '/sys/ins/base.ins.pas';
%include '/sys/ins/vec.ins.pas';

procedure vec_$imat_mult16(
in *matrix1*: **univ vec_$integer16_matrix;**
in *matrix2*: **univ vec_$integer16_matrix;**
out *out_matrix*: **univ vec_$integer16_matrix);**

SYNOPSIS (FORTRAN)

%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/vec.ins.ftn'

integer*2 *matrix1*(4, 4), *matrix2*(4, 4), *out_matrix*(4, 4)

call vec_$imat_mult16(*matrix1*, *matrix2*, *out_matrix*)

DESCRIPTION

Vec_$imat_mult16 multiplies two 4×4 matrixes, *matrix1* and *matrix2*, and supplies the result in *out_matrix*.

In C, **vec_$imat_mult16** calculates the product of *matrix2* on the left and *matrix1* on the right, and the resulting operation is

```
for (i = 0; i < 4; ++i)
       for (j = 0; j < 4; ++j) {
              out_mat[j,i] = 0;
              for (k = 0; k < 4; ++k)
                     out_mat[j][i] += matrix1[k][i]
                                    * matrix2[j][k];
       }
```

In Pascal, **vec_$imat_mult16** calculates the product of *matrix2* on the left and *matrix1* on the right, and the resulting operation is

```
for i := 1 to 4 do
      for j := 1 to 4 do
            begin
            out_mat[j,i] := 0;
            for k := 1 to 4 do
                  out_mat[j,i] := out_mat[j,i]
                               + matrix1[k,i]
                               * matrix2[j,k];
            end;
```

In FORTRAN, vec_$imat_mult16 calculates the product of *matrix1* on the left and *matrix2* on the right, and the resulting operation is

```
      do 10 i = 1, 4
            do 10 j = 1, 4
                  out_mat(i,j) = 0
                  do 10 k = 1, 4
                        out_mat(i,j) = out_mat(i,j)
     &                               + matrix1(i,k)
     &                               * matrix2(k,j)
   10  continue
```

matrix1 A 4×4 matrix.

matrix2 Another 4×4 matrix.

out_matrix
The product of *matrix1* and *matrix2*.

NOTES

Vec_$imat_multn16 performs the same operations for variably dimensioned matrixes.

SEE ALSO

vec_$dmat_mult, vec_$imat_mult, vec_$mat_mult.

NAME

vec_$imat_multn – multiply two 32-bit integer matrixes

SYNOPSIS (C)

```
#include &lt;apollo/base.h&gt;
#include &lt;apollo/vec.h&gt;

void vec_$imat_multn(
        long *matrix1,
        long *matrix2,
        long int &m,
        long int &n,
        long int &s,
        long *out_matrix)
```

SYNOPSIS (Pascal)

```
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/vec.ins.pas';

procedure vec_$imat_multn(
        in matrix1: univ vec_$integer32_matrix;
        in matrix2: univ vec_$integer32_matrix;
        in m: integer32;
        in n: integer32;
        in s: integer32;
        out out_matrix: univ vec_$integer32_matrix);
```

SYNOPSIS (FORTRAN)

```
%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/vec.ins.ftn'

        integer*4 m, n, s
        parameter (m = 3, n = 4, s = 5)

        integer*4 matrix1(m, n), matrix2(n, s), out_matrix(m, s)

        call vec_$imat_multn(matrix1, matrix2, m, n, s, out_matrix)
```

DESCRIPTION

Vec_$imat_multn multiplies two variably dimensioned matrixes, *matrix1* and *matrix2*, and supplies the result in *out_matrix*.

In C, **vec_$imat_multn** calculates the product of *matrix2* on the left and *matrix1* on the right, and the resulting operation is

```
for (i = 0; i < m; ++i)
      for (j = 0; j < s; ++j) {
            out_matrix[j][i] = 0;
            for (k = 0; k < n; ++k)
                  out_matrix[j][i] += matrix1[k][i]
                                    * matrix2[j][k];
      }
```

In Pascal, **vec_$imat_multn** calculates the product of *matrix2* on the left and *matrix1* on the right, and the resulting operation is

```
for i := 1 to m do
      for j := 1 to s do
            begin
            out_matrix[j,i] = 0;
            for k := 1 to n do
                  out_matrix[j,i] := out_matrix[j,i]
                                   + matrix1[k,i]
                                   * matrix2[j,k];
            end;
```

In FORTRAN, **vec_$imat_multn** calculates the product of *matrix1* on the left and *matrix2* on the right, and the resulting operation is

```
        do 10 i = 1, m
              do 10 j = 1, s
                    out_matrix(i,j) = 0
                    do 10 k = 1, n
                          out_matrix(i,j) = out_matrix(i,j)
      &                                   + matrix1(i,k)
      &                                   * matrix2(k,j)
   10   continue
```

matrix1 A matrix to be multiplied.

matrix2 Another matrix to be multiplied.

m, n, s The various matrix dimensions.

out_matrix
The product of *matrix1* and *matrix2*.

SEE ALSO

vec_$dmat_multn, vec_$imat_multn, vec_$mat_multn.

NAME

vec_$imat_multn16 – multiply two 16-bit integer matrixes

SYNOPSIS (C)

```
#include <apollo/base.h>
#include <apollo/vec.h>

void vec_$imat_multn16(
        short *matrix1,
        short *matrix2,
        long int &m,
        long int &n,
        long int &s,
        short *out_matrix)
```

SYNOPSIS (Pascal)

```
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/vec.ins.pas';

procedure vec_$imat_multn16(
        in matrix1: univ vec_$integer16_matrix;
        in matrix2: univ vec_$integer16_matrix;
        in m: integer32;
        in n: integer32;
        in s: integer32;
        out out_matrix: univ vec_$integer16_matrix);
```

SYNOPSIS (FORTRAN)

```
%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/vec.ins.ftn'

        integer*2 m, n, s
        parameter (m = 3, n = 4, s = 5)

        integer*4 matrix1(m, n), matrix2(n, s), out_matrix(m, s)

        call vec_$imat_multn16(matrix1, matrix2, m, n, s, out_matrix)
```

DESCRIPTION

Vec_$imat_multn16 multiplies two variably dimensioned matrixes, *matrix1* and *matrix2*, and supplies the result in *out_matrix*.

In C, **vec_$imat_multn16** calculates the product of *matrix2* on the left and *matrix1* on the right, and the resulting operation is

```
for (i = 0; i < m; ++i)
      for (j = 0; j < s; ++j) {
            out_matrix[j][i] = 0;
            for (k = 0; k < n; ++k)
                  out_matrix[j][i] += matrix1[k][i]
                                    * matrix2[j][k];
      }
```

In Pascal, **vec_$imat_multn16** calculates the product of *matrix2* on the left and *matrix1* on the right, and the resulting operation is

```
for i := 1 to m do
      for j := 1 to s do
            begin
            out_matrix[j,i] = 0;
            for k := 1 to n do
                  out_matrix[j,i] := out_matrix[j,i]
                                   + matrix1[k,i]
                                   * matrix2[j,k];
            end;
```

In FORTRAN, **vec_$imat_multn16** calculates the product of *matrix1* on the left and *matrix2* on the right, and the resulting operation is

```
     do 10 i = 1, m
          do 10 j = 1, s
               out_matrix(i,j) = 0
               do 10 k = 1, n
                    out_matrix(i,j) = out_matrix(i,j)
    &                               + matrix1(i,k)
    &                               * matrix2(k,j)
 10  continue
```

matrix1 A matrix to be multiplied.

matrix2 Another matrix to be multiplied.

m, n, s The various matrix dimensions.

out_matrix
The product of *matrix1* and *matrix2*.

SEE ALSO

vec_$dmat_multn, vec_$imat_multn, vec_$mat_multn.

NAME

vec_$imax – find the maximum absolute value in a 32-bit integer vector

SYNOPSIS (C)

```
#include &lt;apollo/base.h&gt;
#include &lt;apollo/vec.h&gt;

void vec_$imax(
        long *vector,
        long int &length,
        long *result,
        long int *location)
```

SYNOPSIS (Pascal)

```
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/vec.ins.pas';

procedure vec_$imax(
        in vector: univ vec_$integer32_vector;
        in length: integer32;
        out result: integer32;
        out location: integer32);
```

SYNOPSIS (FORTRAN)

```
%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/vec.ins.ftn'

          parameter (nvec = 10)

          integer*4 vector(nvec), result
          integer*4 length, location

          call vec_$imax(vector, length, result, location)
```

DESCRIPTION

Vec_$imax searches through *length* elements of *vector* and supplies the value and location of the element with the greatest absolute value.

In C, the resulting operation is

```
result = abs(vector[0]);
location = 1;
for (i = 1; i < length; ++i)
      if (abs(vector[i]) > result) {
            location = i + 1;
            result = abs(vector[i]);
      }
```

In Pascal, the resulting operation is

```
result := abs(vector[1]);
location = 1;
for 10 i := 2 to length do
      if (abs(vector[i]) > result) then
            begin
            location := i;
            result := abs(vector[i]);
            end
```

In FORTRAN, the resulting operation is

```
      result = iabs(vector(1))
      location = 1
      do 10 i = 2, length
           if (iabs(vector(i)) .gt. result) then
                location = i
                result = iabs(vector(i))
           endif
  10  continue
```

vector The vector to search.

length The number of elements to search.

result The maximum absolute value of all the elements searched.

location The location of the element with the greatest absolute value. The location supplied in *location* is just the index of the element with the greatest absolute value in FORTRAN or Pascal (if *vector* is declared to begin with index 1). In C, *location* - 1 is the index of the element.

NOTES

In C and Pascal, **vec_$imax** searches a row vector; whereas in FORTRAN, it searches a column vector.

SEE ALSO

vec_$dmax, vec_$imax_i, vec_$imax16, vec_$max.

NAME

vec_$imax16 – find the maximum absolute value in a 16-bit integer vector

SYNOPSIS (C)

```
#include <apollo/base.h>
#include <apollo/vec.h>

void vec_$imax16(
        short *vector,
        long int &length,
        short *result,
        long int *location)
```

SYNOPSIS (Pascal)

```
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/vec.ins.pas';

procedure vec_$imax16(
        in vector: univ vec_$integer16_vector;
        in length: integer32;
        out result: integer16;
        out location: integer32);
```

SYNOPSIS (FORTRAN)

```
%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/vec.ins.ftn'

        parameter (nvec = 10)

        integer*2 vector(nvec), result
        integer*4 length, location

        call vec_$imax16(vector, length, result, location)
```

DESCRIPTION

Vec_$imax16 searches through *length* elements of *vector* and supplies the value and location of the element with the greatest absolute value.

In C, the resulting operation is

```
result = (short)abs(vector[0]);
location = 1;
for (i = 1; i < length; ++i)
      if ((short)abs(vector[i]) > result) {
            location = i + 1;
            result = (short)abs(vector[i]);
      }
```

In Pascal, the resulting operation is

```
result := abs(vector[1]);
location = 1;
for 10 i := 2 to length do
      if (abs(vector[i]) > result) then
            begin
            location := i;
            result := abs(vector[i]);
            end
```

In FORTRAN, the resulting operation is

```
      result = iabs(vector(1))
      location = 1
      do 10 i = 2, length
           if (iabs(vector(i)) .gt. result) then
                 location = i
                 result = iabs(vector(i))
           endif
10    continue
```

vector The vector to search.

length The number of elements to search.

result The maximum absolute value of all the elements searched.

location The location of the element with the greatest absolute value. The location supplied in *location* is just the index of the element with the greatest absolute value in FORTRAN or Pascal (if *vector* is declared to begin with index 1). In C, *location* - 1 is the index of the element.

NOTES

In C and Pascal, **vec_$imax16** searches a row vector; whereas in FORTRAN, it searches a column vector.

SEE ALSO

vec_$dmax, vec_$imax, vec_$imax16_i, vec_$max.

NAME

vec_$imax16_i – find the maximum absolute value in a vector from a 16-bit integer matrix

SYNOPSIS (C)

```
#include <apollo/base.h>
#include <apollo/vec.h>

void vec_$imax16_i(
        short *vector,
        long int &inc,
        long int &length,
        short *result,
        long int *location)
```

SYNOPSIS (Pascal)

```
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/vec.ins.pas';

procedure vec_$imax16_i(
        in vector: univ vec_$integer16_vector;
        in inc: integer32;
        in length: integer32;
        out result: integer16;
        out location: integer32);
```

SYNOPSIS (FORTRAN)

```
%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/vec.ins.ftn'

        parameter (nvec = 10)

        integer*2 vector(nvec), result
        integer*4 length, inc, location

        call vec_$imax16_i(vector, inc, length, result, location)
```

DESCRIPTION

Vec_$imax16_i searches through the *length* elements of *vector* selected by *inc*, and supplies the value and location of the element with the greatest absolute value.

Through appropriate choice of *inc*, a program can use **vec_$imax16_i** to search a vector within a matrix. To search the *M*th vector in a matrix, choose *inc* equal to the number of vectors in the matrix, and place the *M*th element of the matrix array at the beginning of *vector*.

In C, the resulting operation is

```
result = (short)abs(vector[0]);
location = 1;
j = inc;
for (i = 1; i < length, ++i) {
      if ((short)abs(vector[j]) > result) {
            location = i + 1;
            result = (short)abs(vector[j]);
      }
      j += inc;
}
```

In Pascal, the resulting operation is

```
result := abs(vector[1]);
location := 1;
j := 1 + inc;
for 10 i := 2 to length do
      begin
      if (abs(vector[j]) > result) then
            begin
            location := i;
            result := abs(vector[j]);
            end
      j := j + inc;
      end
```

In FORTRAN, the resulting operation is

```
       result = iabs(vector(1))
       location = 1
       j = 1 + inc
       do 10 i = 2, length
            if (iabs(vector(j)) .gt. result) then
                  location = i
                  result = iabs(vector(j))
            endif
       j = j + inc
  10   continue
```

vector The array to search.

inc An increment for the index of *vector* that selects the elements to search.

length The number of elements to search.

result The maximum absolute value of all the elements searched.

location The location of the element with the greatest absolute value. The location supplied in *location* is just the index of the element with the greatest absolute value in FORTRAN or Pascal (if *vector* is declared to begin with index 1). In C, *location* - 1 is the index of the element.

NOTES

In C and Pascal, **vec_$imax16_i** searches a column vector; whereas in FORTRAN, it searches a row vector.

SEE ALSO

vec_$dmax_i, vec_$imax16, vec_$imax_i, vec_$max_i.

NAME

vec_$imax_i – find the maximum absolute value in a vector from a 32-bit integer matrix

SYNOPSIS (C)

```
#include <apollo/base.h>
#include <apollo/vec.h>

void vec_$imax_i(
        long *vector,
        long int &inc,
        long int &length,
        long *result,
        long int *location)
```

SYNOPSIS (Pascal)

```
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/vec.ins.pas';

procedure vec_$imax_i(
        in vector: univ vec_$integer32_vector;
        in inc: integer32;
        in length: integer32;
        out result: integer32;
        out location: integer32);
```

SYNOPSIS (FORTRAN)

```
%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/vec.ins.ftn'

        parameter (nvec = 10)

        integer*4 vector(nvec), result
        integer*4 length, inc, location

        call vec_$imax_i(vector, inc, length, result, location)
```

DESCRIPTION

Vec_$imax_i searches through the *length* elements of *vector* selected by *inc*, and supplies the value and location of the element with the greatest absolute value.

Through appropriate choice of *inc*, a program can use **vec_$imax_i** to search a vector within a matrix. To search the *M*th vector in a matrix, choose *inc* equal to the number of vectors in the matrix, and place the *M*th element of the matrix array at the beginning of *vector*.

In C, the resulting operation is

```
result = abs(vector[0]);
location = 1;
j = inc;
for (i = 1; i < length, ++i) {
      if (abs(vector[j]) > result) {
            location = i + 1;
            result = abs(vector[j]);
      }
      j += inc;
}
```

In Pascal, the resulting operation is

```
result := abs(vector[1]);
location := 1;
j := 1 + inc;
for 10 i := 2 to length do
      begin
      if (abs(vector[j]) > result) then
            begin
            location := i;
            result := abs(vector[j]);
            end
      j := j + inc;
      end
```

In FORTRAN, the resulting operation is

```
      result = iabs(vector(1))
      location = 1
      j = 1 + inc
      do 10 i = 2, length
           if (iabs(vector(j)) .gt. result) then
                location = i
                result = iabs(vector(j))
           endif
      j = j + inc
 10   continue
```

vector The array to search.

inc An increment for the index of *vector* that selects the elements to search.

length The number of elements to search.

result The maximum absolute value of all the elements searched.

location The location of the element with the greatest absolute value. The location supplied in *location* is just the index of the element with the greatest absolute value in FORTRAN or Pascal (if *vector* is declared to begin with index 1). In C, *location* - 1 is the index of the element.

NOTES

In C and Pascal, **vec_$imax_i** searches a column vector; whereas in FORTRAN, it searches a row vector.

SEE ALSO

vec_$dmax_i, vec_$imax, vec_$imax16_i, vec_$max_i.

NAME

vec_$imult_add – scale and add one 32-bit integer vector to another

SYNOPSIS (C)

```
#include <apollo/base.h>
#include <apollo/vec.h>

void vec_$imult_add(
        long *add_vec,
        long *mult_vec,
        long int &length,
        long &constant,
        long *result_vec)
```

SYNOPSIS (Pascal)

```
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/vec.ins.pas';

procedure vec_$imult_add(
        in add_vec: univ vec_$integer32_vector;
        in mult_vec: univ vec_$integer32_vector;
        in length: integer32;
        in constant: integer32;
        out result_vec: univ vec_$integer32_vector);
```

SYNOPSIS (FORTRAN)

```
%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/vec.ins.ftn'

      parameter (nvec = 10)

      integer*4 add_vec(nvec), mult_vec(nvec), result_vec(nvec), constant
      integer*4 length

      call vec_$imult_add(add_vec, mult_vec, length,
     &                     constant, result_vec)
```

DESCRIPTION

Vec_$imult_add multiplies the vector *mult_vec* by the scalar *constant*, adds the product to the vector *add_vec*, and supplies the resulting vector in *result_vec*.

In C, the resulting operation is

```
for (i = 0; i< length; ++i)
     result_vec[i] = add_vec[i]
                   + constant * mult_vec[i];
```

In Pascal, the resulting operation is

```
for i := 1 to length do
      result_vec[i] := add_vec[i]
                     + constant * mult_vec[i];
```

In FORTRAN, the resulting operation is

```
       do 10 i = 1, length
            result_vec(i) = add_vec(i)
      &                    + constant * mult_vec(i)
  10   continue
```

add_vec
: The vector to add to the product of *mult_vec* and *constant*.

mult_vec
: The vector to scale by *constant* and add to *add_vec*.

length The number of elements to use in the calculation.

constant
: The scalar value used to scale *mult_vec*.

result_vec
: The vector resulting from multiplying *mult_vec* by *constant* and adding the product to *add_vec*.

NOTES

When **vec_$imult_add** is used to operate on matrixes in C and Pascal, *add_vec*, *mult_vec*, and *result_vec* are row vectors; whereas in FORTRAN, they are column vectors.

SEE ALSO

vec_$dmult_add, vec_$imult_add16, vec_$imult_add_i, vec_$mult_add.

NAME

vec_$imult_add16 – scale and add one 16-bit integer vector to another

SYNOPSIS (C)

```
#include &lt;apollo/base.h&gt;
#include &lt;apollo/vec.h&gt;

void vec_$imult_add16(
        short *add_vec,
        short *mult_vec,
        long int &length,
        short &constant,
        short *result_vec)
```

SYNOPSIS (Pascal)

```
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/vec.ins.pas';

procedure vec_$imult_add16(
        in add_vec: univ vec_$integer16_vector;
        in mult_vec: univ vec_$integer16_vector;
        in length: integer32;
        in constant: integer16;
        out result_vec: univ vec_$integer16_vector);
```

SYNOPSIS (FORTRAN)

```
%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/vec.ins.ftn'

      parameter (nvec = 10)

      integer*2 add_vec(nvec), mult_vec(nvec), result_vec(nvec), constant
      integer*4 length

      call vec_$imult_add16(add_vec, mult_vec, length,
     &                      constant, result_vec)
```

DESCRIPTION

Vec_$imult_add16 multiplies the vector *mult_vec* by the scalar *constant*, adds the product to the vector *add_vec*, and supplies the resulting vector in *result_vec*.

In C, the resulting operation is

```
for (i = 0; i< length; ++i)
      result_vec[i] = add_vec[i]
                    + constant * mult_vec[i];
```

In Pascal, the resulting operation is

```
for i := 1 to length do
     result_vec[i] := add_vec[i]
                    + constant * mult_vec[i];
```

In FORTRAN, the resulting operation is

```
       do 10 i = 1, length
            result_vec(i) = add_vec(i)
      &                    + constant * mult_vec(i)
   10  continue
```

add_vec
: The vector to add to the product of *mult_vec* and *constant*.

mult_vec
: The vector to scale by *constant* and add to *add_vec*.

length
: The number of elements to use in the calculation.

constant
: The scalar value used to scale *mult_vec*.

result_vec
: The vector resulting from multiplying *mult_vec* by *constant* and adding the product to *add_vec*.

NOTES

When vec_**$imult_add16** is used to operate on matrixes in C and Pascal, *add_vec*, *mult_vec*, and *result_vec* are row vectors; whereas in FORTRAN, they are column vectors.

SEE ALSO

vec_$dmult_add, vec_$imult_add, vec_$imult_add16_i, vec_$mult_add.

NAME

vec_$imult_add16_i – scale and add 16-bit vectors in matrixes

SYNOPSIS (C)

```
#include <apollo/base.h>
#include <apollo/vec.h>

void vec_$imult_add16_i(
        short *add_vec,
        long int &inc1,
        short *mult_vec,
        long int &inc2,
        long int &length,
        short &constant,
        short *result_vec,
        long int &inc3)
```

SYNOPSIS (Pascal)

```
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/vec.ins.pas';

procedure vec_$imult_add16_i(
        in add_vec: univ vec_$integer16_vector;
        in inc1: integer32;
        in mult_vec: univ vec_$integer16_vector;
        in inc2: integer32;
        in length: integer32;
        in constant: integer16;
        out result_vec: univ vec_$integer16_vector;
        in inc3: integer32);
```

SYNOPSIS (FORTRAN)

```
%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/vec.ins.ftn'

        parameter (nvec = 10)

        integer*2 add_vec(nvec), mult_vec(nvec), result_vec(nvec), constant
        integer*4 length, inc1, inc2, inc3

        call vec_$imult_add16_i(add_vec, inc1, mult_vec, inc2, length,
       &                        constant, result_vec, inc3)
```

DESCRIPTION

Vec_$imult_add16_i multiplies *length* elements of *mult_vec* selected by *inc2* by the scalar *constant*, adds the resulting products to elements of *add_vec* selected by *inc1*, and supplies the results in elements of *result_vec* selected by *inc3*.

Through appropriate choice of *inc1*, *inc2*, and *inc3*, a program can use **vec_$imult_add16_i** to scale and sum individual vectors in two matrixes and place the result in a vector of another matrix. To scale and add the *N*th vector in matrix *Y* to the *M*th vector in matrix *X*, choose *inc2* equal to the number of vectors in matrix *Y* and *inc1* equal to the number of vectors in matrix *X*. Then place the *M*th element of matrix *X* at the beginning of *add_vec*, and place the *N*th element of matrix *Y* at the beginning of *mult_vec*. To place the result of the operation in the *P*th vector of a resultant matrix, choose *inc3* equal to the number of vectors in the resultant matrix, and place the *P*th element of the matrix array at the beginning of *result_vec*.

In C, the resulting operation is

```
j = 0;
k = 0;
l = 0;
for (i = 0; i< length; ++i) {
      result_vec[l] = add_vec[j]
                    + constant * mult_vec[k];
      j += inc1;
      k += inc2;
      l += inc3;
}
```

In Pascal, the resulting operation is

```
j := 1;
k := 1;
l := 1;
for i := 1 to length do
      begin
      result_vec[l] := add_vec[j]
                     + constant * mult_vec[k];
      j := j + inc1;
      k := k + inc2;
      l := l + inc3;
      end
```

In FORTRAN, the resulting operation is

```
         j = 1
         k = 1
         l = 1
         do 10 i = 1, length
              result(l) = add_vec(j)
        &                  + constant * mult_vec(k)
              j = j + inc1
              k = k + inc2
              l = l + inc3
    10   continue
```

add_vec
The vector to add to the product of *mult_vec* and *constant*.

inc1 An increment for the index of *add_vec* that selects the elements to add to the product of *mult_vec* and *constant.*•

mult_vec
The vector to scale by *constant* and add to *add_vec*.

inc2 An increment for the index of *mult_vec* that selects the elements to multiply by *constant* and add to *add_vec*.

length The number of elements to use in the calculation.

constant
The scalar value used to scale *mult_vec*.

result_vec
The vector resulting from multiplying *mult_vec* by *constant* and adding the product to *add_vec*.

inc3 An increment for the index of *result_vec* that selects the elements to receive the results of multiplying *mult_vec* by *constant* and adding the product to *add_vec*.

NOTES

In C and Pascal, vec_**$imult_add16_i** operates on column vectors; whereas in FORTRAN, it operates on row vectors.

SEE ALSO

vec_$dmult_add_i, vec_$imult_add16, vec_$imult_add_i, vec_$mult_add_i.

NAME

vec_$imult_add_i – scale and add 32-bit vectors in matrixes

SYNOPSIS (C)

```
#include <apollo/base.h>
#include <apollo/vec.h>

void vec_$imult_add_i(
        long *add_vec,
        long int &inc1,
        long *mult_vec,
        long int &inc2,
        long int &length,
        long &constant,
        long *result_vec,
        long int &inc3)
```

SYNOPSIS (Pascal)

```
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/vec.ins.pas';

procedure vec_$imult_add_i(
        in add_vec: univ vec_$integer32_vector;
        in inc1: integer32;
        in mult_vec: univ vec_$integer32_vector;
        in inc2: integer32;
        in length: integer32;
        in constant: integer32;
        out result_vec: univ vec_$integer32_vector;
        in inc3: integer32);
```

SYNOPSIS (FORTRAN)

```
%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/vec.ins.ftn'

      parameter (nvec = 10)

      integer*4 add_vec(nvec), mult_vec(nvec), result_vec(nvec), constant
      integer*4 length, inc1, inc2, inc3

      call vec_$imult_add_i(add_vec, inc1, mult_vec, inc2, length,
     &                      constant, result_vec, inc3)
```

DESCRIPTION

Vec_$imult_add_i multiplies *length* elements of *mult_vec* selected by *inc2* by the scalar *constant*, adds the resulting products to elements of *add_vec* selected by *inc1*, and supplies the results in elements of *result_vec* selected by *inc3*.

Through appropriate choice of *inc1*, *inc2*, and *inc3*, a program can use **vec_$imult_add_i** to scale and sum individual vectors in two matrixes and place the result in a vector of another matrix. To scale and add the *N*th vector in matrix *Y* to the *M*th vector in matrix *X*, choose *inc2* equal to the number of vectors in matrix *Y* and *inc1* equal to the number of vectors in matrix *X*. Then place the *M*th element of matrix *X* at the beginning of *add_vec*, and place the *N*th element of matrix *Y* at the beginning of *mult_vec*. To place the result of the operation in the *P*th vector of a resultant matrix, choose *inc3* equal to the number of vectors in the resultant matrix, and place the *P*th element of the matrix array at the beginning of *result_vec*.

In C, the resulting operation is

```
j = 0;
k = 0;
l = 0;
for (i = 0; i< length; ++i) {
      result_vec[l] = add_vec[j]
                    + constant * mult_vec[k];
      j += inc1;
      k += inc2;
      l += inc3;
}
```

In Pascal, the resulting operation is

```
j := 1;
k := 1;
l := 1;
for i := 1 to length do
      begin
      result_vec[l] := add_vec[j]
                    + constant * mult_vec[k];
      j := j + inc1;
      k := k + inc2;
      l := l + inc3;
      end
```

In FORTRAN, the resulting operation is

```
      j = 1
      k = 1
      l = 1
      do 10 i = 1, length
           result(l) = add_vec(j)
     &                  + constant * mult_vec(k)
           j = j + inc1
           k = k + inc2
           l = l + inc3
 10   continue
```

add_vec
The vector to add to the product of *mult_vec* and *constant*.

inc1 An increment for the index of *add_vec* that selects the elements to add to the product of *mult_vec* and *constant*.

mult_vec
The vector to scale by *constant* and add to *add_vec*.

inc2 An increment for the index of *mult_vec* that selects the elements to multiply by *constant* and add to *add_vec*.

length The number of elements to use in the calculation.

constant
The scalar value used to scale *mult_vec*.

result_vec
The vector resulting from multiplying *mult_vec* by *constant* and adding the product to *add_vec*.

inc3 An increment for the index of *result_vec* that selects the elements to receive the results of multiplying *mult_vec* by *constant* and adding the product to *add_vec*.

NOTES

In C and Pascal, **vec_$imult_add_i** operates on column vectors; whereas in FORTRAN, it operates on row vectors.

SEE ALSO

vec_$dmult_add_i, vec_$imult_add, vec_$imult_add16_i, vec_$mult_add_i.

NAME

vec_$imult_constant – multiply a 32-bit integer vector by a scalar

SYNOPSIS (C)

#include <apollo/base.h>
#include <apollo/vec.h>

void vec_$imult_constant(
  long **mult_vec*,
  long int *&length*,
  long *&constant*,
  long **result_vec*)

SYNOPSIS (Pascal)

%include '/sys/ins/base.ins.pas';
%include '/sys/ins/vec.ins.pas';

procedure vec_$imult_constant(
  in *start_vec*: **univ vec_$integer32_vector;**
  in *length*: **integer32;**
  in *constant*: **integer32;**
  out *result_vec*: **univ vec_$integer32_vector);**

SYNOPSIS (FORTRAN)

%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/vec.ins.ftn'

parameter (*nvec* = **10**)

integer*4 *mult_vec(nvec), result_vec(nvec), constant*
integer*4 *length*

call vec_$imult_constant(*mult_vec, length, constant, result_vec***)**

DESCRIPTION

Vec_$imult_constant multiplies the 32-bit integer array *mult_vec* by the scalar value *constant* and supplies the result in the array *result_vec*.

In C, the resulting operation is

```
for (i = 0; i < length; ++i)
     result_vec[i] = constant * start_vec[i];
```

In Pascal, the resulting operation is

```
for i := 1 to length do
      begin
      result_vec[i] := constant * start_vec[i];
      end
```

In FORTRAN, the resulting operation is

```
      do 10 i = 1, length
            result_vec(i) = constant * mult_vec(i)
  10  continue
```

mult_vec
The vector to multiply by *constant*.

length The number of elements to multiply.

constant
The scalar constant to multiply *mult_vec* by.

result_vec
The vector resulting from multiplying *mult_vec* by *constant*.

NOTES

When **vec_$imult_constant** is used to operate on matrixes in C and Pascal, *mult_vec* and *result_vec* are row vectors; whereas in FORTRAN, they are column vectors.

SEE ALSO

vec_$dmult_constant, vec_$imult_constant16, vec_$imult_constant_i, vec_$mult_constant.

NAME

vec_$imult_constant16 – multiply a 16-bit integer vector by a scalar

SYNOPSIS (C)

```
#include <apollo/base.h>
#include <apollo/vec.h>

void vec_$imult_constant16(
        short *mult_vec,
        long int &length,
        short &constant,
        short *result_vec)
```

SYNOPSIS (Pascal)

```
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/vec.ins.pas';

procedure vec_$imult_constant16(
        in mult_vec: univ vec_$integer16_vector;
        in length: integer32;
        in constant: integer16;
        out result_vec: univ vec_$integer16_vector);
```

SYNOPSIS (FORTRAN)

```
%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/vec.ins.ftn'

          parameter (nvec = 10)

          integer*2 mult_vec(nvec), result_vec(nvec), constant
          integer*4 length

          call vec_$imult_constant16(mult_vec, length, constant, result_vec)
```

DESCRIPTION

Vec_$imult_constant16 multiplies the 16-bit integer array *mult_vec* by the scalar value *constant* and supplies the result in the array *result_vec*.

In C, the resulting operation is

```
for (i = 0; i < length; ++i)
     result_vec[i] = constant * start_vec[i];
```

In Pascal, the resulting operation is

```
for i := 1 to length do
      begin
      result_vec[i] := constant * start_vec[i];
      end
```

In FORTRAN, the resulting operation is

```
      do 10 i = 1, length
           result_vec(i) = constant * mult_vec(i)
 10   continue
```

mult_vec
The vector to multiply by *constant*.

length The number of elements to multiply.

constant
The scalar constant to multiply *mult_vec* by.

result_vec
The vector resulting from multiplying *mult_vec* by *constant*.

NOTES

When **vec_$imult_constant16** is used to operate on matrixes in C and Pascal, *mult_vec* and *result_vec* are row vectors; whereas in FORTRAN, they are column vectors.

SEE ALSO

vec_$dmult_constant, vec_$imult_constant, vec_$imult_constant16_i, vec_$mult_constant.

NAME

vec_$imult_constant16_i – multiply a vector in a 16-bit integer matrix by a scalar

SYNOPSIS (C)

```
#include <apollo/base.h>
#include <apollo/vec.h>

void vec_$imult_constant16_i(
        short *mult_vec,
        long int &inc1,
        long int &length,
        short &constant,
        short *result_vec,
        long int &inc2)
```

SYNOPSIS (Pascal)

```
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/vec.ins.pas';

procedure vec_$imult_constant16_i(
        in mult_vec: univ vec_$integer16_vector;
        in inc1: integer32;
        in length: integer32;
        in constant: integer16;
        out result_vec: univ vec_$integer16_vector;
        in inc2: integer32);
```

SYNOPSIS (FORTRAN)

```
%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/vec.ins.ftn'

      parameter (nvec = 10)

      integer*2 mult_vec(nvec), result_vec(nvec), constant
      integer*4 length, inc1, inc2

      call vec_$imult_constant16_i(mult_vec, inc1, length, constant,
     &                             result_vec, inc2)
```

DESCRIPTION

Vec_$imult_constant16_i multiplies *length* elements of the 16-bit integer array *mult_vec* selected by *inc1* by the scalar value *constant*, and supplies the result in elements of the array *result_vec* selected by *inc2*.

Through appropriate choice of *inc1* and *inc2*, a program can use **vec_$imult_constant16_i** to operate on a vector in a matrix. To multiply *M*th vector in a matrix by *constant*, choose *inc1* equal to the number of vectors in the matrix, and place the *M*th element of the matrix array at the beginning of *mult_vec*. To place the result of the operation in the *N*th vector of a matrix, choose *inc2* equal to the number of vectors in the resultant matrix, and place the *N*th element of the matrix array at the beginning of *result_vec*.

In C, the resulting operation is

```
j = 0;
k = 0;
for (i = 0; i < length; ++i) {
      result_vec[k] = constant * mult_vec[j];
      k += inc2;
      j += inc1;
}
```

In Pascal, the resulting operation is

```
j := 1;
k := 1;
for i := 1 to length do
      begin
      result_vec[k] := constant * mult_vec[j];
      k := k + inc2;
      j := j + inc1;
      end
```

In FORTRAN, the resulting operation is

```
      j = 1
      k = 1
      do 10 i = 1, length
           result(j) = constant * mult_vec(k)
           j = j + inc2
           k = k + inc1
 10   continue
```

mult_vec
The vector to multiply by *constant*.

inc1 An increment for the index of the array *mult_vec* that selects elements to multiply by *constant*.

length The number of products to calculate.

constant
The scalar constant to multiply elements of *mult_vec* by.

result_vec
An array whose elements receive the product of *constant* and *mult_vec*.

inc2 An increment for the index of the array *result_vec* that selects elements to receive the product of *constant* and *mult_vec*.

NOTES

In C and Pascal, **vec_$imult_constant16_i** operates on column vectors; whereas in FORTRAN, it operates on row vectors.

SEE ALSO

vec_$dmult_constant_i, vec_$imult_constant16, vec_$imult_constant_i, vec_$mult_constant_i.

NAME

vec_$imult_constant_i – multiply a vector in a 32-bit integer matrix by a scalar

SYNOPSIS (C)

```
#include <apollo/base.h>
#include <apollo/vec.h>

void vec_$imult_constant_i(
        long *mult_vec,
        long int &inc1,
        long int &length,
        long &constant,
        long *result_vec,
        long int &inc2)
```

SYNOPSIS (Pascal)

```
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/vec.ins.pas';

procedure vec_$imult_constant_i(
        in mult_vec: univ vec_$integer32_vector;
        in inc1: integer32;
        in length: integer32;
        in constant: integer32;
        out result_vec: univ vec_$integer32_vector;
        in inc2: integer32);
```

SYNOPSIS (FORTRAN)

```
%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/vec.ins.ftn'

      parameter (nvec = 10)

      integer*4 mult_vec(nvec), result_vec(nvec), constant
      integer*4 length, inc1, inc2

      call vec_$imult_constant_i(mult_vec, inc1, length, constant,
     &                          result_vec, inc2)
```

DESCRIPTION

Vec_$imult_constant_i multiplies *length* elements of the 32-bit integer array *mult_vec* selected by *inc1* by the scalar value *constant* and supplies the result in elements of the array *result_vec* selected by *inc2*.

Through appropriate choice of *inc1* and *inc2*, a program can use **vec_$imult_constant_i** to operate on a vector in a matrix. To multiply *M*th vector in a matrix by *constant*, choose *inc1* equal to the number of vectors in the matrix, and place the *M*th element of the matrix array at the beginning of *mult_vec*. To place the result of the operation in the

*N*th vector of a matrix, choose *inc2* equal to the number of vectors in the resultant matrix, and place the *N*th element of the matrix array at the beginning of *result_vec*.

In C, the resulting operation is

```
j = 0;
k = 0;
for (i = 0; i < length; ++i) {
      result_vec[k] = constant * mult_vec[j];
      k += inc2;
      j += inc1;
}
```

In Pascal, the resulting operation is

```
j := 1;
k := 1;
for i := 1 to length do
      begin
      result_vec[k] := constant * mult_vec[j];
      k := k + inc2;
      j := j + inc1;
      end
```

In FORTRAN, the resulting operation is

```
       j = 1
       k = 1
       do 10 i = 1, length
            result(j) = constant * mult_vec(k)
            j = j + inc2
            k = k + inc1
  10   continue
```

mult_vec
: The vector to multiply by *constant*.

inc1
: An increment for the index of the array *mult_vec* that selects elements to multiply by *constant*.

length
: The number of products to calculate.

constant
: The scalar constant to multiply elements of *mult_vec* by.

result_vec
: An array whose elements receive the product of *constant* and *mult_vec*.

inc2 An increment for the index of the array *result_vec* that selects elements to receive the product of *constant* and *mult_vec*.

NOTES

In C and Pascal, vec_**$imult_constant_i** operates on column vectors; whereas in FORTRAN, it operates on row vectors.

SEE ALSO

vec_$dmult_constant_i, vec_$imult_constant16_i, vec_$imult_constant_i, vec_$mult_constant_i.

NAME

vec_$init – initialize a single-precision vector

SYNOPSIS (C)

#include <apollo/base.h>
#include <apollo/vec.h>

void vec_$init(
float **vector*,
long int *&length*,
float *&constant*)

SYNOPSIS (Pascal)

%include '/sys/ins/base.ins.pas';
%include '/sys/ins/vec.ins.pas';

procedure vec_$init(
var *vector*: **univ vec_$real_vector;**
in *length*: **integer32;**
in *constant*: **real);**

SYNOPSIS (FORTRAN)

%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/vec.ins.ftn'

parameter (*nvec* = **10**)

real *vector(nvec)*, *constant*
integer*4 *length*

call vec_$init(*vector*, *length*, *constant*)

DESCRIPTION

Vec_$init sets *length* elements of *vector* to the single-precision value *constant*.

In C, the resulting operation is

```
for (i = 0; i < length; ++i)
      vector[i] = constant;
```

In Pascal, the resulting operation is

```
for i := 1 to length do
      begin
      vector[i] := constant;
      end
```

In FORTRAN, the resulting operation is

```
      do 10 i = 1, length
           vector(i) = constant
10    continue
```

vector The vector to initialize.

length The number of elements in *vector* to initialize.

constant
The value that the elements of *vector* should be set to.

NOTES
In C and Pascal, vec_**$init** initializes a row vector; whereas in FORTRAN, it initializes a column vector.

SEE ALSO
vec_$dinit, vec_$iinit, vec_$iinit16, vec_$init_i.

NAME

vec_$ipostmult – multiply a 32-bit integer vector by a 4×4 matrix

SYNOPSIS (C)

#include <apollo/base.h>
#include <apollo/vec.h>

void vec_$ipostmult(
long **matrix*,
long **start_vec*,
long **result_vec*)

SYNOPSIS (Pascal)

%include '/sys/ins/base.ins.pas';
%include '/sys/ins/vec.ins.pas';

procedure vec_$ipostmult(
in *matrix*: **univ vec_$integer32_matrix;**
in *start_vec*: **univ vec_$integer32_vector;**
out *result_vec*: **univ vec_$integer32_vector);**

SYNOPSIS (FORTRAN)

%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/vec.ins.ftn'

integer*4 *matrix*(**4, 4**), *start_vec*(**4**), *result_vec*(**4**)

call vec_$ipostmult(*matrix*, *start_vec*, *result_vec*)

DESCRIPTION

Vec_$ipostmult multiplies the 4-element vector *start_vec* by the 4×4 matrix *matrix*.

In C, **vec_$ipostmult** applies *matrix* as a right transform to a row vector *start_vec*, and the resulting operation is

```
for (j = 0; j < 4; ++j) {
      result_vec[j] = 0;
      for (i = 0; i < 4; ++i)
            result_vec[j] += start_vec[i]
                             * matrix[i][j];
}
```

In Pascal, **vec_$ipostmult** applies *matrix* as a right transform to a row vector *start_vec*, and the resulting operation is

```
for j := 1 to 4 do
      begin
      result_vec[j] := 0;
      for i := 1 to 4 do
            result_vec[j] := result_vec[j]
                           + start_vec[i]
                           * matrix[i,j];
      end
```

In FORTRAN, **vec_$ipostmult** applies *matrix* as a left transform to a column vector *start_vec*, and the resulting operation is

```
      do 10 j = 1, 4
           result_vec(j) = 0
           do 10 i = 1, 4
                result_vec(j) = result_vec(j)
     &                        + start_vec(i)
     &                        * matrix(j,i)
   10 continue
```

matrix The matrix to multiply by *start_vec*.

start_vec
The vector to multiply by *matrix*.

result_vec
The product of *start_vec* and *matrix*.

NOTES
Vec_$ipremult transforms 32-bit integer vectors from the other side.

SEE ALSO
vec_$dpostmult, vec_$ipostmult16, vec_$postmult.

NAME

vec_$ipostmult16 – multiply a 16-bit integer vector by a 4×4 matrix

SYNOPSIS (C)

```
#include <apollo/base.h>
#include <apollo/vec.h>

void vec_$ipostmult16(
        short *matrix,
        short *start_vec,
        short *result_vec)
```

SYNOPSIS (Pascal)

```
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/vec.ins.pas';

procedure vec_$ipostmult16(
        in matrix: univ vec_$integer16_matrix;
        in start_vec: univ vec_$integer16_vector;
        out result_vec: univ vec_$integer16_vector);
```

SYNOPSIS (FORTRAN)

```
%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/vec.ins.ftn'

        integer*2 matrix(4, 4), start_vec(4), result_vec(4)

        call vec_$ipostmult16(matrix, start_vec, result_vec)
```

DESCRIPTION

Vec_$ipostmult16 multiplies the 4-element vector *start_vec* by the 4×4 matrix *matrix*.

In C, **vec_$ipostmult16** applies *matrix* as a right transform to a row vector *start_vec*, and the resulting operation is

```
for (j = 0; j < 4; ++j) {
      result_vec[j] = 0;
      for (i = 0; i < 4; ++i)
            result_vec[j] += start_vec[i]
                           * matrix[i][j];
}
```

In Pascal, **vec_$ipostmult16** applies *matrix* as a right transform to a row vector *start_vec*, and the resulting operation is

```
for j := 1 to 4 do
      begin
      result_vec[j] := 0;
      for i := 1 to 4 do
            result_vec[j] := result_vec[j]
                           + start_vec[i]
                           * matrix[i,j];
      end
```

In FORTRAN, **vec_$ipostmult16** applies *matrix* as a left transform to a column vector *start_vec*, and the resulting operation is

```
      do 10 j = 1, 4
           result_vec(j) = 0
           do 10 i = 1, 4
                 result_vec(j) = result_vec(j)
     &                         + start_vec(i)
     &                         * matrix(j,i)
   10 continue
```

matrix The matrix to multiply by *start_vec*.

start_vec
The vector to multiply by *matrix*.

result_vec
The product of *start_vec* and *matrix*.

NOTES
Vec_$ipremult16 transforms 16-bit integer vectors from the other side.

SEE ALSO
vec_$dpostmult, vec_$ipostmult, vec_$postmult.

NAME

vec_$ipostmultn – multiply a 32-bit integer vector by a matrix

SYNOPSIS (C)

```
#include <apollo/base.h>
#include <apollo/vec.h>

void vec_$ipostmultn(
        long *matrix,
        long *start_vec,
        long int &m,
        long int &n,
        long *result_vec)
```

SYNOPSIS (Pascal)

```
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/vec.ins.pas';

procedure vec_$ipostmultn(
        in matrix: univ vec_$integer32_matrix;
        in start_vec: univ vec_$integer32_vector;
        in m: integer32;
        in n: integer32;
        out result_vec: univ vec_$integer32_vector);
```

SYNOPSIS (FORTRAN)

```
%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/vec.ins.ftn'

        integer*4 m, n
        parameter (m = 3, n = 4)

        integer*4 matrix(m, n), start_vec(n), result_vec(m)

        call vec_$ipostmultn(matrix, start_vec, m, n, result_vec)
```

DESCRIPTION

Vec_$ipostmultn multiplies the *n*-element vector *start_vec* by the variably dimensioned matrix *matrix*, and supplies the resulting *m*-element vector in *result_vec*.

In C, **vec_$ipostmultn** applies the $n \times m$ matrix *matrix* as a right transform to the *m*-element row vector *start_vec*, and supplies the transformed *n*-element result in *result_vec*:

```
for (i = 0; i < m; ++i) {
      result_vec[i] = 0;
      for (j = 0; j < n; ++j)
            result_vec[i] += start_vec[j]
                           * matrix[j][i];
}
```

In Pascal, **vec_$ipostmultn** applies the $n \times m$ matrix *matrix* as a right transform to the m-element row vector *start_vec*, and supplies the transformed n-element result in *result_vec*:

```
for i := 1 to m do
      begin
      result_vec[i] := 0.0;
      for j := 1 to n do
            result_vec[i] := result_vec[i]
                           + start_vec[j]
                           * matrix[j,i];
      end;
```

In FORTRAN, **vec_$ipostmultn** applies the $m \times n$ matrix *matrix* as a left transform to the m-element column vector *start_vec*, and supplies the transformed n-element result in *result_vec*:

```
       do 10 i = 1, m
            result_vec(i) = 0.0
            do 10 j = 1, n
                  result_vec(i) = result_vec(i)
      &                         + start_vec(j)
      &                         * matrix(i,j)
   10  continue
```

matrix A matrix to multiply *start_vec* by.

start_vec
An n-element vector to multiply by *matrix*.

m The number of elements in *start_vec*.

n The number of elements in *result_vec*.

result_vec
An m-element vector that is the product of *matrix* and *start_vec*.

NOTES

Vec_$ipremultn transforms 32-bit integer vectors from the other side.

SEE ALSO

vec_$dpostmultn, vec_$ipostmultn16, vec_$postmultn.

NAME

vec_$ipostmultn16 – multiply a 16-bit integer vector by a matrix

SYNOPSIS (C)

```
#include <apollo/base.h>
#include <apollo/vec.h>

void vec_$ipostmultn16(
        short *matrix,
        short *start_vec,
        long int &m,
        long int &n,
        short *result_vec)
```

SYNOPSIS (Pascal)

```
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/vec.ins.pas';

procedure vec_$ipostmultn16(
        in matrix: univ vec_$integer16_matrix;
        in start_vec: univ vec_$integer16_vector;
        in m: integer32;
        in n: integer32;
        out result_vec: univ vec_$integer16_vector);
```

SYNOPSIS (FORTRAN)

```
%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/vec.ins.ftn'

        integer*4 m, n
        parameter (m = 3, n = 4)

        integer*2 matrix(m, n), start_vec(n), result_vec(m)

        call vec_$ipostmultn16(matrix, start_vec, m, n, result_vec)
```

DESCRIPTION

Vec_$ipostmultn16 multiplies the *n*-element vector *start_vec* by the variably dimensioned matrix *matrix*, and supplies the resulting *m*-element vector in *result_vec*.

In C, vec_$ipostmultn16 applies the $n \times m$ matrix *matrix* as a right transform to the *m*-element row vector *start_vec*, and supplies the transformed *n*-element result in *result_vec*:

```
for (i = 0; i < m; ++i) {
      result_vec[i] = 0;
      for (j = 0; j < n; ++j)
            result_vec[i] += start_vec[j]
                           * matrix[j][i];
}
```

In Pascal, vec_**$ipostmultn16** applies the $n \times m$ matrix *matrix* as a right transform to the m-element row vector *start_vec*, and supplies the transformed n-element result in *result_vec*:

```
for i := 1 to m do
      begin
      result_vec[i] := 0.0;
      for j := 1 to n do
            result_vec[i] := result_vec[i]
                           + start_vec[j]
                           * matrix[j,i];
      end;
```

In FORTRAN, vec_**$ipostmultn16** applies the $m \times n$ matrix *matrix* as a left transform to the m-element column vector *start_vec*, and supplies the transformed n-element result in *result_vec*:

```
      do 10 i = 1, m
            result_vec(i) = 0.0
            do 10 j = 1, n
                  result_vec(i) = result_vec(i)
     &                          + start_vec(j)
     &                          * matrix(i,j)
  10  continue
```

matrix A matrix to multiply *start_vec* by.

start_vec
An n-element vector to multiply by *matrix*.

m The number of elements in *start_vec*.

n The number of elements in *result_vec*.

result_vec
An m-element vector that is the product of *matrix* and *start_vec*.

NOTES

Vec_**$ipremultn16** transforms 16-bit integer vectors from the other side.

SEE ALSO

vec_$dpostmultn, vec_$ipostmultn, vec_$postmultn.

NAME

vec_$ipremult – multiply a 32-bit vector by a 4×4 matrix

SYNOPSIS (C)

#include <apollo/base.h>
#include <apollo/vec.h>

void vec_$ipremult(
long **start_vec*,
long **matrix*,
long **result_vec*)

SYNOPSIS (Pascal)

%include '/sys/ins/base.ins.pas';
%include '/sys/ins/vec.ins.pas';

procedure vec_$ipremult(
in *start_vec*: **univ vec_$integer32_vector;**
in *matrix*: **univ vec_$integer32_matrix;**
out *result_vec*: **univ vec_$integer32_vector);**

SYNOPSIS (FORTRAN)

%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/vec.ins.ftn'

integer*4 *start_vec*(4), *matrix*(4,4), *result_vec*(4)

call vec_$ipremult(*start_vec*, *matrix*, *result_vec*)

DESCRIPTION

Vec_$ipremult multiplies the 4-element vector *start_vec* by the 4×4 matrix *matrix*.

In C, **vec_$ipremult** applies *matrix* as a left transform to a column vector *start_vec*, and the resulting operation is

```
for (i = 0; i < 4; ++i) {
      result_vec[i] = 0;
      for (j = 0; i < 4; ++j)
            result_vec[i] += start_vec[i]
                           * matrix[i][j];
}
```

In Pascal, **vec_$ipremult** applies *matrix* as a left transform to a column vector *start_vec*, and the resulting operation is

```
for i := 1 to 4 do
      begin
      result_vec[i] := 0;
      for j := 1 to 4 do
            result_vec[i] := result_vec[i]
                           + start_vec[i]
                           * matrix[i,j];
      end
```

In FORTRAN, **vec_$ipremult** applies *matrix* as a right transform to a row vector *start_vec*, and the resulting operation is

```
        do 10 i = 1, 4
              result_vec(i) = 0
              do 10 j = 1, 4
                    result_vec(i) = result_vec(i)
       &                          + start_vec(j)
       &                          * matrix(j,i)
   10   continue
```

start_vec
The vector to multiply by *matrix*.

matrix The matrix to multiply by *start_vec*.

result_vec
The product of *start_vec* and *matrix*.

NOTES

Vec_$ipostmult transforms 32-bit integer vectors from the other side.

SEE ALSO

vec_$dpremult, vec_$ipremult16, vec_$premult.

NAME

vec_$ipremult16 – multiply a 16-bit vector by a 4×4 matrix

SYNOPSIS (C)

#include <apollo/base.h>
#include <apollo/vec.h>

void vec_$ipremult16(
    short **start_vec*,
    short **matrix*,
    short **result_vec*)

SYNOPSIS (Pascal)

%include '/sys/ins/base.ins.pas';
%include '/sys/ins/vec.ins.pas';

procedure vec_$ipremult16(
    in *start_vec*: **univ vec_$integer16_vector;**
    in *matrix*: **univ vec_$integer16_matrix;**
    out *result_vec*: **univ vec_$integer16_vector);**

SYNOPSIS (FORTRAN)

%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/vec.ins.ftn'

integer*2 *start_vec*(4), *matrix*(4,4), *result_vec*(4)

call vec_$ipremult16(*start_vec*, *matrix*, *result_vec***)**

DESCRIPTION

Vec_$ipremult16 multiplies the 4-element vector *start_vec* by the 4×4 matrix *matrix*.

In C, **vec_$ipremult16** applies *matrix* as a left transform to a column vector *start_vec*, and the resulting operation is

```
for (i = 0; i < 4; ++i) {
      result_vec[i] = 0;
      for (j = 0; i < 4; ++j)
            result_vec[i] += start_vec[i]
                           * matrix[i][j];
}
```

In Pascal, **vec_$ipremult16** applies *matrix* as a left transform to a column vector *start_vec*, and the resulting operation is

```
for i := 1 to 4 do
     begin
     result_vec[i] := 0;
     for j := 1 to 4 do
          result_vec[i] := result_vec[i]
                         + start_vec[i]
                         * matrix[i,j];
     end
```

In FORTRAN, **vec_$ipremult16** applies *matrix* as a right transform to a row vector *start_vec*, and the resulting operation is

```
       do 10 i = 1, 4
            result_vec(i) = 0
            do 10 j = 1, 4
                 result_vec(i) = result_vec(i)
     &                         + start_vec(j)
     &                         * matrix(j,i)
  10   continue
```

start_vec
The vector to multiply by *matrix*.

matrix The matrix to multiply by *start_vec*.

result_vec
The product of *start_vec* and *matrix*.

NOTES

Vec_$ipostmult16 transforms 16-bit integer vectors from the other side.

SEE ALSO

vec_$dpremult, vec_$ipremult, vec_$premult.

NAME

vec_$ipremultn – multiply a 32-bit integer vector by a matrix

SYNOPSIS (C)

```
#include <apollo/base.h>
#include <apollo/vec.h>

void vec_$ipremultn(
        long *start_vec,
        long *matrix,
        long int &m,
        long int &n,
        long *result_vec)
```

SYNOPSIS (Pascal)

```
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/vec.ins.pas';

procedure vec_$ipremultn(
        in start_vec: univ vec_$integer32_vector;
        in matrix: univ vec_$integer32_matrix;
        in m: integer32;
        in n: integer32;
        out result_vec: univ vec_$integer32_vector);
```

SYNOPSIS (FORTRAN)

```
%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/vec.ins.ftn'

        integer*4 m, n
        parameter (m = 3, n = 4)

        integer*4 start_vec(m), matrix(m, n), result_vec(n)

        call vec_$ipremultn(start_vec, matrix, m, n, result_vec)
```

DESCRIPTION

Vec_$ipremultn multiplies the m-element vector *start_vec* by the variably dimensioned matrix *matrix*, and supplies the resulting n-element vector in *result_vec*.

In C, **vec_$ipremultn** applies the $n \times m$ matrix *matrix* as a left transform to the m-element column vector *start_vec*, and supplies the transformed n-element result in *result_vec*:

```
for (i = 0; i < n; ++i) {
      result_vec[i] = 0;
      for (j = 0; i < m; ++i)
            result_vec[i] += start_vec[j]
                              * matrix[i][j];
}
```

In Pascal, **vec_$ipremultn** applies the $n \times m$ matrix *matrix* as a left transform to the m-element column vector *start_vec*, and supplies the transformed n-element result in *result_vec*:

```
for i := 1 to n do
      begin
      result_vec[i] := 0;
      for j := 1 to m do
            result_vec[i] := result_vec[i]
                              + start_vec[j]
                              * matrix[i,j];
      end
```

In FORTRAN, **vec_$ipremultn** applies the $m \times n$ matrix *matrix* as a right transform to the m-element row vector *start_vec*, and supplies the transformed n-element result in *result_vec*:

```
       do 10 i = 1, n
            result_vec(i) = 0
            do 10 j = 1, m
                  result_vec(i) = result_vec(i)
     &                            + start_vec(j)
     &                            * matrix(j,i)
  10   continue
```

start_vec
: An m-element vector to multiply by *matrix*.

matrix
: A matrix to multiply *start_vec* by.

m
: The number of elements in *start_vec*.

n
: The number of elements in *result_vec*.

result_vec
: An n-element vector that is the product of *matrix* and *start_vec*.

NOTES

Vec_$ipostmultn transforms 32-bit integer vectors from the other side.

SEE ALSO

vec_$dpremultn, vec_$ipremultn16, vec_$premultn.

NAME

vec_$ipremultn16 – multiply a 16-bit integer vector by a matrix

SYNOPSIS (C)

```
#include <apollo/base.h>
#include <apollo/vec.h>

void vec_$ipremultn16(
        short *start_vec,
        short *matrix,
        long int &m,
        long int &n,
        short *result_vec)
```

SYNOPSIS (Pascal)

```
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/vec.ins.pas';

procedure vec_$ipremultn16(
        in start_vec: univ vec_$integer16_vector;
        in matrix: univ vec_$integer16_matrix;
        in m: integer32;
        in n: integer32;
        out result_vec: univ vec_$integer16_vector);
```

SYNOPSIS (FORTRAN)

```
%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/vec.ins.ftn'

        integer*4 m, n
        parameter (m = 3, n = 4)

        integer*2 start_vec(m), matrix(m, n), result_vec(n)

        call vec_$ipremultn16(start_vec, matrix, m, n, result_vec)
```

DESCRIPTION

Vec_$ipremultn16 multiplies the *m*-element vector *start_vec* by the variably dimensioned matrix *matrix*, and supplies the resulting *n*-element vector in *result_vec*.

In C, **vec_$ipremultn16** applies the $n \times m$ matrix *matrix* as a left transform to the *m*-element column vector *start_vec*, and supplies the transformed *n*-element result in *result_vec*:

```
for (i = 0; i < n; ++i) {
      result_vec[i] = 0;
      for (j = 0; i < m; ++i)
            result_vec[i] += start_vec[j]
                           * matrix[i][j];
}
```

In Pascal, **vec_$ipremultn16** applies the $n \times m$ matrix *matrix* as a left transform to the m-element column vector *start_vec*, and supplies the transformed n-element result in *result_vec*:

```
for i := 1 to n do
      begin
      result_vec[i] := 0;
      for j := 1 to m do
            result_vec[i] := result_vec[i]
                           + start_vec[j]
                           * matrix[i,j];
      end
```

In FORTRAN, **vec_$ipremultn16** applies the $m \times n$ matrix *matrix* as a right transform to the m-element row vector *start_vec*, and supplies the transformed n-element result in *result_vec*:

```
       do 10 i = 1, n
            result_vec(i) = 0
            do 10 j = 1, m
                  result_vec(i) = result_vec(i)
     &                          + start_vec(j)
     &                          * matrix(j,i)
  10   continue
```

start_vec
: An m-element vector to multiply by *matrix*.

matrix
: A matrix to multiply *start_vec* by.

m
: The number of elements in *start_vec*.

n
: The number of elements in *result_vec*.

result_vec
: An n-element vector that is the product of *matrix* and *start_vec*.

NOTES

Vec_$ipostmultn16 transforms 16-bit integer vectors from the other side.

SEE ALSO

vec_$dpremultn, vec_$ipremultn, vec_$premultn.

NAME

vec_$isub – subtract 32-bit integer vectors

SYNOPSIS (C)

#include <apollo/base.h>
#include <apollo/vec.h>

void vec_$isub(
long ******start_vec*,
long ******sub_vec*,
long int &_length_,
long ******result_vec***)**

SYNOPSIS (Pascal)

%include '/sys/ins/base.ins.pas';
%include '/sys/ins/vec.ins.pas';

procedure vec_$isub(
in *start_vec*: **univ vec_$integer32_vector;**
in *sub_vec*: **univ vec_$integer32_vector;**
in *length*: **integer32;**
out *result_vec*: **univ vec_$integer32_vector);**

SYNOPSIS (FORTRAN)

%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/vec.ins.ftn'

parameter (*nvec* = **10**)

integer*4 *start_vec(nvec), sub_vec(nvec), result_vec(nvec)*
integer*4 *length*

call vec_$isub(*start_vec, sub_vec, length, result_vec***)**

DESCRIPTION

Vec_$isub subtracts *length* elements of *sub_vec* from *start_vec* and supplies the difference in *result_vec*.

In C, the resulting operation is

```
for (i = 0; i < length; ++i)
     result_vec[i] = start_vec[i] - sub_vec[i];
```

In Pascal, the resulting operation is

```
for i := 1 to length do
     result_vec[i] := start_vec[i] - sub_vec[i];
```

In FORTRAN, the resulting operation is

```
      do 10 i = 1, length
           result_vec(i) = start_vec(i) - sub_vec(i)
  10  continue
```

start_vec
The vector to subtract *sub_vec* from.

sub_vec The vector to subtract from *start_vec*.

length The number of differences to calculate.

result_vec
The difference of *start_vec* and *sub_vec*.

NOTES

When **vec_$isub** is used to operate on matrixes in C and Pascal, *start_vec*, *result_vec*, and *result_vec* are row vectors; whereas in FORTRAN, they are column vectors.

SEE ALSO

vec_$dsub, vec_$isub16, vec_$isub_i, vec_$sub.

NAME

vec_$isub16 – subtract 16-bit integer vectors

SYNOPSIS (C)

#include <apollo/base.h>
#include <apollo/vec.h>

void vec_$isub16(
  short **start_vec*,
  short **sub_vec*,
  long int *&length*,
  short **result_vec*)

SYNOPSIS (Pascal)

%include '/sys/ins/base.ins.pas';
%include '/sys/ins/vec.ins.pas';

procedure vec_$isub16(
  in *start_vec*: **univ vec_$integer16_vector;**
  in *sub_vec*: **univ vec_$integer16_vector;**
  in *length*: **integer32;**
  out *result_vec*: **univ vec_$integer16_vector);**

SYNOPSIS (FORTRAN)

%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/vec.ins.ftn'

parameter (*nvec* = **10)**

integer*2 *start_vec(nvec), sub_vec(nvec), result_vec(nvec)*
integer*4 *length*

call vec_$isub16(*start_vec, sub_vec, length, result_vec*)

DESCRIPTION

Vec_$isub16 subtracts *length* elements of *sub_vec* from *start_vec* and supplies the difference in *result_vec*.

In C, the resulting operation is

```
for (i = 0; i < length; ++i)
     result_vec[i] = start_vec[i] - sub_vec[i];
```

In Pascal, the resulting operation is

```
for i := 1 to length do
     result_vec[i] := start_vec[i] - sub_vec[i];
```

In FORTRAN, the resulting operation is

```
      do 10 i = 1, length
           result_vec(i) = start_vec(i) - sub_vec(i)
   10 continue
```

start_vec
The vector to subtract *sub_vec* from.

sub_vec The vector to subtract from *start_vec*.

length The number of differences to calculate.

result_vec
The difference of *start_vec* and *sub_vec*.

NOTES

When **vec_$isub16** is used to operate on matrixes in C and Pascal, *start_vec*, *result_vec*, and *result_vec* are row vectors; whereas in FORTRAN, they are column vectors.

SEE ALSO

vec_$dsub, vec_$isub16_i, vec_$isub_i, vec_$sub.

NAME

vec_$isub16_i – subtract 16-bit integer vectors in matrixes

SYNOPSIS (C)

#include <apollo/base.h>
#include <apollo/vec.h>

SYNOPSIS (Pascal)

%include '/sys/ins/base.ins.pas';
%include '/sys/ins/vec.ins.pas';

SYNOPSIS (FORTRAN)

%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/vec.ins.ftn'

```
        parameter (nvec = 10)

        integer*2 start_vec(nvec), sub_vec(nvec), result_vec(nvec)
        integer*4 length, inc1, inc2, inc3

        call vec_$isub16_i(start_vec, inc1, sub_vec, inc2,
       &                   length, result_vec, inc3)
```

DESCRIPTION

Vec_$isub16_i subtracts *length* elements of *sub_vec* selected by *inc2* from *length* elements of *start_vec* selected by *inc1*, and supplies the result in the elements of *result_vec* selected by *inc3*.

Through appropriate choice of *inc1*, *inc2*, and *inc3*, a program can use **vec_$isub16_i** to subtract individual vectors in two matrixes and place the difference in a vector of another matrix. To subtract the *N*th vector in matrix *Y* from the *M*th vector in matrix *X*, choose *inc2* equal to the number of vectors in matrix *Y* and *inc1* equal to the number of vectors in matrix *X*. Then place the *M*th element of matrix *X* at the beginning of *start_vec*, and place the *N*th element of matrix *Y* at the beginning of *sub_vec*. To place the result of the operation in the *P*th vector of a resultant matrix, choose *inc3* equal to the number of vectors in the resultant matrix, and place the *P*th element of the matrix array at the beginning of *result_vec*.

In C, the resulting operation is

```
j = 0;
k = 0;
l = 0;
for (i = 0; i < length, ++i) {
      result_vec[j] = start_vec[k] - sub_vec[l];
      j += inc3;
      k += inc1;
      l += inc2;
}
```

In Pascal, the resulting operation is

```
j := 1;
k := 1;
l := 1;
for i := 1 to length do
      begin
      result_vec[j] := start_vec[k] - sub_vec[l];
      j := j + inc3;
      k := k + inc1;
      l := l + inc2;
      end
```

In FORTRAN, the resulting operation is

```
       j = 1
       k = 1
       l = 1
       do 10 i = 1, length
            result_vec(j) = start_vec(k) - sub_vec(l)
            j = j + inc3
            k = k + inc1
            l = l + inc2
  10   continue
```

start_vec
The vector to subtract *sub_vec* from.

inc1 An increment for the index of *start_vec* that chooses which elements the elements of *sub_vec* will be subtracted from.

sub_vec The vector to subtract from *start_vec*.

inc2 An increment for the index of *sub_vec* that chooses which elements will be subtracted from the elements of *start_vec*.

length The number of scalar differences to calculate.

result_vec
The difference of *start_vec* and *sub_vec*.

inc3 An increment for the index of *result_vec* that chooses which elements will received the differences.

NOTES

In C and Pascal, **vec_$isub16_i** operates on column vectors; whereas in FORTRAN, it operates on row vectors.

SEE ALSO

vec_$dsub_i, vec_$isub16, vec_$isub_i, vec_$sub_i.

NAME

vec_$isub_i – subtract 32-bit integer vectors in matrixes

SYNOPSIS (C)

```
#include <apollo/base.h>
#include <apollo/vec.h>

void vec_$isub_i(
        long *start_vec,
        long int &inc1,
        long *sub_vec,
        long int &inc2,
        long int &length,
        long *result_vec,
        long int &inc3)
```

SYNOPSIS (Pascal)

```
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/vec.ins.pas';

procedure vec_$isub_i(
        in start_vec: univ vec_$integer32_vector;
        in inc1: integer32;
        in sub_vec: univ vec_$integer32_vector;
        in inc2: integer32;
        in length: integer32;
        out result_vec: univ vec_$integer32_vector;
        in inc3: integer32);
```

SYNOPSIS (FORTRAN)

```
%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/vec.ins.ftn'

      parameter (nvec = 10)

      integer*4 start_vec(nvec), sub_vec(nvec), result_vec(nvec)
      integer*4 length, inc1, inc2, inc3

      call vec_$isub_i(start_vec, inc1, sub_vec, inc2,
     &                 length, result_vec, inc3)
```

DESCRIPTION

Vec_$isub_i subtracts *length* elements of *sub_vec* selected by *inc2* from *length* elements of *start_vec* selected by *inc1*, and supplies the result in the elements of *result_vec* selected by *inc3*.

Through appropriate choice of *inc1*, *inc2*, and *inc3*, a program can use vec_$isub_i to subtract individual vectors in two matrixes and place the difference in a vector of another

matrix. To subtract the *N*th vector in matrix *Y* from the *M*th vector in matrix *X*, choose *inc2* equal to the number of vectors in matrix *Y* and *inc1* equal to the number of vectors in matrix *X*. Then place the *M*th element of matrix *X* at the beginning of *start_vec*, and place the *N*th element of matrix *Y* at the beginning of *sub_vec*. To place the result of the operation in the *P*th vector of a resultant matrix, choose *inc3* equal to the number of vectors in the resultant matrix, and place the *P*th element of the matrix array at the beginning of *result_vec*.

In C, the resulting operation is

```
j = 0;
k = 0;
l = 0;
for (i = 0; i < length, ++i) {
      result_vec[j] = start_vec[k] - sub_vec[l];
      j += inc3;
      k += inc1;
      l += inc2;
}
```

In Pascal, the resulting operation is

```
j := 1;
k := 1;
l := 1;
for i := 1 to length do
      begin
      result_vec[j] := start_vec[k] - sub_vec[l];
      j := j + inc3;
      k := k + inc1;
      l := l + inc2;
      end
```

In FORTRAN, the resulting operation is

```
      j = 1
      k = 1
      l = 1
      do 10 i = 1, length
           result_vec(j) = start_vec(k) - sub_vec(l)
           j = j + inc3
           k = k + inc1
           l = l + inc2
 10   continue
```

start_vec
: The vector to subtract *sub_vec* from.

inc1
: An increment for the index of *start_vec* that chooses which elements the elements of *sub_vec* will be subtracted from.

sub_vec
: The vector to subtract from *start_vec*.

inc2
: An increment for the index of *sub_vec* that chooses which elements will be subtracted from the elements of *start_vec*.

length
: The number of scalar differences to calculate.

result_vec
: The difference of *start_vec* and *sub_vec*.

inc3
: An increment for the index of *result_vec* that chooses which elements will received the differences.

NOTES

In C and Pascal, **vec_$isub_i** operates on column vectors; whereas in FORTRAN, it operates on row vectors.

SEE ALSO

vec_$dsub_i, vec_$isub, vec_$isub16_i, vec_$sub_i.

NAME

vec_$isum – sum the elements of a 32-bit integer vector

SYNOPSIS (C)

```
#include <apollo/base.h>
#include <apollo/vec.h>

long vec_$isum(
        long *vec,
        long int &length)
```

SYNOPSIS (Pascal)

```
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/vec.ins.pas';

function vec_$isum(
        in vec: univ vec_$integer32_vector;
        in length: integer32): integer32
```

SYNOPSIS (FORTRAN)

```
%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/vec.ins.ftn'

        parameter (nvec = 10)

        integer*4 vec(nvec), sum
        integer*4 length

        sum = vec_$isum(vec, length)
```

DESCRIPTION

Vec_$isum adds together *length* elements of the 32-bit integer array *vector* and returns the sum.

In C, the resulting operation is

```
return_value = 0;
for (i = 0; i < length; ++i)
     return_value += vec[i];
```

In Pascal, the resulting operation is

```
return_value := 0;
for i := 1 to length do
     return_value := return_value + vec[i];
```

In FORTRAN, the resulting operation is

```
       vec_$isum = 0
       do 10 i = 1, length
            vec_$isum = vec_$isum + vec(i)
  10   continue
```

vec The vector to sum.

length The number of elements in *vec* to sum.

NOTES

When vec_$isum is used to operate on matrixes in C and Pascal, *vec* is a row vector; whereas in FORTRAN, it is a column vector.

SEE ALSO

vec_$dsum, vec_$isum16, vec_$isum_i, vec_$sum.

NAME

vec_$isum16 – sum the elements of a 16-bit integer vector

SYNOPSIS (C)

#include <apollo/base.h>
#include <apollo/vec.h>

short vec_$isum16(
short **vec*,
long int *&length***)**

SYNOPSIS (Pascal)

%include '/sys/ins/base.ins.pas';
%include '/sys/ins/vec.ins.pas';

function vec_$isum16(
in *vec*: **univ vec_$integer16_vector;**
in *length*: **integer32): integer16;**

SYNOPSIS (FORTRAN)

%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/vec.ins.ftn'

parameter (*nvec* = **10**)

integer*2 *vec*(*nvec*), *sum*
integer*4 *length*

sum = **vec_$isum16**(*vec*, *length*)

DESCRIPTION

Vec_$isum16 adds together *length* elements of the 16-bit integer array *vector* and returns the sum.

In C, the resulting operation is

```
return_value = 0;
for (i = 0; i < length; ++i)
      return_value += vec[i];
```

In Pascal, the resulting operation is

```
return_value := 0;
for i := 1 to length do
      return_value := return_value + vec[i];
```

In FORTRAN, the resulting operation is

```
       vec_$isum16 = 0
       do 10 i = 1, length
            vec_$isum16 = vec_$isum16 + vec(i)
  10   continue
```

vec The vector to sum.

length The number of elements in *vec* to sum.

NOTES

When **vec_$isum16** is used to operate on matrixes in C and Pascal, *vec* is a row vector; whereas in FORTRAN, it is a column vector.

SEE ALSO

vec_$dsum, vec_$isum, vec_$isum16_i, vec_$sum.

NAME

vec_$isum16_i – sum the elements of a vector in a 16-bit integer matrix

SYNOPSIS (C)

```
#include <apollo/base.h>
#include <apollo/vec.h>

short vec_$isum16_i(
        short *vec,
        long int &inc,
        long int &length)
```

SYNOPSIS (Pascal)

```
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/vec.ins.pas';

function vec_$isum16_i(
        in vec: univ vec_$integer16_vector;
        in inc: integer32;
        in length: integer32): integer16;
```

SYNOPSIS (FORTRAN)

```
%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/vec.ins.ftn'

        parameter (nvec = 10)

        integer*2 vec(nvec), sum
        integer*4 length, inc

        sum = vec_$isum16_i(vec, inc, length)
```

DESCRIPTION

Vec_$isum16_i adds *length* elements from the 16-bit integer array *vector* selected by *inc* and returns the sum.

Through appropriate choice of *inc*, a program can use **vec_$isum16_i** to sum an individual vector in a matrix. To sum the *M*th vector in a matrix, choose *inc* equal to the number of vectors in the matrix.

In C, the resulting operation is

```
return_value = 0;
j = 0;
for (i = 0; i < length; ++i) {
      return_value += vec[j];
      j += inc;
}
```

In Pascal, the resulting operation is

```
return_value := 0;
j := 1;
for i := 1 to length do
      begin
      return_value := return_value + vec[j];
      j := j + inc;
      end
```

In FORTRAN, the resulting operation is

```
       vec_$isum16_i = 0
       j = 1
       do 10 i = 1, length
            vec_$isum16_i = vec_$isum16_i + vec(j)
            j = j + inc
  10   continue
```

vec An array that contains the elements to sum.

inc An increment for the index of *vec* that selects which elements of *vec* are summed.

length The number of elements in *vec* to sum.

NOTES

In C and Pascal, **vec_$isum16_i** sums a column vector; whereas in FORTRAN, it sums a row vector.

SEE ALSO

vec_$dsum_i, vec_$isum16, vec_$isum_i, vec_$sum_i.

NAME

vec_$isum_i – sum the elements of a vector in a 32-bit integer matrix

SYNOPSIS (C)

```
#include <apollo/base.h>
#include <apollo/vec.h>

long vec_$isum_i(
        long *vec,
        long int &inc,
        long int &length)
```

SYNOPSIS (Pascal)

```
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/vec.ins.pas';

function vec_$isum_i(
        in vec: univ vec_$integer32_vector;
        in inc: integer32;
        in length: integer32): integer32;
```

SYNOPSIS (FORTRAN)

```
%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/vec.ins.ftn'

        parameter (nvec = 10)

        integer*4 vec(nvec), sum
        integer*4 length, inc

        sum = vec_$isum_i(vec, inc, length)
```

DESCRIPTION

Vec_$isum_i adds *length* elements from the 32-bit integer array *vector* selected by *inc* and returns the sum.

Through appropriate choice of *inc*, a program can use **vec_$isum_i** to sum an individual vector in a matrix. To sum the *M*th vector in a matrix, choose *inc* equal to the number of vectors in the matrix.

In C, the resulting operation is

```
return_value = 0;
j = 0;
for (i = 0; i < length; ++i) {
      return_value += vec[j];
      j += inc;
}
```

In Pascal, the resulting operation is

```
return_value := 0;
j := 1;
for i := 1 to length do
      begin
      return_value := return_value + vec[j];
      j := j + inc;
      end
```

In FORTRAN, the resulting operation is

```
      vec_$isum_i = 0
      j = 1
      do 10 i = 1, length
           vec_$isum_i = vec_$isum_i + vec(j)
           j = j + inc
  10  continue
```

vec An array that contains the elements to sum.

inc An increment for the index of *vec* that selects which elements of *vec* are summed.

length The number of elements in *vec* to sum.

NOTES

In C and Pascal, **vec_$isum_i** sums a column vector; whereas in FORTRAN, it sums a row vector.

SEE ALSO

vec_$dsum_i, vec_$isum, vec_$isum16_i, vec_$sum_i.

NAME

vec_$iswap – swap two 32-bit integer vectors

SYNOPSIS (C)

```
#include <apollo/base.h>
#include <apollo/vec.h>

void vec_$iswap(
        long *vec1,
        long *vec2,
        long int &length)
```

SYNOPSIS (Pascal)

```
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/vec.ins.pas';

procedure vec_$iswap(
        var vec1: univ vec_$integer32_vector;
        var vec2: univ vec_$integer32_vector;
        in length: integer32);
```

SYNOPSIS (FORTRAN)

```
%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/vec.ins.ftn'

        parameter (nvec = 10)

        integer*4 vec1(nvec), vec2(nvec)
        integer*4 length

        call vec_$iswap(vec1, vec2, length)
```

DESCRIPTION

Vec_$iswap swaps *length* elements between the 32-bit integer vectors *vec1* and *vec2*.

In C, the resulting operation is

```
for (i = 0; i < length, ++i) {
      temp = vec1[i];
      vec1[i] = vec2[i];
      vec2[i] = temp;
}
```

In Pascal, the resulting operation is

```
for i := 1 to length do
      begin
      temp := vec1[i];
      vec1[i] := vec2[i];
      vec2[i] := temp;
      end
```

In FORTRAN, the resulting operation is

```
       do 10 i = 1, length
            temp = vec1(i)
            vec1(i) = vec2(i)
            vec2(i) = temp
  10   continue
```

vec1 The vector to be swapped with *vec2*.

vec2 The vector to be swapped with *vec1*.

length The number of elements to swap.

NOTES

When **vec_$iswap** is used to operate on matrixes in C and Pascal, *vec1* and *vec2* are row vectors; whereas in FORTRAN, they are column vectors.

SEE ALSO

vec_$dswap, vec_$iswap16, vec_$iswap_i, vec_$swap.

NAME

vec_$iswap16 – swap two 16-bit integer vectors

SYNOPSIS (C)

#include <apollo/base.h>
#include <apollo/vec.h>

void vec_$iswap16(
short **vec1*,
short **vec2*,
long int *&length***)**

SYNOPSIS (Pascal)

%include '/sys/ins/base.ins.pas';
%include '/sys/ins/vec.ins.pas';

procedure vec_$iswap16(
var *vec1*: **univ vec_$integer16_vector;**
var *vec2*: **univ vec_$integer16_vector;**
in *length*: **integer32);**

SYNOPSIS (FORTRAN)

%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/vec.ins.ftn'

parameter (*nvec* = **10)**

integer*2 *vec1(nvec)*, *vec2(nvec)*
integer*4 *length*

call vec_$iswap16(*vec1*, *vec2*, *length***)**

DESCRIPTION

Vec_$iswap16 swaps *length* elements between the 16-bit integer vectors *vec1* and *vec2*.

In C, the resulting operation is

```
for (i = 0; i < length, ++i) {
      temp = vec1[i];
      vec1[i] = vec2[i];
      vec2[i] = temp;
}
```

In Pascal, the resulting operation is

```
for i := 1 to length do
      begin
      temp := vec1[i];
      vec1[i] := vec2[i];
      vec2[i] := temp;
      end
```

In FORTRAN, the resulting operation is

```
      do 10 i = 1, length
           temp = vec1(i)
           vec1(i) = vec2(i)
           vec2(i) = temp
 10   continue
```

vec1 The vector to be swapped with *vec2*.

vec2 The vector to be swapped with *vec1*.

length The number of elements to swap.

NOTES

When vec_**$iswap16** is used to operate on matrixes in C and Pascal, *vec1* and *vec2* are row vectors; whereas in FORTRAN, they are column vectors.

SEE ALSO

vec_$swap, vec_$dswap, vec_$iswap, vec_$iswap16.

NAME

vec_$iswap16_i – swap two vectors in a 16-bit integer matrix

SYNOPSIS (C)

```
#include <apollo/base.h>
#include <apollo/vec.h>

void vec_$iswap16_i(
        short *vec1,
        long int &inc1,
        short *vec2,
        long int &inc2,
        long int &length)
```

SYNOPSIS (Pascal)

```
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/vec.ins.pas';

procedure vec_$iswap16_i(
        var vec1: univ vec_$integer16_vector;
        in inc1: integer32;
        var vec1: univ vec_$integer16_vector;
        in inc2: integer32;
        in length: integer32);
```

SYNOPSIS (FORTRAN)

```
%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/vec.ins.ftn'

        parameter (nvec = 10)

        integer*2 vec1(nvec), vec2(nvec)
        integer*4 length, inc1, inc2

        call vec_$iswap16_i(vec1, inc1, vec2, inc2, length)
```

DESCRIPTION

Vec_$iswap16_i swaps the *length* elements of *vec1* selected by *inc1* with the *length* elements of *vec2* selected by *inc2*.

Through appropriate choice of *inc1* and *inc2*, a program can use vec_$iswap16_i to swap vectors between two matrixes. To select the *M*th vector in a matrix, choose *inc1* equal to the number of vectors in the matrix, and place the *M*th element of the matrix array at the beginning of *vec1*. To swap the selected vector with the *N*th vector of the same or another matrix, choose *inc2* equal to the number of vectors in the same or other matrix, and place the *N*th element of the matrix array at the beginning of *vec2*.

In C, the resulting operation is

```
j = 0;
k = 0;
for (i = 0; i < length, ++i) {
      temp = vec1[i];
      vec1[i] = vec2[i];
      vec2[i] = temp;
      j += inc1;
      k += inc2;
}
```

In Pascal, the resulting operation is

```
j := 1;
k := 1;
for i := 1 to length do
      begin
      temp := vec1[i];
      vec1[i] := vec2[i];
      vec2[i] := temp;
      j := j + inc1;
      k := k + inc2;
      end
```

In FORTRAN, the resulting operation is

```
      j = 1
      k = 1
      do 10 i = 1, length
           temp = vec1(j)
           vec1(j) = vec2(k)
           vec2(k) = temp
           j = j + inc1
           k = k + inc2
10    continue
```

vec1 The vector to be swapped with *vec2*.

inc1 The increment for the index of *vec1* that selects the elements to be swapped with *vec2*.

vec2 The vector to be swapped with *vec1*.

inc2 The increment for the index of *vec2* that selects the elements to be swapped with *vec1*.

length The number of elements to swap.

NOTES

In C and Pascal, **vec_$iswap16_i** swaps column vectors; whereas in FORTRAN, it swaps row vectors.

SEE ALSO

vec_$dswap_i, vec_$iswap16, vec_$iswap_i, vec_$swap_i.

NAME

vec_$iswap_i – swap two vectors in a 32-bit integer matrix

SYNOPSIS (C)

```
#include <apollo/base.h>
#include <apollo/vec.h>

void vec_$iswap_i(
        long *vec1,
        long int &inc1,
        long *vec2,
        long int &inc2,
        long int &length)
```

SYNOPSIS (Pascal)

```
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/vec.ins.pas';

procedure vec_$iswap_i(
        var vec1: univ vec_$integer32_vector;
        in inc1: integer32;
        var vec2: univ vec_$integer32_vector;
        in inc2: integer32;
        in length: integer32);
```

SYNOPSIS (FORTRAN)

```
%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/vec.ins.ftn'

         parameter (nvec = 10)

         integer*4 vec1(nvec), vec2(nvec)
         integer*4 length, inc1, inc2

         call vec_$iswap_i(vec1, inc1, vec2, inc2, length)
```

DESCRIPTION

Vec_$iswap_i swaps the *length* elements of *vec1* selected by *inc1* with the *length* elements of *vec2* selected by *inc2*.

Through appropriate choice of *inc1* and *inc2*, a program can use **vec_$iswap_i** to swap vectors between two matrixes. To select the *M*th vector in a matrix, choose *inc1* equal to the number of vectors in the matrix, and place the *M*th element of the matrix array at the beginning of *vec1*. To swap the selected vector with the *N*th vector of the same or another matrix, choose *inc2* equal to the number of vectors in the same or other matrix, and place the *N*th element of the matrix array at the beginning of *vec2*.

In C, the resulting operation is

```
j = 0;
k = 0;
for (i = 0; i < length, ++i) {
      temp = vec1[i];
      vec1[i] = vec2[i];
      vec2[i] = temp;
      j += inc1;
      k += inc2;
}
```

In Pascal, the resulting operation is

```
j := 1;
k := 1;
for i := 1 to length do
      begin
      temp := vec1[i];
      vec1[i] := vec2[i];
      vec2[i] := temp;
      j := j + inc1;
      k := k + inc2;
      end
```

In FORTRAN, the resulting operation is

```
       j = 1
       k = 1
       do 10 i = 1, length
            temp = vec1(j)
            vec1(j) = vec2(k)
            vec2(k) = temp
            j = j + inc1
            k = k + inc2
  10   continue
```

vec1 The vector to be swapped with *vec2*.

inc1 The increment for the index of *vec1* that selects the elements to be swapped with *vec2*.

vec2 The vector to be swapped with *vec1*.

inc2 The increment for the index of *vec2* that selects the elements to be swapped with *vec1*.

length The number of elements to swap.

NOTES

In C and Pascal, vec_$**iswap_i** swaps column vectors; whereas in FORTRAN, it swaps row vectors.

SEE ALSO

vec_$dswap_i, vec_$iswap, vec_$iswap16_i, vec_$swap_i.

NAME

vec_$izero – zero a 32-bit integer vector

SYNOPSIS (C)

#include <apollo/base.h>
#include <apollo/vec.h>

void vec_$izero(
long **vector*,
long int &*length***)**

SYNOPSIS (Pascal)

%include '/sys/ins/base.ins.pas';
%include '/sys/ins/vec.ins.pas';

procedure vec_$izero(
var *vector*: **univ vec_$integer32_vector;**
in *length*: **integer32);**

SYNOPSIS (FORTRAN)

%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/vec.ins.ftn'

parameter (*nvec* = **10)**

integer*4 *vector*(*nvec*)
integer*4 *length*

call vec_$izero(*vector*, *length***)**

DESCRIPTION

Vec_$izero zeros the first *length* elements of the 32-bit integer vector *vector*.

In C, the resulting operation is

```
for (i = 0; i < length; ++i)
      vector[i] = 0;
```

In Pascal, the resulting operation is

```
for i := 1 to length do
      vector[i] := 0;
```

In FORTRAN, the resulting operation is

```
      do 10 i = 1, length
           vec(i) = 0
 10   continue
```

vector The vector to be zeroed.

length The number of elements in *vector* to zero.

NOTES

In C and Pascal, vec_$izero zeros row vectors; whereas in FORTRAN, it zeros column vectors.

SEE ALSO

vec_$dzero, vec_$izero16, vec_$izero_i, vec_$zero.

NAME

vec_$izero16 – zero a 16-bit integer vector

SYNOPSIS (C)

#include <apollo/base.h>
#include <apollo/vec.h>

void vec_$izero16(
    short ***vector*****,**
    long int &***length*****)**

SYNOPSIS (Pascal)

%include '/sys/ins/base.ins.pas';
%include '/sys/ins/vec.ins.pas';

procedure vec_$izero16(
    var *vector*: **univ vec_$integer16_vector;**
    in *length*: **integer32);**

SYNOPSIS (FORTRAN)

%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/vec.ins.ftn'

parameter (***nvec*** **= 10)**

integer*2 *vector(nvec)*
integer*4 *length*

call vec_$izero16(*vector*, *length***)**

DESCRIPTION

Vec_$izero16 zeros the first *length* elements of the 16-bit integer vector *vector*.

In C, the resulting operation is

```
for (i = 0; i < length; ++i)
      vector[i] = 0;
```

In Pascal, the resulting operation is

```
for i := 1 to length do
      vector[i] := 0;
```

In FORTRAN, the resulting operation is

```
      do 10 i = 1, length
           vec(i) = 0
10    continue
```

vector The vector to be zeroed.

length The number of elements in *vector* to zero.

NOTES

In C and Pascal, **vec_$izero16** zeros row vectors; whereas in FORTRAN, it zeros column vectors.

SEE ALSO

vec_$dzero, vec_$izero, vec_$izero16_i, vec_$zero.

NAME

vec_$izero16_i – zero a vector in a 16-bit integer matrix

SYNOPSIS (C)

```
#include <apollo/base.h>
#include <apollo/vec.h>

void vec_$izero16_i(
        short *vector,
        long int &inc,
        long int &length)
```

SYNOPSIS (Pascal)

```
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/vec.ins.pas';

procedure vec_$izero16_i(
        var vector: univ vec_$integer16_vector;
        in inc: integer32;
        in length: integer32);
```

SYNOPSIS (FORTRAN)

```
%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/vec.ins.ftn'

        parameter (nvec = 10)

        integer*2 vector(nvec)
        integer*4 length, inc

        call vec_$izero16_i(vector, inc, length)
```

DESCRIPTION

Vec_$izero16_i zeros the *length* elements of the 16-bit integer array *vector* selected by *inc*.

Through appropriate choice of *inc*, a program can use vec_$izero16_i to zero a vector within a matrix. To search the *M*th vector in a matrix, choose *inc* equal to the number of vectors in the matrix, and place the *M*th element of the matrix array at the beginning of *vector*.

In C, the resulting operation is

```
j = 0;
for (i = 0; i < length; ++i) {
      vector[i] = 0;
      j += inc;
}
```

In Pascal, the resulting operation is

```
j := 1;
for i := 1 to length do
      begin
      vector[i] := 0;
      j := j + inc;
      end
```

In FORTRAN, the resulting operation is

```
      j = 1
      do 10 i = 1, length
           vec(j) = 0
           j = j + inc
  10  continue
```

vector The vector to be zeroed.

inc An increment for the index of *vector* that chooses the elements to be zeroed.

length The number of elements in *vector* to zero.

NOTES

In C and Pascal, **vec_$izero16_i** zeros column vectors; whereas in FORTRAN, it zeros row vectors.

SEE ALSO

vec_$dzero_i, vec_$izero16, vec_$izero_i, vec_$zero_i.

NAME

vec_$izero_i – zero a vector in a 32-bit integer matrix

SYNOPSIS (C)

```
#include <apollo/base.h>
#include <apollo/vec.h>

void vec_$izero_i(
        long *vector,
        long int &inc,
        long int &length)
```

SYNOPSIS (Pascal)

```
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/vec.ins.pas';

procedure vec_$izero_i(
        var vector: univ vec_$integer32_vector;
        in inc: integer32;
        in length: integer32);
```

SYNOPSIS (FORTRAN)

```
%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/vec.ins.ftn'

        parameter (nvec = 10)

        integer*4 vector(nvec)
        integer*4 length, inc

        call vec_$izero_i(vector, inc, length)
```

DESCRIPTION

Vec_$izero_i zeros the *length* elements of the 32-bit integer array *vector* selected by *inc*.

Through appropriate choice of *inc*, a program can use vec_$izero_i to zero a vector within a matrix. To search the *M*th vector in a matrix, choose *inc* equal to the number of vectors in the matrix, and place the *M*th element of the matrix array at the beginning of *vector*.

In C, the resulting operation is

```
j = 0;
for (i = 0; i < length; ++i) {
      vector[i] = 0;
      j += inc;
}
```

In Pascal, the resulting operation is

```
j := 1;
for i := 1 to length do
      begin
      vector[i] := 0;
      j := j + inc;
      end
```

In FORTRAN, the resulting operation is

```
      j = 1
      do 10 i = 1, length
           vec(j) = 0
           j = j + inc
 10   continue
```

vector The vector to be zeroed.

inc An increment for the index of *vector* that chooses the elements to be zeroed.

length The number of elements in *vector* to zero.

NOTES

In C and Pascal, vec_$izero_i zeros column vectors; whereas in FORTRAN, it zeros row vectors.

SEE ALSO

vec_$dzero_i, vec_$izero, vec_$izero16_i, vec_$zero_i.

NAME

vec_$mat_mult – multiply two 4×4 single-precision matrixes

SYNOPSIS (C)

#include <apollo/base.h>
#include <apollo/vec.h>

void vec_$mat_mult(
float **matrix1*,
float **matrix2*,
float **out_matrix*)

SYNOPSIS (Pascal)

%include '/sys/ins/base.ins.pas';
%include '/sys/ins/vec.ins.pas';

procedure vec_$mat_mult(
in *matrix1*: **univ vec_$real_matrix;**
in *matrix2*: **univ vec_$real_matrix;**
out *out_matrix*: **univ vec_$real_matrix);**

SYNOPSIS (FORTRAN)

%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/vec.ins.ftn'

real *matrix1*(4, 4), *matrix2*(4, 4), *out_matrix*(4, 4)

call vec_$mat_mult(*matrix1*, *matrix2*, *out_matrix*)

DESCRIPTION

Vec_$mat_mult multiplies two 4×4 matrixes, *matrix1* and *matrix2*, and supplies the result in *out_matrix*.

In C, **vec_$mat_mult** calculates the product of *matrix2* on the left and *matrix1* on the right, and the resulting operation is

```
for (i = 0; i < 4; ++i)
      for (j = 0; j < 4; ++j) {
            out_mat[j,i] = 0.0;
            for (k = 0; k < 4; ++k)
                  out_mat[j][i] += matrix1[k][i]
                                 * matrix2[j][k];
      }
```

In Pascal, **vec_$mat_mult** calculates the product of *matrix2* on the left and *matrix1* on the right, and the resulting operation is

```
for i := 1 to 4 do
     for j := 1 to 4 do
          begin
          out_mat[j,i] := 0.0;
          for k := 1 to 4 do
               out_mat[j,i] := out_mat[j,i]
                             + matrix1[k,i]
                             * matrix2[j,k];
          end;
```

In FORTRAN, **vec_$mat_mult** calculates the product of *matrix1* on the left and *matrix2* on the right, and the resulting operation is

```
      do 10 i = 1, 4
           do 10 j = 1, 4
                out_mat(i,j) = 0.0
                do 10 k = 1, 4
                     out_mat(i,j) = out_mat(i,j)
     &                            + matrix1(i,k)
     &                            * matrix2(k,j)
 10   continue
```

matrix1 A 4×4 matrix.

matrix2 Another 4×4 matrix.

out_matrix
The product of *matrix1* and *matrix2*.

NOTES

Vec_$mat_multn performs the same operations for variably dimensioned matrixes.

SEE ALSO

vec_$dmat_mult, vec_$imat_mult, vec_$imat_mult16.

NAME

vec_$mat_multn – multiply two single-precision matrixes

SYNOPSIS (C)

```
#include <apollo/base.h>
#include <apollo/vec.h>

void vec_$mat_multn(
        float *matrix1,
        float *matrix2,
        long int &m,
        long int &n,
        long int &s,
        float *out_matrix)
```

SYNOPSIS (Pascal)

```
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/vec.ins.pas';

procedure vec_$mat_multn(
        in matrix1: univ vec_$real_matrix;
        in matrix2: univ vec_$real_matrix;
        in m: integer32;
        in n: integer32;
        in s: integer32;
        out out_matrix: univ vec_$real_matrix);
```

SYNOPSIS (FORTRAN)

```
%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/vec.ins.ftn'

        integer*4 m, n, s
        parameter (m = 3, n = 4, s = 5)

        real matrix1(m, n), matrix2(n, s), out_matrix(m, s)

        call vec_$mat_multn(matrix1, matrix2, m, n, s, out_matrix)
```

DESCRIPTION

Vec_$mat_multn multiplies two variably dimensioned, single-precision matrixes, *matrix1* and *matrix2*, and supplies the result in *out_matrix*.

In C, **vec_$mat_multn** calculates the product of *matrix2* on the left and *matrix1* on the right, and the resulting operation is

```
for (i = 0; i < m; ++i)
      for (j = 0; j < s; ++j) {
            out_matrix[j][i] = 0.0;
            for (k = 0; k < n; ++k)
                  out_matrix[j][i] += matrix1[k][i]
                                    * matrix2[j][k];
      }
```

In Pascal, **vec_$mat_multn** calculates the product of *matrix2* on the left and *matrix1* on the right, and the resulting operation is

```
for i := 1 to m do
      for j := 1 to s do
            begin
            out_matrix[j,i] = 0.0;
            for k := 1 to n do
                  out_matrix[j,i] := out_matrix[j,i]
                                   + matrix1[k,i]
                                   * matrix2[j,k];
            end;
```

In FORTRAN, **vec_$mat_multn** calculates the product of *matrix1* on the left and *matrix2* on the right, and the resulting operation is

```
       do 10 i = 1, m
             do 10 j = 1, s
                   out_matrix(i,j) = 0.0
                   do 10 k = 1, n
                         out_matrix(i,j) = out_matrix(i,j)
     &                                   + matrix1(i,k)
     &                                   * matrix2(k,j)
  10   continue
```

matrix1 A matrix to be multiplied.

matrix2 Another matrix to be multiplied.

m, n, s The various matrix dimensions.

out_matrix
The product of *matrix1* and *matrix2*.

SEE ALSO
vec_$dmat_multn, vec_$imat_multn, vec_$imat_multn16.

NAME

vec_$max – find the maximum absolute value in a single-precision vector

SYNOPSIS (C)

```
#include <apollo/base.h>
#include <apollo/vec.h>

void vec_$max(
        float *vector,
        long int &length,
        float *result,
        long int *location)
```

SYNOPSIS (Pascal)

```
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/vec.ins.pas';

procedure vec_$max(
        in vector: univ vec_$real_vector;
        in length: integer32;
        out result: real;
        out location: integer32);
```

SYNOPSIS (FORTRAN)

```
%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/vec.ins.ftn'

      parameter (nvec = 10)

      real vector(nvec), result
      integer*4 length, location

      call vec_$max(vector, length, result, location)
```

DESCRIPTION

Vec_$max searches through *length* elements of *vector*, and supplies the value and location of the element with the greatest absolute value.

In C, the resulting operation is

```
result = (float)fabs(vector[0]);
location = 1;
for (i = 1; i < length; ++i)
      if ((float)fabs(vector[i]) > result) {
            location = i + 1;
            result = (float)fabs(vector[i]);
      }
```

In Pascal, the resulting operation is

```
result := abs(vector[1]);
location = 1;
for 10 i := 2 to length do
      if (abs(vector[i]) > result) then
            begin
            location := i;
            result := abs(vector[i]);
            end
```

In FORTRAN, the resulting operation is

```
      result = abs(vector(1))
      location = 1
      do 10 i = 2, length
           if (abs(vector(i)) .gt. result) then
                 location = i
                 result = abs(vector(i))
           endif
  10  continue
```

vector The vector to search.

length The number of elements to search.

result The maximum absolute value of all the elements searched.

location The location of the element with the greatest absolute value. The location supplied in *location* is just the index of the element with the greatest absolute value in FORTRAN or Pascal (if *vector* is declared to begin with index 1). In C, *location* - 1 is the index of the element.

NOTES

In C and Pascal, vec_$max searches a row vector; whereas in FORTRAN, it searches a column vector.

SEE ALSO

vec_$dmax, vec_$imax, vec_$imax16, vec_$max_i.

NAME

vec_$max_i – find the maximum absolute value in a vector from a single-precision matrix

SYNOPSIS (C)

```
#include &lt;apollo/base.h&gt;
#include &lt;apollo/vec.h&gt;

void vec_$max_i(
        float *vector,
        long int &inc,
        long int &length,
        float *result,
        long int *location)
```

SYNOPSIS (Pascal)

```
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/vec.ins.pas';

procedure vec_$max_i(
        in vector: univ vec_$real_vector;
        in inc: integer32;
        in length: integer32;
        out result: real;
        out location: integer32);
```

SYNOPSIS (FORTRAN)

```
%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/vec.ins.ftn'

        parameter (nvec = 10)

        real vector(nvec), result
        integer*4 length, inc, location

        call vec_$max_i(vector, inc, length, result, location)
```

DESCRIPTION

Vec_$max_i searches through the *length* elements of *vector* selected by *inc*, and supplies the value and location of the element with the greatest absolute value.

Through appropriate choice of *inc*, a program can use **vec_$max_i** to search a vector within a matrix. To search the *M*th vector in a matrix, choose *inc* equal to the number of vectors in the matrix, and place the *M*th element of the matrix array at the beginning of *vector*.

In C, the resulting operation is

```
result = (float)fabs(vector[0]);
location = 1;
j = inc;
for (i = 1; i < length, ++i) {
      if ((float)fabs(vector[j]) > result) {
            location = i + 1;
            result = (float)fabs(vector[j]);
      }
      j += inc;
}
```

In Pascal, the resulting operation is

```
result := abs(vector[1]);
location := 1;
j := 1 + inc;
for 10 i := 2 to length do
      begin
      if (abs(vector[j]) > result) then
            begin
            location := i;
            result := abs(vector[j]);
            end
      j := j + inc;
      end
```

In FORTRAN, the resulting operation is

```
       result = abs(vector(1))
       location = 1
       j = 1 + inc
       do 10 i = 2, length
            if (abs(vector(j)) .gt. result) then
                  location = i
                  result = abs(vector(j))
            endif
       j = j + inc
  10   continue
```

vector The array to search.

inc An increment for the index of *vector* that selects the elements to search.

length The number of elements to search.

result The maximum absolute value of all the elements searched.

location The location of the element with the greatest absolute value. The location supplied in *location* is just the index of the element with the greatest absolute value in FORTRAN or Pascal (if *vector* is declared to begin with index 1). In C, *location* - 1 is the index of the element.

NOTES

In C and Pascal, **vec_$max_i** searches a column vector; whereas in FORTRAN, it searches a row vector.

SEE ALSO

vec_$dmax_i, vec_$imax16_i, vec_$imax_i, vec_$max.

NAME

vec_$mult_add – scale and add one single-precision vector to another

SYNOPSIS (C)

```
#include <apollo/base.h>
#include <apollo/vec.h>

void vec_$mult_add(
        float *add_vec,
        float *mult_vec,
        long int &length,
        float &constant,
        float *result_vec)
```

SYNOPSIS (Pascal)

```
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/vec.ins.pas';

procedure vec_$mult_add(
        in add_vec: univ vec_$real_vector;
        in mult_vec: univ vec_$real_vector;
        in length: integer32;
        in constant: real;
        out result_vec: univ vec_$real_vector);
```

SYNOPSIS (FORTRAN)

```
%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/vec.ins.ftn'

      parameter (nvec = 10)

      real add_vec(nvec), mult_vec(nvec), result_vec(nvec), constant
      integer*4 length

      call vec_$mult_add(add_vec, mult_vec, length,
     &                   constant, result_vec)
```

DESCRIPTION

Vec_$mult_add multiplies the vector *mult_vec* by the scalar *constant*, adds the product to the vector *add_vec*, and supplies the resulting vector in *result_vec*.

In C, the resulting operation is

```
for (i = 0; i< length; ++i)
     result_vec[i] = add_vec[i]
                   + constant * mult_vec[i];
```

In Pascal, the resulting operation is

```
for i := 1 to length do
      result_vec[i] := add_vec[i]
                     + constant * mult_vec[i];
```

In FORTRAN, the resulting operation is

```
       do 10 i = 1, length
             result_vec(i) = add_vec(i)
      &                    + constant * mult_vec(i)
  10   continue
```

add_vec
: The vector to add to the product of *mult_vec* and *constant*.

mult_vec
: The vector to scale by *constant* and add to *add_vec*.

length
: The number of elements to use in the calculation.

constant
: The scalar value used to scale *mult_vec*.

result_vec
: The vector resulting from multiplying *mult_vec* by *constant* and adding the product to *add_vec*.

NOTES

When **vec_$mult_add** is used to operate on matrixes in C and Pascal, *add_vec*, *mult_vec*, and *result_vec* are row vectors; whereas in FORTRAN, they are column vectors.

SEE ALSO

vec_$dmult_add, vec_$imult_add, vec_$imult_add16, vec_$mult_add_i.

NAME

vec_$mult_add_i – scale and add single-precision vectors in matrixes

SYNOPSIS (C)

```
#include <apollo/base.h>
#include <apollo/vec.h>

void vec_$mult_add_i(
        float *add_vec,
        long int &inc1,
        float *mult_vec,
        long int &inc2,
        long int &length,
        float &constant,
        float *result_vec,
        long int &inc3)
```

SYNOPSIS (Pascal)

```
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/vec.ins.pas';

procedure vec_$mult_add_i(
        in add_vec: univ vec_$real_vector;
        in inc1: integer32;
        in mult_vec: univ vec_$real_vector;
        in inc2: integer32;
        in length: integer32;
        in constant: real;
        out result_vec: univ vec_$real_vector;
        in inc3: integer32);
```

SYNOPSIS (FORTRAN)

```
%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/vec.ins.ftn'

      parameter (nvec = 10)

      real add_vec(nvec), mult_vec(nvec), result_vec(nvec), constant
      integer*4 length, inc1, inc2, inc3

      call vec_$mult_add_i(add_vec, inc1, mult_vec, inc2, length,
     &                     constant, result_vec, inc3)
```

DESCRIPTION

Vec_$mult_add_i multiplies *length* elements of *mult_vec* selected by *inc2* by the scalar *constant*, adds the resulting products to elements of *add_vec* selected by *inc1*, and supplies the results in elements of *result_vec* selected by *inc3*.

Through appropriate choice of *inc1*, *inc2*, and *inc3*, a program can use **vec_$mult_add_i** to scale and sum individual vectors in two matrixes and place the result in a vector of another matrix. To scale and add the *N*th vector in matrix *Y* to the *M*th vector in matrix *X*, choose *inc2* equal to the number of vectors in matrix *Y* and *inc1* equal to the number of vectors in matrix *X*. Then place the *M*th element of matrix *X* at the beginning of *add_vec*, and place the *N*th element of matrix *Y* at the beginning of *mult_vec*. To place the result of the operation in the *P*th vector of a resultant matrix, choose *inc3* equal to the number of vectors in the resultant matrix, and place the *P*th element of the matrix array at the beginning of *result_vec*.

In C, the resulting operation is

```
j = 0;
k = 0;
l = 0;
for (i = 0; i< length; ++i) {
      result_vec[l] = add_vec[j]
                    + constant * mult_vec[k];
      j += inc1;
      k += inc2;
      l += inc3;
}
```

In Pascal, the resulting operation is

```
j := 1;
k := 1;
l := 1;
for i := 1 to length do
      begin
      result_vec[l] := add_vec[j]
                    + constant * mult_vec[k];
      j := j + inc1;
      k := k + inc2;
      l := l + inc3;
      end
```

In FORTRAN, the resulting operation is

```
         j = 1
         k = 1
         l = 1
         do 10 i = 1, length
              result(l) = add_vec(j)
        &                 + constant * mult_vec(k)
              j = j + inc1
              k = k + inc2
              l = l + inc3
   10    continue
```

add_vec
The vector to add to the product of *mult_vec* and *constant*.

inc1 An increment for the index of *add_vec* that selects the elements to add to the product of *mult_vec* and *constant*.

mult_vec
The vector to scale by *constant* and add to *add_vec*.

inc2 An increment for the index of *mult_vec* that selects the elements to multiply by *constant* and add to *add_vec*.

length The number of elements to use in the calculation.

constant
The scalar value used to scale *mult_vec*.

result_vec
The vector resulting from multiplying *mult_vec* by *constant* and adding the product to *add_vec*.

inc3 An increment for the index of *result_vec* that selects the elements to receive the results of multiplying *mult_vec* by *constant* and adding the product to *add_vec*.

NOTES

In C and Pascal, vec_**$mult_add_i** operates on column vectors; whereas in FORTRAN, it operates on row vectors.

SEE ALSO

vec_$dmult_add_i, vec_$imult_add16_i, vec_$imult_add_i, vec_$mult_add.

NAME

vec_$mult_constant – multiply a single-precision vector by a scalar

SYNOPSIS (C)

#include <apollo/base.h>
#include <apollo/vec.h>

void vec_$mult_constant(
    float **mult_vec*,
    long int *&length*,
    float *&constant*,
    float **result_vec*)

SYNOPSIS (Pascal)

%include '/sys/ins/base.ins.pas';
%include '/sys/ins/vec.ins.pas';

procedure vec_$mult_constant(
    in *mult_vec*: **univ vec_$real_vector;**
    in *length*: **integer32;**
    in *constant*: **real;**
    out *result_vec*: **univ vec_$real_vector);**

SYNOPSIS (FORTRAN)

%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/vec.ins.ftn'

parameter (*nvec* = **10**)

real *mult_vec*(*nvec*), *result_vec*(*nvec*), *constant*
integer*4 *length*

call vec_$mult_constant(*mult_vec*, *length*, *constant*, *result_vec*)

DESCRIPTION

Vec_$mult_constant multiplies the single-precision array *mult_vec* by the scalar value *constant* and supplies the result in the array *result_vec*.

In C, the resulting operation is

```
for (i = 0; i < length; ++i)
     result_vec[i] = constant * start_vec[i];
```

In Pascal, the resulting operation is

```
for i := 1 to length do
      begin
      result_vec[i] := constant * start_vec[i];
      end
```

In FORTRAN, the resulting operation is

```
       do 10 i = 1, length
            result_vec(i) = constant * mult_vec(i)
  10   continue
```

mult_vec
The vector to multiply by *constant*.

length The number of elements to multiply.

constant
The scalar constant to multiply *mult_vec* by.

result_vec
The vector resulting from multiplying *mult_vec* by *constant*.

NOTES

When vec_**$mult_constant** is used to operate on matrixes in C and Pascal, *mult_vec* and *result_vec* are row vectors; whereas in FORTRAN, they are column vectors.

SEE ALSO

vec_$dmult_constant, vec_$imult_constant, vec_$imult_constant16, vec_$mult_constant_i.

NAME

vec_$mult_constant_i – multiply a vector in a single-precision matrix by a scalar

SYNOPSIS (C)

```
#include <apollo/base.h>
#include <apollo/vec.h>

void vec_$mult_constant_i(
        float *mult_vec,
        long int &inc1,
        long int &length,
        float &constant,
        float *result_vec,
        long int &inc2)
```

SYNOPSIS (Pascal)

```
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/vec.ins.pas';

procedure vec_$mult_constant_i(
        in mult_vec: univ vec_$real_vector;
        in inc1: integer32;
        in length: integer32;
        in constant: real;
        out result_vec: univ vec_$real_vector;
        in inc2: integer32);
```

SYNOPSIS (FORTRAN)

```
%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/vec.ins.ftn'

      parameter (nvec = 10)

      real mult_vec(nvec), result_vec(nvec), constant
      integer*4 length, inc1, inc2

      call vec_$mult_constant_i(mult_vec, inc1, length, constant,
     &                          result_vec, inc2)
```

DESCRIPTION

Vec_$mult_constant_i multiplies *length* elements of the single-precision array *mult_vec* selected by *inc1* by the scalar value *constant*, and supplies the result in elements of the array *result_vec* selected by *inc2*.

Through appropriate choice of *inc1* and *inc2*, a program can use **vec_$mult_constant_i** to operate on a vector in a matrix. To multiply *M*th vector in a matrix by *constant*, choose *inc1* equal to the number of vectors in the matrix, and place the *M*th element of the matrix array at the beginning of *mult_vec*. To place the result of the operation in the

*N*th vector of a matrix, choose *inc2* equal to the number of vectors in the resultant matrix, and place the *N*th element of the matrix array at the beginning of *result_vec*.

In C, the resulting operation is

```
j = 0;
k = 0;
for (i = 0; i < length; ++i) {
      result_vec[k] = constant * mult_vec[j];
      k += inc2;
      j += inc1;
}
```

In Pascal, the resulting operation is

```
j := 1;
k := 1;
for i := 1 to length do
      begin
      result_vec[k] := constant * mult_vec[j];
      k := k + inc2;
      j := j + inc1;
      end
```

In FORTRAN, the resulting operation is

```
      j = 1
      k = 1
      do 10 i = 1, length
           result(j) = constant * mult_vec(k)
           j = j + inc2
           k = k + inc1
 10   continue
```

mult_vec
The vector to multiply by *constant*.

inc1 An increment for the index of the array *mult_vec* that selects elements to multiply by *constant*.

length The number of products to calculate.

constant
The scalar constant to multiply elements of *mult_vec* by.

result_vec
An array whose elements receive the product of *constant* and *mult_vec*.

inc2 An increment for the index of the array *result_vec* that selects elements to receive the product of *constant* and *mult_vec*.

NOTES

In C and Pascal, **vec_$mult_constant_i** operates on column vectors; whereas in FORTRAN, it operates on row vectors.

SEE ALSO

vec_$dmult_constant_i, vec_$imult_constant16_i, vec_$imult_constant_i, vec_$mult_constant.

NAME

vec_$postmult – multiply a single-precision vector by a 4×4 matrix

SYNOPSIS (C)

#include <apollo/base.h>
#include <apollo/vec.h>

void vec_$postmult(
    float **matrix*,
    float **start_vec*,
    float **result_vec*)

SYNOPSIS (Pascal)

%include '/sys/ins/base.ins.pas';
%include '/sys/ins/vec.ins.pas';

procedure vec_$postmult(
    in *matrix*: **univ vec_$real_matrix;**
    in *start_vec*: **univ vec_$real_vector;**
    out *result_vec*: **univ vec_$real_vector);**

SYNOPSIS (FORTRAN)

%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/vec.ins.ftn'

real *matrix*(4, 4), *start_vec*(4), *result_vec*(4)

call vec_$postmult(*matrix*, *start_vec*, *result_vec*)

DESCRIPTION

Vec_$postmult multiplies the 4-element vector *start_vec* by the 4×4 matrix *matrix*.

In C, **vec_$postmult** applies *matrix* as a right transform to a row vector *start_vec*, and the resulting operation is

```
for (j = 0; j < 4; ++j) {
      result_vec[j] = 0.0;
      for (i = 0; i < 4; ++i)
            result_vec[j] += start_vec[i]
                           * matrix[i][j];
}
```

In Pascal, **vec_$postmult** applies *matrix* as a right transform to a row vector *start_vec*, and the resulting operation is

```
for j := 1 to 4 do
      begin
      result_vec[j] := 0.0;
      for i := 1 to 4 do
            result_vec[j] := result_vec[j]
                           + start_vec[i]
                           * matrix[i,j];
      end
```

In FORTRAN, **vec_$postmult** applies *matrix* as a left transform to a column vector *start_vec*, and the resulting operation is

```
      do 10 j = 1, 4
            result_vec(j) = 0.0
            do 10 i = 1, 4
                  result_vec(j) = result_vec(j)
     &                          + start_vec(i)
     &                          * matrix(j,i)
  10  continue
```

matrix The matrix to multiply by *start_vec*.

start_vec
The vector to multiply by *matrix*.

result_vec
The product of *start_vec* and *matrix*.

NOTES

Vec_$premult transforms single-precision vectors from the other side.

SEE ALSO

vec_$dpostmult, vec_$ipostmult, vec_$ipostmult16.

NAME

vec_$postmultn – multiply a single-precision vector by a matrix

SYNOPSIS (C)

```
#include <apollo/base.h>
#include <apollo/vec.h>

void vec_$postmultn(
        float *matrix,
        float *start_vec,
        long int &m,
        long int &n,
        float *result_vec)
```

SYNOPSIS (Pascal)

```
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/vec.ins.pas';

procedure vec_$postmultn(
        in matrix: univ vec_$real_matrix;
        in start_vec: univ vec_$real_vector;
        in m: integer32;
        in n: integer32;
        out result_vec: univ vec_$real_vector);
```

SYNOPSIS (FORTRAN)

```
%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/vec.ins.ftn'

        integer*4 m, n
        parameter (m = 3, n = 4)

        real matrix(m, n), start_vec(n), result_vec(m)

        call vec_$postmultn(matrix, start_vec, m, n, result_vec)
```

DESCRIPTION

Vec_$postmultn multiplies the *n*-element vector *start_vec* by the variably dimenstioned matrix *matrix*, and supplies the resulting *m*-element vector in *result_vec*.

In C, **vec_$postmultn** applies the $n \times m$ matrix *matrix* as a right transform to the *m*-element row vector *start_vec*, and supplies the transformed *n*-element result in *result_vec*:

```
for (i = 0; i < m; ++i) {
      result_vec[i] = 0.0;
      for (j = 0; j < n; ++j)
            result_vec[i] += start_vec[j]
                          * matrix[j][i];
}
```

In Pascal, **vec_$postmultn** applies the $n \times m$ matrix *matrix* as a right transform to the m-element row vector *start_vec*, and supplies the transformed n-element result in *result_vec*:

```
for i := 1 to m do
      begin
      result_vec[i] := 0.0;
      for j := 1 to n do
            result_vec[i] := result_vec[i]
                          + start_vec[j]
                          * matrix[j,i];
      end;
```

In FORTRAN, **vec_$postmultn** applies the $m \times n$ matrix *matrix* as a left transform to the m-element column vector *start_vec*, and supplies the transformed n-element result in *result_vec*:

```
       do 10 i = 1, m
            result_vec(i) = 0.0
            do 10 j = 1, n
                  result_vec(i) = result_vec(i)
     &                          + start_vec(j)
     &                          * matrix(i,j)
  10   continue
```

matrix A matrix to multiply *start_vec* by.

start_vec
An n-element vector to multiply by *matrix*.

m The number of elements in *start_vec*.

n The number of elements in *result_vec*.

result_vec
An m-element vector that is the product of *matrix* and *start_vec*.

NOTES

Vec_$premultn transforms single-precision vectors from the other side.

SEE ALSO

vec_$dpostmultn, vec_$ipostmultn, vec_$ipostmultn16.

NAME

vec_$premult – multiply a single-precision vector by a 4×4 matrix

SYNOPSIS (C)

#include <apollo/base.h>
#include <apollo/vec.h>

void vec_$premult(
  float **start_vec*,
  float **matrix*,
  float **result_vec*)

SYNOPSIS (Pascal)

%include '/sys/ins/base.ins.pas';
%include '/sys/ins/vec.ins.pas';

procedure vec_$premult(
  in *start_vec*: **univ vec_$real_vector;**
  in *matrix*: **univ vec_$real_matrix;**
  out *result_vec*: **univ vec_$real_vector);**

SYNOPSIS (FORTRAN)

%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/vec.ins.ftn'

  real *start_vec*(**4**), *matrix*(**4,4**), *result_vec*(**4**)

  call vec_$premult(*start_vec*, *matrix*, *result_vec***)**

DESCRIPTION

Vec_$premult multiplies the 4-element vector *start_vec* by the 4×4 matrix *matrix*.

In C, **vec_$premult** applies *matrix* as a left transform to a column vector *start_vec*, and the resulting operation is

```
for (i = 0; i < 4; ++i) {
      result_vec[i] = 0.0;
      for (j = 0; i < 4; ++j)
            result_vec[i] += start_vec[i]
                          * matrix[i][j];
}
```

In Pascal, **vec_$premult** applies *matrix* as a left transform to a column vector *start_vec*, and the resulting operation is

```
for i := 1 to 4 do
      begin
      result_vec[i] := 0.0;
      for j := 1 to 4 do
            result_vec[i] := result_vec[i]
                           + start_vec[i]
                           * matrix[i,j];
      end
```

In FORTRAN, **vec_$premult** applies *matrix* as a right transform to a row vector *start_vec*, and the resulting operation is

```
       do 10 i = 1, 4
            result_vec(i) = 0.0
            do 10 j = 1, 4
                  result_vec(i) = result_vec(i)
                                + start_vec(j)
                                * matrix(j,i)
   10  continue
```

start_vec
The vector to multiply by *matrix*.

matrix The matrix to multiply by *start_vec*.

result_vec
The product of *start_vec* and *matrix*.

NOTES

Vec_$postmult transforms single-precision vectors from the other side.

SEE ALSO

vec_$dpremult, vec_$ipremult, vec_$ipremult16.

NAME

vec_$premultn – multiply a single-precision vector by a matrix

SYNOPSIS (C)

```
#include &lt;apollo/base.h&gt;
#include &lt;apollo/vec.h&gt;

void vec_$premultn(
        float *start_vec,
        float *matrix,
        long int &m,
        long int &n,
        float *result_vec)
```

SYNOPSIS (Pascal)

```
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/vec.ins.pas';

procedure vec_$premultn(
        in start_vec: univ vec_$real_vector;
        in matrix: univ vec_$real_matrix;
        in m: integer32;
        in n: integer32;
        out result_vec: univ vec_$real_vector);
```

SYNOPSIS (FORTRAN)

```
%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/vec.ins.ftn'

          integer*4 m, n
          parameter (m = 3, n = 4)

          real start_vec(m), matrix(m, n), result_vec(n)

          call vec_$premultn(start_vec, matrix, m, n, result_vec)
```

DESCRIPTION

Vec_$premultn multiplies the *m*-element vector *start_vec* by the variably dimenstioned matrix *matrix*, and supplies the resulting *n*-element vector in *result_vec*.

In C, **vec_$premultn** applies the $n \times m$ matrix *matrix* as a left transform to the *m*-element column vector *start_vec*, and supplies the transformed *n*-element result in *result_vec*:

```
for (i = 0; i < n; ++i) {
      result_vec[i] = 0.0;
      for (j = 0; i < m; ++i)
            result_vec[i] += start_vec[j]
                           * matrix[i][j];
}
```

In Pascal, **vec_$premultn** applies the $n \times m$ matrix *matrix* as a left transform to the m-element column vector *start_vec*, and supplies the transformed n-element result in *result_vec*:

```
for i := 1 to n do
      begin
      result_vec[i] := 0.0;
      for j := 1 to m do
            result_vec[i] = result_vec[i]
                          + start_vec[j]
                          * matrix[i,j];
      end
```

In FORTRAN, **vec_$premultn** applies the $m \times n$ matrix *matrix* as a right transform to the m-element row vector *start_vec*, and supplies the transformed n-element result in *result_vec*:

```
       do 10 i = 1, n
            result_vec(i) = 0.0
            do 10 j = 1, m
                 result_vec(i) = result_vec(i)
     &                         + start_vec(j)
     &                         * matrix(j,i)
  10   continue
```

start_vec
: An m-element vector to multiply by *matrix*.

matrix A matrix to multiply *start_vec* by.

m The number of elements in *start_vec*.

n The number of elements in *result_vec*.

result_vec
: An n-element vector that is the product of *matrix* and *start_vec*.

NOTES

Vec_$postmultn transforms single-precision vectors from the other side.

SEE ALSO

vec_$dpremultn, vec_$ipremultn, vec_$ipremultn16.

NAME

vec_$sp_dp – copy a single-precision vector to a double-precision vector

SYNOPSIS (C)

```
#include <apollo/base.h>
#include <apollo/vec.h>

void vec_$sp_dp(
        float *sp_vec,
        double *dp_vec,
        long int &length)
```

SYNOPSIS (Pascal)

```
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/vec.ins.pas';

procedure vec_$sp_dp(
        in sp_vec: univ vec_$real_vector;
        in dp_vec: univ vec_$double_vector;
        in length: integer32);
```

SYNOPSIS (FORTRAN)

```
%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/vec.ins.ftn'

        parameter (nvec = 10)

        double precision dp_vec(nvec)
        real sp_vec(nvec)
        integer*4 length

        call vec_$sp_dp(sp_vec, dp_vec, length)
```

DESCRIPTION

Vec_$sp_dp copies *length* elements from the single-precision vector *sp_vec* to the double-precision vector *dp_vec*.

In C, the resulting operation is

```
for (i = 0; i < length; ++i)
     dp_vec[i] = (double)sp_vec[i];
```

In Pascal, the resulting operation is

```
for i := 1 to length do
      begin
      dp_vec[i] := sp_vec[i];
      end
```

In FORTRAN, the resulting operation is

```
       do 10 i=1, length
          dp_vec(i) = dble(sp_vec(i))
  10   continue
```

sp_vec The single-precision vector to copy from.

dp_vec The double-precision vector to copy to.

length The number of elements to copy.

NOTES

In C and Pascal, **vec_$sp_dp** copies row vectors; whereas in FORTRAN, it copies column vectors.

SEE ALSO

vec_$dp_sp, vec_$sp_dp_i.

NAME

vec_$sp_dp_i – copy a single-precision vector to a double-precision vector in matrixes

SYNOPSIS (C)

```
#include <apollo/base.h>
#include <apollo/vec.h>

void vec_$sp_dp_i(
        float *sp_vec,
        long int &inc1,
        double *dp_vec,
        long int &inc2,
        long int &length)
```

SYNOPSIS (Pascal)

```
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/vec.ins.pas';

procedure vec_$sp_dp_i(
        in sp_vec: univ vec_$real_vector;
        in inc1: integer32;
        in dp_vec: univ vec_$double_vector;
        in inc2: integer32;
        in length: integer32) ;
```

SYNOPSIS (FORTRAN)

```
%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/vec.ins.ftn'

        parameter (nvec = 10)

        double precision dp_vec(nvec)
        real sp_vec(nvec)
        integer*4 length, inc1, inc2

        call vec_$sp_dp_i(sp_vec, inc1, dp_vec, inc2, length)
```

DESCRIPTION

Vec_$sp_dp_i copies *length* elements selected by *inc1* from the single-precision array *sp_vec* to elements of the double-precision array *dp_vec* chosen by *inc2*.

Through appropriate choice of *inc1* and *inc2*, a program can use **vec_$sp_dp_i** to copy a vector from one matrix to another. To copy the *M*th vector in a matrix, choose *inc1* equal to the number of vectors in the matrix, and place the *M*th element of the matrix array at the beginning of *sp_vec*. To place the copy into the *N*th vector of a matrix, choose *inc2* equal to the number of vectors in the resultant matrix, and place the *N*th element of the matrix array at the beginning of *dp_vec*.

In C, the resulting operation is

```
j = 0;
k = 0;
for (i = 0; i < length; ++i) {
      dp_vec[i] = (double)sp_vec[i];
      j += inc1;
      k += inc2;
}
```

In Pascal, the resulting operation is

```
j := 1;
k := 1;
for i := 1 to length do
      begin
      dp_vec[i] := sp_vec[i];
      j := j + inc1;
      k := k + inc2;
      end
```

In FORTRAN, the resulting operation is

```
       j = 1
       k = 1
       do 10 i = 1, length
            dp_vec(k) = dble(sp_vec(j))
            j = j + incl
            k = k + inc2
  10   continue
```

sp_vec The single-precision array to copy from.

inc1 Increment for the index of *sp_vec* that selects the elements to copy from.

dp_vec The double-precision array to copy to.

inc2 Increment for the index of *dp_vec* that selects the elements to copy to.

length The number of elements to copy from *sp_vec* to *dp_vec*.

NOTES

In C and Pascal, **vec_$sp_dp_i** copies column vectors; whereas in FORTRAN, it copies row vectors.

SEE ALSO

vec_$dp_sp_i, vec_$sp_dp.

NAME

vec_$sub – subtract single-precision vectors

SYNOPSIS (C)

```
#include <apollo/base.h>
#include <apollo/vec.h>

void vec_$sub(
        float *start_vec,
        float *sub_vec,
        long int &length,
        float *result_vec)
```

SYNOPSIS (Pascal)

```
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/vec.ins.pas';

procedure vec_$sub(
        in start_vec: univ vec_$real_vector;
        in sub_vec: univ vec_$real_vector;
        in length: integer32;
        out result_vec: univ vec_$real_vector);
```

SYNOPSIS (FORTRAN)

```
%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/vec.ins.ftn'

        parameter (nvec = 10)

        real start_vec(nvec), sub_vec(nvec), result_vec(nvec)
        integer*4 length

        call vec_$sub(start_vec, sub_vec, length, result_vec)
```

DESCRIPTION

Vec_$sub subtracts *length* elements of *sub_vec* from *start_vec*, and supplies the difference in *result_vec*.

In C, the resulting operation is

```
for (i = 0; i < length; ++i)
     result_vec[i] = start_vec[i] - sub_vec[i];
```

In Pascal, the resulting operation is

```
for i := 1 to length do
     result_vec[i] := start_vec[i] - sub_vec[i];
```

In FORTRAN, the resulting operation is

```
      do 10 i = 1, length
           result_vec(i) = start_vec(i) - sub_vec(i)
 10   continue
```

start_vec
The vector to subtract *sub_vec* from.

sub_vec The vector to subtract from *start_vec*.

length The number of differences to calculate.

result_vec
The difference of *start_vec* and *sub_vec*.

NOTES

When **vec_$sub** is used to operate on matrixes in C and Pascal, *start_vec*, *result_vec*, and *result_vec* are row vectors; whereas in FORTRAN, they are column vectors.

SEE ALSO

vec_$dsub, vec_$isub, vec_$isub16, vec_$sub_i.

NAME

vec_$sub_i – subtract single-precision vectors in matrixes

SYNOPSIS (C)

```
#include <apollo/base.h>
#include <apollo/vec.h>

void vec_$sub_i(
        float *start_vec,
        long int &inc1,
        float *sub_vec,
        long int &inc2,
        long int &length,
        float *result_vec,
        long int &inc3)
```

SYNOPSIS (Pascal)

```
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/vec.ins.pas';

procedure vec_$sub_i(
        in start_vec: univ vec_$real_vector;
        in inc1: integer32;
        in sub_vec: univ vec_$real_vector;
        in inc2: integer32;
        in length: integer32;
        out result_vec: univ vec_$real_vector;
        in inc3: integer32);
```

SYNOPSIS (FORTRAN)

```
%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/vec.ins.ftn'

      parameter (nvec = 10)

      real start_vec(nvec), sub_vec(nvec), result_vec(nvec)
      integer*4 length, inc1, inc2, inc3

      call vec_$sub_i(start_vec, inc1, sub_vec, inc2,
     &                length, result_vec, inc3)
```

DESCRIPTION

Vec_$sub_i subtracts *length* elements of *sub_vec* selected by *inc2* from *length* elements of *start_vec* selected by *inc1*, and supplies the result in the elements of *result_vec* selected by *inc3*.

Through appropriate choice of *inc1*, *inc2*, and *inc3*, a program can use **vec_$sub_i** to subtract individual vectors in two matrixes and place the difference in a vector of another

matrix. To subtract the *N*th vector in matrix *Y* from the *M*th vector in matrix *X*, choose *inc2* equal to the number of vectors in matrix *Y* and *inc1* equal to the number of vectors in matrix *X*. Then place the *M*th element of matrix *X* at the beginning of *start_vec*, and place the *N*th element of matrix *Y* at the beginning of *sub_vec*. To place the result of the operation in the *P*th vector of a resultant matrix, choose *inc3* equal to the number of vectors in the resultant matrix, and place the *P*th element of the matrix array at the beginning of *result_vec*.

In C, the resulting operation is

```
j = 0;
k = 0;
l = 0;
for (i = 0; i < length, ++i) {
      result_vec[j] = start_vec[k] - sub_vec[l];
      j += inc3;
      k += inc1;
      l += inc2;
}
```

In Pascal, the resulting operation is

```
j := 1;
k := 1;
l := 1;
for i := 1 to length do
      begin
      result_vec[j] := start_vec[k] - sub_vec[l];
      j := j + inc3;
      k := k + inc1;
      l := l + inc2;
      end
```

In FORTRAN, the resulting operation is

```
      j = 1
      k = 1
      l = 1
      do 10 i = 1, length
           result_vec(j) = start_vec(k) - sub_vec(l)
           j = j + inc3
           k = k + inc1
           l = l + inc2
10    continue
```

start_vec
The vector to subtract *sub_vec* from.

inc1 An increment for the index of *start_vec* that chooses which elements the elements of *sub_vec* will be subtracted from.

sub_vec The vector to subtract from *start_vec*.

inc2 An increment for the index of *sub_vec* that chooses which elements will be subtracted from the elements of *start_vec*.

length The number of scalar differences to calculate.

result_vec
The difference of *start_vec* and *sub_vec*.

inc3 An increment for the index of *result_vec* that chooses which elements will received the differences.

NOTES

In C and Pascal, **vec_$sub_i** operates on column vectors; whereas in FORTRAN, it operates on row vectors.

SEE ALSO

vec_$dsub_i, vec_$isub16_i, vec_$isub_i, vec_$sub.

NAME

vec_$sum – sum the elements of a single-precision vector

SYNOPSIS (C)

```
#include <apollo/base.h>
#include <apollo/vec.h>

float vec_$sum(
        float *vec,
        long int &length)
```

SYNOPSIS (Pascal)

```
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/vec.ins.pas';

function vec_$sum(
        in vec: univ vec_$real_vector;
        in length: integer32): real;
```

SYNOPSIS (FORTRAN)

```
%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/vec.ins.ftn'

        parameter (nvec = 10)

        real vec(nvec), sum
        integer*4 length

        sum = vec_$sum(vec, length)
```

DESCRIPTION

Vec_$sum adds together *length* elements of the single-precision array *vector* and returns the sum.

In C, the resulting operation is

```
return_value = 0.0;
for (i = 0; i < length; ++i)
      return_value += vec[i];
```

In Pascal, the resulting operation is

```
return_value := 0.0;
for i := 1 to length do
      return_value := return_value + vec[i];
```

In FORTRAN, the resulting operation is

```
      vec_$sum = 0.0
      do 10 i = 1, length
          vec_$sum = vec_$sum + vec(i)
  10  continue
```

vec The vector to sum.

length The number of elements in *vec* to sum.

NOTES

When vec_$**sum** is used to operate on matrixes in C and Pascal, *vec* is a row vector; whereas in FORTRAN, it is a column vector.

SEE ALSO

vec_$dsum, vec_$isum, vec_$isum16, vec_$sum_i.

NAME

vec_$sum_i – sum the elements of a vector in a single-precision matrix

SYNOPSIS (C)

```
#include &lt;apollo/base.h&gt;
#include &lt;apollo/vec.h&gt;

float vec_$sum_i(
        float *vec,
        long int &inc,
        long int &length)
```

SYNOPSIS (Pascal)

```
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/vec.ins.pas';

function vec_$sum_i(
        in vec: univ vec_$real_vector;
        in inc: integer32;
        in length: integer32): real;
```

SYNOPSIS (FORTRAN)

```
%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/vec.ins.ftn'

          parameter (nvec = 10)

          real vec(nvec), sum
          integer*4 length, inc

          sum = vec_$sum_i(vec, inc, length)
```

DESCRIPTION

Vec_$sum_i adds *length* elements from the single-precision array *vector* selected by *inc* and returns the sum.

Through appropriate choice of *inc*, a program can use **vec_$sum_i** to sum an individual vector in a matrix. To sum the *M*th vector in a matrix, choose *inc* equal to the number of vectors in the matrix.

In C, the resulting operation is

```
return_value = 0.0;
j = 0;
for (i = 0; i < length; ++i) {
      return_value += vec[j];
      j += inc;
}
```

In Pascal, the resulting operation is

```
return_value := 0.0;
j := 1;
for i := 1 to length do
      begin
      return_value := return_value + vec[j];
      j := j + inc;
      end
```

In FORTRAN, the resulting operation is

```
       vec_$sum_i = 0.0
       j = 1
       do 10 i = 1, length
            vec_$sum_i = vec_$sum_i + vec(j)
            j = j + inc
  10   continue
```

vec An array that contains the elements to sum.

inc An increment for the index of *vec* that selects which elements of *vec* are summed.

length The number of elements in *vec* to sum.

NOTES

In C and Pascal, **vec_$sum_i** sums a column vector; whereas in FORTRAN, it sums a row vector.

SEE ALSO

vec_$dsum_i, vec_$isum16_i, vec_$isum_i, vec_$sum.

NAME

vec_$swap – swap two single-precision vectors

SYNOPSIS (C)

#include <apollo/base.h>
#include <apollo/vec.h>

```
void vec_$swap(
        float *vec1,
        float *vec2,
        long int &length)
```

SYNOPSIS (Pascal)

%include '/sys/ins/base.ins.pas';
%include '/sys/ins/vec.ins.pas';

```
procedure vec_$swap(
        var vec1: univ vec_$real_vector;
        var vec2: univ vec_$real_vector;
        in length: integer32);
```

SYNOPSIS (FORTRAN)

%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/vec.ins.ftn'

```
        parameter (nvec = 10)

        real vec1(nvec), vec2(nvec)
        integer*4 length

        call vec_$swap(vec1, vec2, length)
```

DESCRIPTION

Vec_$swap swaps *length* elements between the single-precision vectors *vec1* and *vec2*.

In C, the resulting operation is

```
for (i = 0; i < length, ++i) {
      temp = vec1[i];
      vec1[i] = vec2[i];
      vec2[i] = temp;
}
```

In Pascal, the resulting operation is

```
for i := 1 to length do
      begin
      temp := vec1[i];
      vec1[i] := vec2[i];
      vec2[i] := temp;
      end
```

In FORTRAN, the resulting operation is

```
      do 10 i = 1, length
          temp = vec1(i)
          vec1(i) = vec2(i)
          vec2(i) = temp
10    continue
```

vec1 The vector to be swapped with *vec2*.

vec2 The vector to be swapped with *vec1*.

length The number of elements to swap.

NOTES

When vec_$swap is used to operate on matrixes in C and Pascal, *vec1* and *vec2* are row vectors; whereas in FORTRAN, they are column vectors.

SEE ALSO

vec_$dswap, vec_$iswap, vec_$iswap16, vec_$swap_i.

NAME

vec_$swap_i – swap two vectors in a single-precision matrix

SYNOPSIS (C)

```
#include <apollo/base.h>
#include <apollo/vec.h>

void vec_$swap_i(
        float *vec1,
        long int &inc1,
        float *vec2,
        long int &inc2,
        long int &length)
```

SYNOPSIS (Pascal)

```
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/vec.ins.pas';

procedure vec_$swap_i(
        var vec1: univ vec_$real_vector;
        in inc1: integer32;
        var vec2: univ vec_$real_vector;
        in inc2: integer32;
        in length: integer32);
```

SYNOPSIS (FORTRAN)

```
%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/vec.ins.ftn'

        parameter (nvec = 10)

        real vec1(nvec), vec2(nvec)
        integer*4 length, inc1, inc2

        call vec_$swap_i(vec1, inc1, vec2, inc2, length)
```

DESCRIPTION

Vec_$swap_i swaps the *length* elements of *vec1* selected by *inc1* with the *length* elements of *vec2* selected by *inc2*.

Through appropriate choice of *inc1* and *inc2*, a program can use **vec_$swap_i** to swap vectors between two matrixes. To select the *M*th vector in a matrix, choose *inc1* equal to the number of vectors in the matrix, and place the *M*th element of the matrix array at the beginning of *vec1*. To swap the selected vector with the *N*th vector of the same or another matrix, choose *inc2* equal to the number of vectors in the same or other matrix, and place the *N*th element of the matrix array at the beginning of *vec2*.

In C, the resulting operation is

```
j = 0;
k = 0;
for (i = 0; i < length, ++i) {
      temp = vec1[i];
      vec1[i] = vec2[i];
      vec2[i] = temp;
      j += inc1;
      k += inc2;
}
```

In Pascal, the resulting operation is

```
j := 1;
k := 1;
for i := 1 to length do
      begin
      temp := vec1[i];
      vec1[i] := vec2[i];
      vec2[i] := temp;
      j := j + inc1;
      k := k + inc2;
      end
```

In FORTRAN, the resulting operation is

```
      j = 1
      k = 1
      do 10 i = 1, length
           temp = vec1(j)
           vec1(j) = vec2(k)
           vec2(k) = temp
           j = j + inc1
           k = k + inc2
  10  continue
```

vec1 The vector to be swapped with *vec2*.

inc1 The increment for the index of *vec1* that selects the elements to be swapped with *vec2*.

vec2 The vector to be swapped with *vec1*.

inc2 The increment for the index of *vec2* that selects the elements to be swapped with *vec1*.

length The number of elements to swap.

NOTES

In C and Pascal, vec_$swap_i swaps column vectors; whereas in FORTRAN, it swaps row vectors.

SEE ALSO

vec_$dswap_i, vec_$iswap16_i, vec_$iswap_i, vec_$swap.

NAME

vec_$zero – zero a single-precision vector

SYNOPSIS (C)

```
#include <apollo/base.h>
#include <apollo/vec.h>

void vec_$zero(
        float *vector,
        long int &length)
```

SYNOPSIS (Pascal)

```
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/vec.ins.pas';

procedure vec_$zero(
        var vector: univ vec_$real_vector;
        in length: integer32);
```

SYNOPSIS (FORTRAN)

```
%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/vec.ins.ftn'

        parameter (nvec = 10)

        real vector(nvec)
        integer*4 length

        call vec_$zero(vector, length)
```

DESCRIPTION

Vec_$zero zeros the first *length* elements of the single-precision vector *vector*.

In C, the resulting operation is

```
for (i = 0; i < length; ++i)
      vector[i] = 0.0;
```

In Pascal, the resulting operation is

```
for i := 1 to length do
      vector[i] := 0.0;
```

In FORTRAN, the resulting operation is

```
      do 10 i = 1, length
           vec(i) = 0.0
  10  continue
```

vector The vector to be zeroed.

length The number of elements in *vector* to zero.

NOTES

In C and Pascal, **vec_$zero** zeros row vectors; whereas in FORTRAN, it zeros column vectors.

SEE ALSO

vec_$dzero, vec_$izero, vec_$izero16, vec_$zero_i.

NAME

vec_$zero_i – zero a vector in a single-precision matrix

SYNOPSIS (C)

```
#include <apollo/base.h>
#include <apollo/vec.h>

void vec_$zero_i(
        float *vector,
        long int &inc,
        long int &length)
```

SYNOPSIS (Pascal)

```
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/vec.ins.pas';

procedure vec_$zero_i(
        var vector: univ vec_$real_vector;
        in inc: integer32;
        in length: integer32);
```

SYNOPSIS (FORTRAN)

```
%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/vec.ins.ftn'

        parameter (nvec = 10)

        real vector(nvec)
        integer*4 length, inc

        call vec_$zero_i(vector, inc, length)
```

DESCRIPTION

Vec_$zero_i zeros the *length* elements of the single-precision array *vector* selected by *inc*.

Through appropriate choice of *inc*, a program can use vec_$zero_i to zero a vector within a matrix. To search the *M*th vector in a matrix, choose *inc* equal to the number of vectors in the matrix, and place the *M*th element of the matrix array at the beginning of *vector*.

In C, the resulting operation is

```
j = 0;
for (i = 0; i < length; ++i) {
      vector[i] = 0.0;
      j += inc;
}
```

In Pascal, the resulting operation is

```
j := 1;
for i := 1 to length do
      begin
      vector[i] := 0.0;
      j := j + inc;
      end
```

In FORTRAN, the resulting operation is

```
      j = 1
      do 10 i = 1, length
           vec(j) = 0.0
           j = j + inc
 10   continue
```

vector The vector to be zeroed.

inc An increment for the index of *vector* that chooses the elements to be zeroed.

length The number of elements in *vector* to zero.

NOTES

In C and Pascal, **vec_$zero_i** zeros column vectors; whereas in FORTRAN, it zeros row vectors.

SEE ALSO

vec_$dzero_i, vec_$izero16_i, vec_$izero_i, vec_$zero.

vfmt

Contents

NAME

intro – variable formatting package

SYNOPSIS (C)

#include <apollo/base.h>
#include <apollo/vfmt.h>

SYNOPSIS (Pascal)

%include '/sys/ins/base.ins.pas';
%include '/sys/ins/vfmt.ins.pas';

SYNOPSIS (FORTRAN)

%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/vfmt.ins.ftn'

DESCRIPTION

The **vfmt_$** calls facilitate formatted I/O, and arithmetic-to-ASCII conversions. They are most useful in Pascal or FORTRAN programs because their functionality largely overlaps that of C's **stdio** package.

Control Strings

All **vfmt_$** calls take a control string argument. The control string defines fields in an ASCII string that correspond to source or destination variables also provided in the call. Each **vfmt_$** call either writes data from source variables into fields of an ASCII string defined by the control string, or reads ASCII data fields defined by the control string into destination variables. The control string defines the types and the sizes of the data fields in the ASCII string, which must be compatible with the types of the source or destination variables. Control strings are made up of "format directives" that begin with a percent sign (%) and end with some terminating character defining their function. The **vfmt_$** calls ignore case in their control strings.

Data Directives

Each data field is defined by a single format directive in the control string. The **vfmt_$** calls recognize six different data formats: ASCII (**%A**), decimal (**%D**), scientific (**%E**), floating (**%F**), hexadecimal (**%H**), and octal (**%O**). Each basic data directive can take options between the percent sign and the terminating format character that modify the field definition. The following table summarizes the options appropriate for each data directive.

Data-Related Format Directives	
[Options] Directive	Function
% [fw][M length][E][Z][K][U|L] A	encode/decode ASCII
% [fw][E][Z][J][U|S|P][W|L] D	encode/decode integer decimal
% [fw|fw.dr][Z][J][S|P][W|L] E	encode scientific floating
% [fw|fw.dr][E][Z][J][S|P][W|L] F	encode/decode floating point
% [fw][E][Z][J][U|S|P][W|L][Y] H	encode/decode integer hexadecimal
% [fw][E][Z][J][U|S|P][W|L] O	encode/decode integer octal

ASCII Data (%A)

The %A directive is used to define ASCII fields, and can be used in control strings with both read and write calls.

The options have the following effects on ASCII data:

fw — *Fw* is an integer between 1 and 65536 inclusive, indicating the width of the field. A field width option means read or write exactly this many characters, padding with blanks if necessary, unless overridden by the Z, E, or M options. If no field width is specified, a **vfmt_$** call will read or write only non-blank characters.

M *length*

Length is an integer between 1 and 65536 inclusive, indicating the number of significant bytes in a source variable, or the number of bytes provided in a destination buffer. The M option can be used to specify the number of bytes read from a source variable or written to a destination variable. If the M option is not used and there is no *fw* specifier, a read call will look at the value passed in a destination variable allocated to receive a string length.

E — The E option is only valid in read calls, where it forces early termination of a data field when a delimiter is encountered. The default delimiters are blank and comma, but other delimiters can be declared with the %"..." directive.

K — The K option is only valid in read calls, where it means ignore leading spaces in the input field; that is, spaces that precede the first visible character. K overrides E.

Z — The Z option means include trailing spaces in the data field; that is, spaces that follow the last visible character. The Z option causes write calls to include trailing spaces from the source variable in the output. The Z option causes read calls to include trailing spaces from the input field when filling the destination variable. Without Z, trailing blanks are ignored.

L The **L** option means convert all characters read or written to lowercase. It is not valid with the U option.

U The **U** option means convert all characters read or written to uppercase. It is not valid with the L option.

Floating-Point Data (%E and %F)

The %E and %F directives are used to define floating-point fields. The %E directive is only valid in write calls, but the %F directive can be used in any control string.

The options have the following effects on floating-point data:

fw *Fw* is an integer between 1 and 100 inclusive, indicating the total width of the data field including integer and fractional parts of the mantissa, the sign, decimal point, and exponent (if any). In a write call, if the field width specified is too small to contain the value to be written, the field will be filled with asterisks (*) to indicate an overflow condition. If no field width is specified, a write call uses a field just large enough to hold the value, and a read call stops at the first field delimiter it encounters.

dw *Dw* is an integer specifying the width of the fractional part of the mantissa; that is, the number of digits to the right of the decimal point. It is only valid in write calls, and the default is 2.

E The E option is only valid in read calls, where it forces early termination of a data field when a delimiter is encountered. The default delimiters are blank and comma, but other delimiters can be declared with the %"..." directive. The E option overrides the field width specifier.

Z The Z option is only valid in write calls, where it means add zeros (0) to the left of the number to fill the output data field. Z is not valid without a field width specifier (*fw*).

J The **J** option is only valid in write calls, where it means left-justify the number in the output data field. Without the **J** option, numbers are right justified in the field.

S The S option means the number to be read or written has a minus sign if it is negative, and no sign if it is positive. This is the default unless the P option is specified. S is always redundant for read calls because ASCII data must be signed if negative.

P The P option is only valid in write calls, and means prepend a minus sign to the number if it is negative, and prepend a plus sign if it is positive.

W The W option means the number to be read or written is a single-precision floating-point value. If W is not specified, L is the default.

L The L option means the number to be read or written is a double-precision floating-point value. It is not a valid option if W is specified.

Integer Data (%D, %H, and %O)

The **%D**, **%H**, and **%O** directives are used to define integer fields. They can be used in both read and write calls.

fw — *Fw* is an integer between 1 and 65536 inclusive, indicating the minimum width of the data field. If the specified field width is too small to hold the number, the field is widened to accommodate the number. If no field width is specified, a write call uses a field just large enough to hold the value, and a read call stops at the first field delimiter it encounters.

E — The **E** option is only valid in read calls, where it forces early termination of a data field when a delimiter is encountered. The default delimiters are blank and comma, but other delimiters can be declared with the %"..." directive. The **E** option overrides the field width specifier.

Z — The **Z** option is only valid in write calls, where it means add zeros (0) to the left of the number to fill the output data field. **Z** is not valid without a field width specifier (*fw*).

J — The **J** option is only valid in write calls, where it means left-justify the number in the output data field. Without the **J** option, numbers are right justified in the field.

U — The **U** option means the number to be read or written is unsigned, and the call should ignore any evidence to the contrary. A write call will write an unsigned (and possibly very large) positive integer into the field even if the source variable appears negative. A read call will ignore plus and minus signs in the data field. The **U** option is most often used when writing via the **%H** and **%O** directives.

S — The **S** option means the number to be read or written has a minus sign if it is negative, and no sign if it is positive. This is the default unless the **P** option is specified. **S** is always redundant for read calls because ASCII data must be signed if negative.

P — The **P** option is only valid in write calls, and means prepend a minus sign to the number if it is negative, and prepend a plus sign if it is positive.

W — The **W** option means the number to be read or written is a 2-byte integer value. If **W** is not specified, **L** is the default.

L — The **L** option means the number to be read or written is a 4-byte integer value. It is not a valid option if **W** is specified.

Y — The **Y** option means write hexadecimal integers with lowercase letters instead of uppercase. It is only valid with hexadecimal output fields in write call control strings.

Control String Directives

The control string directives modify a call's interpretation of the control string passed to it. The following table summarizes the control string directives:

Control String-Related Format Directives	
Directive	**Function**
%"..."	declare characters to be used as field delimiters
%$	end control string
%.	end control string, inserting new line character
%n(	repeat string n times start
%)	repeat string n times stop

%"..." The %"..." directive defines the field delimiters that read calls use when the E option is specified in a data directive. When the %"..." directive appears in a control string, the characters between the double quotes, following the percent sign, become the field delimiters for that string. If no %"..." directive appears, then the read call uses the default delimiters, comma (,) and space ( ), equivalent to %", ". For example, %", "";" adds a double quote (") and semicolon (;) to the default delimiters.

%$ The %$ directive marks the end of a control string. It has no other effect.

%. The %. directive marks the end of a control string too. When used in a write call, it also means add a record terminator to the output. The record terminator is usually an ASCII newline character. When used in a read call, "%." is identical to "%$".

%*n*(and %)

Together, the %*n*(and %) directives delimit a portion of a control string to be repeated *n* times. *N* is required, and must be an integer between 1 and 65536 inclusive. For example, the fragment "%5(... %)" is identical to writing the delimited portion, denoted here as "..." five times in succession in the control string that contains it. Repeat directives cannot be nested.

Format Directives

The format directives modify the format of input or output between data fields. The following table summarizes the format directives:

Format-Related Format Directives	
Directive	**Function**
%%	write a single %
%/	write a record terminator
%nT	tab to column *n*
%nX	skip *n* characters
%n@	move argument pointer to argument *n*

%% When a write call encounters a "%%" in a control string, it writes a single percent sign (%) to the output.

%/ When a write call encounters a "%/" in a control string, it writes a record terminator to the output. The record terminator is usually an ASCII newline character.

*%n*T The *%n*T directive causes a read or write call to tab to column *n* before interpreting the control string further. *N* is optional; if it is omitted (%T), the call uses the next variable in the calling sequence for the tab value. Write calls use space as the tab fill character.

*%n*X The *%n*X directive causes a read or write call to skip *n* spaces before interpreting the control string further. The directive causes a write call to write *n* spaces to the output, and causes a read call to skip *n* characters in the input. *N* is optional; if it is omitted (%X), the call uses the next variable in the calling sequence for the number of spaces.

%n@ The *%n@* directive causes a write call to move it's argument pointer to the *n*th source argument. It allows programs interpret source arguments in an order specified by the control string, but does not generate any output. For example, "%1@" resets the argument pointer. *N* is required.

Data Types

vfmt_$string_t

A 200-byte array for passing control strings to **vfmt_$** calls.

Errors

vfmt_$unterminated_ctl_string

Unterminated control string.

vfmt_$invalid_ctl_string

Invalid control string.

vfmt_$too_few_args

Too few arguments supplied for a read or decode call.

vfmt_$fw_required

Field width missing on "(" designator.

vfmt_$eos
Encountered end of string where more text was expected.

vfmt_$null_token
Encountered null token where numeric token was expected.

vfmt_$nonnumeric_char
Non-numeric character found where numeric was expected.

vfmt_$sign_not_allowed
Sign encountered in unsigned field.

vfmt_$value_too_large
Value out of range in text string.

vfmt_$nonmatching_char
Character in text string does not match control string.

vfmt_$nonmatching_delimiter
Terminator in text string does not match specified terminator.

NAME

vfmt_$decode10 – formatted read from a string

SYNOPSIS (C)

```
#include <apollo/base.h>
#include <apollo/vfmt.h>

int vfmt_$decode10(
        char *control_string,
        char *source_string,
        int &source_length,
        int *decode_count,
        status_$t *status,
        void *a1, *a2, *a3, *a4, *a5, *a6, *a7, *a8, *a9, *a10)
```

SYNOPSIS (Pascal)

```
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/vfmt.ins.pas';

function vfmt_$decode10(
        in control_string: univ vfmt_$string_t;
        in source_string: univ vfmt_$string_t;
        in source_length: integer;
        out decode_count: integer;
        out status: status_$t;
        var a1, a2, a3, a4, a5, a6, a7, a8, a9, a10:
                univ vfmt_$generic_unsigned_arg): integer;
```

SYNOPSIS (FORTRAN)

```
%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/vfmt.ins.ftn'

        integer*2 nchar
        parameter (nchar = 128)

        integer*4 status
        integer*2 return_value, source_length, decode_count
        character control_string*(nchar), source_string*200
        integer*4 a1, a2, a3, a4, a5, a6, a7, a8, a9, a10

        last = vfmt_$decode10(control_string, source_string, source_length,
       &                      decode_count, status,
       &                      a1, a2, a3, a4, a5,
       &                      a6, a7, a8, a9, a10)
```

DESCRIPTION

Vfmt_$decode10 decodes an ASCII *source_string*, formatted into fields as defined by *control_string*, into 10 destination variables, and returns the number of bytes from *source_string* decoded into the destination variables.

control_string
A VFMT format string defining the fields in *source_string* to be decoded.

source_string
An ASCII string to decode.

source_length
The number of bytes in *source_string*.

decode_count
The number of fields in *source_string* decoded into the destination variables.

status The completion status.

a1, a2, a3, a4, a5, a6, a7, a8, a9, a10
Destination variables to receive the decoded data.

NOTES

A string field in *control_string* requires two destination variables: a character array to receive the value of the string, and a 2-byte integer variable to receive the number of bytes in the decoded string.

The length of a string field can be defined in *control_string* with the M option or, if the M option is not present, by passing a maximum string length in the destination variable intended to receive the string length. Regardless of whether there is an M option associated with the string field, the value returned in the destination variable is the minimum of the actual length of the string supplied by **vfmt_$decode10** and the maximum string length specified in the call.

SEE ALSO

vfmt_$decode2, vfmt_$decode5.

NAME

vfmt_$decode2 – formatted read from a string

SYNOPSIS (C)

```
#include &lt;apollo/base.h&gt;
#include &lt;apollo/vfmt.h&gt;

int vfmt_$decode2(
        char *control_string,
        char *source_string,
        int &source_length,
        int *decode_count,
        status_$t *status,
        void *a1, *a2)
```

SYNOPSIS (Pascal)

```
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/vfmt.ins.pas';

function vfmt_$decode2(
        in control_string: univ vfmt_$string_t;
        in source_string: univ vfmt_$string_t;
        in source_length: integer;
        out decode_count: integer;
        out status: status_$t;
        var a1, a2: univ vfmt_$generic_unsigned_arg): integer;
```

SYNOPSIS (FORTRAN)

```
%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/vfmt.ins.ftn'

      integer*2 nchar
      parameter (nchar = 128)

      integer*4 status
      integer*2 return_value, source_length, decode_count
      character control_string*(nchar), source_string*200
      integer*4 a1, a2

      last = vfmt_$decode2(control_string, source_string, source_length,
     &                     decode_count, status, a1, a2)
```

DESCRIPTION

Vfmt_$decode2 decodes an ASCII *source_string*, formatted into fields as defined by *control_string*, into two destination variables, and returns the number of bytes from *source_string* decoded into the destination variables.

control_string
A VFMT format string defining the fields in *source_string* to be decoded.

source_string
An ASCII string to decode.

source_length
The number of bytes in *source_string*.

decode_count
The number of fields in *source_string* decoded into the destination variables.

status The completion status.

a1, a2 Destination variables to receive the decoded data.

NOTES

A string field in *control_string* requires two destination variables: a character array to receive the value of the string, and a 2-byte integer variable to receive the number of bytes in the decoded string.

The length of a string field can be defined in *control_string* with the M option or, if the M option is not present, by passing a maximum string length in the destination variable intended to receive the string length. Regardless of whether there is an M option associated with the string field, the value returned in the destination variable is the minimum of the actual length of the string supplied by **vfmt_$decode2** and the maximum string length specified in the call.

SEE ALSO

vfmt_$decode10, vfmt_$decode5.

NAME

vfmt_$decode5 – formatted read from a string

SYNOPSIS (C)

```
#include <apollo/base.h>
#include <apollo/vfmt.h>

int vfmt_$decode(
        char *control_string,
        char *source_string,
        int &source_length,
        int *decode_count,
        status_$t *status,
        void *a1, *a2, *a3, *a4, *a5)
```

SYNOPSIS (Pascal)

```
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/vfmt.ins.pas';

function vfmt_$decode5(
        in control_string: univ vfmt_$string_t;
        in source_string: univ vfmt_$string_t;
        in source_length: integer;
        out decode_count: integer;
        out status: status_$t;
        var a1, a2, a3, a4, a5:
                univ vfmt_$generic_unsigned_arg): integer;
```

SYNOPSIS (FORTRAN)

```
%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/vfmt.ins.ftn'

        integer*2 nchar
        parameter (nchar = 128)

        integer*4 status
        integer*2 return_value, source_length, decode_count
        character control_string*(nchar), source_string*200
        integer*4 a1, a2, a3, a4, a5

        last = vfmt_$decode5(control_string, source_string, source_length,
       &                     decode_count, status,
       &                     a1, a2, a3, a4, a5)
```

DESCRIPTION

Vfmt_$decode5 decodes an ASCII *source_string*, formatted into fields as defined by *control_string*, into five destination variables, and returns the number of bytes from *source_string* decoded into the destination variables.

control_string
: A VFMT format string defining the fields in *source_string* to be decoded.

source_string
: An ASCII string to decode.

source_length
: The number of bytes in *source_string*.

decode_count
: The number of fields in *source_string* decoded into the destination variables.

status The completion status.

a1, a2, a3, a4, a5
: Destination variables to receive the decoded data.

NOTES

A string field in *control_string* requires two destination variables: a character array to receive the value of the string, and a 2-byte integer variable to receive the number of bytes in the decoded string.

The length of a string field can be defined in *control_string* with the **M** option or, if the **M** option is not present, by passing a maximum string length in the destination variable intended to receive the string length. Regardless of whether there is an **M** option associated with the string field, the value returned in the destination variable is the minimum of the actual length of the string supplied by **vfmt_$decode5** and the maximum string length specified in the call.

SEE ALSO

vfmt_$decode10, vfmt_$decode2.

NAME

vfmt_$encode – formatted write to a string

SYNOPSIS (C)

```
#include <apollo/base.h>
#include <apollo/vfmt.h>

void vfmt_$encode(
        char *control_string,
        char *string_buffer,
        int &buffer_size,
        int *string_length,
        ...)
```

SYNOPSIS (Pascal)

```
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/vfmt.ins.pas';

procedure vfmt_$encode(
        in control_string: univ vfmt_$string_t ;
        out string_buffer: univ vfmt_$string_t;
        in buffer_size: integer;
        out string_length: integer;
        in a1, a2, a3, a4, a5,
                a6, a7, a8, a9, a10,
                a11, a12, a13, a14, a15,
                a16, a17, a18, a19, a20: univ vfmt_$generic_signed_arg);
```

SYNOPSIS (FORTRAN)

```
%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/vfmt.ins.ftn'

        integer*2 nchar
        parameter (nchar = 128)

        integer*2 buffer_size, string_length
        character control_string*(nchar), string_buffer*200
        integer*4 a1, a2, a3, a4, a5, a6, a7, a8, a9, a10, a11
        integer*4 a12, a13, a14, a15, a16, a17, a18, a19, a20

        call vfmt_$encode(control_string, string_buffer, buffer_size, string_length,
       &                  a1, a2, a3, a4, a5, a6, a7, a8, a9, a10, a11,
       &                  a12, a13, a14, a15, a16, a17, a18, a19, a20)
```

DESCRIPTION

Vfmt_$encode encodes up to twenty source variables into an ASCII string whose format is defined by *control_string*, and writes the result into *string_buffer*.

control_string
A VFMT format string defining how to format the source variables.

string_buffer
A buffer allocated to receive the formatted string supplied by **vfmt_$encode**.

buffer_size
The number of bytes allocated at *string_buffer* to receive the formatted string. **Vfmt_$encode** will not write more than *buffer_size* bytes into *string_buffer*.

string_length
The number of bytes that **vfmt_$encode** wrote into *string_buffer*.

a1, a2, a3, ... a19, a20
Up to 20 source variables containing data to encode into ASCII.

NOTES

A string field in *control_string* requires two source variables: a character array containing the value of the string, and a 2-byte integer variable specifying the number of bytes in the string.

SEE ALSO

vfmt_$encode10, vfmt_$encode2, vfmt_$encode5.

NAME

vfmt_$encode10 – formatted write to a string

SYNOPSIS (C)

```
#include <apollo/base.h>
#include <apollo/vfmt.h>

void vfmt_$encode10(
        char *control_string,
        char *string_buffer,
        int &buffer_size,
        int *string_length,
        ...)
```

SYNOPSIS (Pascal)

```
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/vfmt.ins.pas';

procedure vfmt_$encode10(
        in control_string: univ vfmt_$string_t ;
        out string_buffer: univ vfmt_$string_t;
        in buffer_size: integer;
        out string_length: integer;
        in a1, a2, a3, a4, a5,
                a6, a7, a8, a9, a10: univ vfmt_$generic_signed_arg);
```

SYNOPSIS (FORTRAN)

```
%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/vfmt.ins.ftn'

      integer*2 nchar
      parameter (nchar = 128)

      integer*2 buffer_size, string_length
      character control_string*(nchar), string_buffer*200
      integer*4 a1, a2, a3, a4, a5, a6, a7, a8, a9, a10

      call vfmt_$encode10(control_string, string_buffer,
     &                    buffer_size, string_length,
     &                    a1, a2, a3, a4, a5, a6, a7, a8, a9, a10)
```

DESCRIPTION

Vfmt_$encode10 encodes up to ten source variables into an ASCII string whose format is defined by *control_string*, and writes the result into *string_buffer*.

control_string
A VFMT format string defining how to format the source variables.

string_buffer
A buffer allocated to receive the formatted string supplied by **vfmt_$encode10**.

buffer_size
The number of bytes allocated at *string_buffer* to receive the formatted string. **Vfmt_$encode10** will not write more than *buffer_size* bytes into *string_buffer*.

string_length
The number of bytes that **vfmt_$encode10** wrote into *string_buffer*.

a1, a2, a3, ... a9, a10
Up to 10 source variables containing data to encode into ASCII.

NOTES

A string field in *control_string* requires two source variables: a character array containing the value of the string, and a 2-byte integer variable specifying the number of bytes in the string.

SEE ALSO

vfmt_$encode, vfmt_$encode2, vfmt_$encode5.

NAME

vfmt_$encode2 – formatted write to a string

SYNOPSIS (C)

```
#include <apollo/base.h>
#include <apollo/vfmt.h>

void vfmt_$encode2(
        char *control_string,
        char *string_buffer,
        int &buffer_size,
        int *string_length,
        ...)
```

SYNOPSIS (Pascal)

```
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/vfmt.ins.pas';

procedure vfmt_$encode2(
        in control_string: univ vfmt_$string_t ;
        out string_buffer: univ vfmt_$string_t;
        in buffer_size: integer;
        out string_length: integer;
        in a1, a2: univ vfmt_$generic_signed_arg);
```

SYNOPSIS (FORTRAN)

```
%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/vfmt.ins.ftn'

      integer*2 nchar
      parameter (nchar = 128)

      integer*2 buffer_size, string_length
      character control_string*(nchar), string_buffer*200
      integer*4 a1, a2

      call vfmt_$encode2(control_string, string_buffer,
     &                   buffer_size, string_length,
     &                   a1, a2)
```

DESCRIPTION

Vfmt_$encode2 encodes up to two source variables into an ASCII string whose format is defined by *control_string*, and writes the result into *string_buffer*.

control_string
    A VFMT format string defining how to format the source variables.

string_buffer
A buffer allocated to receive the formatted string supplied by **vfmt_$encode2**.

buffer_size
The number of bytes allocated at *string_buffer* to receive the formatted string. **Vfmt_$encode2** will not write more than *buffer_size* bytes into *string_buffer*.

string_length
The number of bytes that **vfmt_$encode2** wrote into *string_buffer*.

a1, a2 Up to two source variables containing data to encode into ASCII.

NOTES

A string field in *control_string* requires two source variables: a character array containing the value of the string, and a 2-byte integer variable specifying the number of bytes in the string.

SEE ALSO

vfmt_$encode, vfmt_$encode10, vfmt_$encode5.

NAME

vfmt_$encode5 – formatted write to a string

SYNOPSIS (C)

```
#include <apollo/base.h>
#include <apollo/vfmt.h>

void vfmt_$encode5(
        char *control_string,
        char *string_buffer,
        int &buffer_size,
        int *string_length,
        ...)
```

SYNOPSIS (Pascal)

```
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/vfmt.ins.pas';

procedure vfmt_$encode5(
        in control_string: univ vfmt_$string_t ;
        out string_buffer: univ vfmt_$string_t;
        in buffer_size: integer;
        out string_length: integer;
        in a1, a2, a3, a4, a5: univ vfmt_$generic_signed_arg);
```

SYNOPSIS (FORTRAN)

```
%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/vfmt.ins.ftn'

        integer*2 nchar
        parameter (nchar = 128)

        integer*2 buffer_size, string_length
        character control_string*(nchar), string_buffer*200
        integer*4 a1, a2, a3, a4, a5

        call vfmt_$encode5(control_string, string_buffer,
       &                   buffer_size, string_length,
       &                   a1, a2, a3, a4, a5)
```

DESCRIPTION

Vfmt_$encode5 encodes up to five source variables into an ASCII string whose format is defined by *control_string*, and writes the result into *string_buffer*.

control_string
: A VFMT format string defining how to format the source variables.

string_buffer
A buffer allocated to receive the formatted string supplied by **vfmt_$encode5**.

buffer_size
The number of bytes allocated at *string_buffer* to receive the formatted string. **Vfmt_$encode5** will not write more than *buffer_size* bytes into *string_buffer*.

string_length
The number of bytes that **vfmt_$encode5** wrote into *string_buffer*.

a1, a2, a3, a4, a5
Up to five source variables containing data to encode into ASCII.

NOTES

A string field in *control_string* requires two source variables: a character array containing the value of the string, and a 2-byte integer variable specifying the number of bytes in the string.

SEE ALSO

vfmt_$encode, vfmt_$encode10, vfmt_$encode2.

NAME

vfmt_$read10 – formatted read from standard input

SYNOPSIS (C)

```
#include <apollo/base.h>
#include <apollo/vfmt.h>

void vfmt_$read10(
        char *control_string,
        int *field_count,
        status_$t *status,
        void *a1, *a2, *a3, *a4, *a5, *a6, *a7, *a8, *a9, *a10)
```

SYNOPSIS (Pascal)

```
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/vfmt.ins.pas';

procedure vfmt_$read10(
        in control_string: univ vfmt_$string_t;
        out field_count: integer;
        out status: status_$t;
        var a1, a2, a3, a4, a5, a6, a7, a8, a9, a10:
                univ vfmt_$generic_unsigned_arg);
```

SYNOPSIS (FORTRAN)

```
%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/vfmt.ins.ftn'

        integer*2 nchar
        parameter (nchar = 128)

        integer*4 status
        integer*2 field_count
        character control_string*(nchar)
        integer*4 a1, a2, a3, a4, a5, a6, a7, a8, a9, a10

        call vfmt_$read10(control_string, field_count, status,
       &                  a1, a2, a3, a4, a5,
       &                  a6, a7, a8, a9, a10)
```

DESCRIPTION

Vfmt_$read10 reads an ASCII line from standard input, formatted in fields as defined by *control_string*, into 10 destination variables.

control_string

A VFMT format string defining the input fields to be decoded.

field_count
The number of input fields decoded into the destination variables.

status The completion status.

a1, a2, a3, a4, a5, a6, a7, a8, a9, a10
Destination variables to receive the decoded input.

NOTES

A string field in *control_string* requires two destination variables: a character array to receive the value of the string, and a 2-byte integer variable to receive the number of bytes in the decoded string.

The length of a string field can be defined in *control_string* with the **M** option or, if the **M** option is not present, by passing a maximum string length in the destination variable intended to receive the string length. Regardless of whether there is an **M** option associated with the string field, the value returned in the destination variable is the minimum of the actual length of the string supplied by **vfmt_$read10** and the maximum string length specified in the call.

SEE ALSO

vfmt_$read2, vfmt_$read5.

NAME

vfmt_$read2 – formatted read from standard input

SYNOPSIS (C)

```
#include <apollo/base.h>
#include <apollo/vfmt.h>

void vfmt_$read2(
        char *control_string,
        int *field_count,
        status_$t *status,
        void *a1, *a2)
```

SYNOPSIS (Pascal)

```
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/vfmt.ins.pas';

procedure vfmt_$read2(
        in control_string: univ vfmt_$string_t;
        out field_count: integer;
        out status: status_$t;
        var a1, a2: univ vfmt_$generic_unsigned_arg);
```

SYNOPSIS (FORTRAN)

```
%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/vfmt.ins.ftn'

      integer*2 nchar
      parameter (nchar = 128)

      integer*4 status
      integer*2 field_count
      character control_string*(nchar)
      integer*4 a1, a2

      call vfmt_$read2(control_string, field_count, status, a1, a2)
```

DESCRIPTION

Vfmt_$read2 reads an ASCII line from standard input, formatted in fields as defined by *control_string*, into two destination variables.

control_string
: A VFMT format string defining the input fields to be decoded.

field_count
: The number of input fields decoded into the destination variables.

status
: The completion status.

a1, a2 Destination variables to receive the decoded input.

NOTES

A string field in *control_string* requires two destination variables: a character array to receive the value of the string, and a 2-byte integer variable to receive the number of bytes in the decoded string.

The length of a string field can be defined in *control_string* with the **M** option or, if the **M** option is not present, by passing a maximum string length in the destination variable intended to receive the string length. Regardless of whether there is an **M** option associated with the string field, the value returned in the destination variable is the minimum of the actual length of the string supplied by **vfmt_$read2** and the maximum string length specified in the call.

SEE ALSO

vfmt_$read10, vfmt_$read5.

NAME

vfmt_$read5 – formatted read from standard input

SYNOPSIS (C)

```
#include <apollo/base.h>
#include <apollo/vfmt.h>

void vfmt_$read5(
        char *control_string,
        int *field_count,
        status_$t *status,
        void *a1, *a2, *a3, *a4, *a5)
```

SYNOPSIS (Pascal)

```
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/vfmt.ins.pas';

procedure vfmt_$read5(
        in control_string: univ vfmt_$string_t;
        out field_count: integer;
        out status: status_$t;
        var a1, a2, a3, a4, a5: univ vfmt_$generic_unsigned_arg);
```

SYNOPSIS (FORTRAN)

```
%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/vfmt.ins.ftn'

      integer*2 nchar
      parameter (nchar = 128)

      integer*4 status
      integer*2 field_count
      character control_string*(nchar)
      integer*4 a1, a2, a3, a4, a5

      call vfmt_$read5(control_string, field_count, status,
     &                 a1, a2, a3, a4, a5)
```

DESCRIPTION

Vfmt_$read5 reads an ASCII line from standard input, formatted in fields as defined by *control_string*, into five destination variables.

control_string
: A VFMT format string defining the input fields to be decoded.

field_count
: The number of input fields decoded into the destination variables.

status The completion status.

a1, a2, a3, a4, a5
Destination variables to receive the decoded input.

NOTES

A string field in *control_string* requires two destination variables: a character array to receive the value of the string, and a 2-byte integer variable to receive the number of bytes in the decoded string.

The length of a string field can be defined in *control_string* with the **M** option or, if the **M** option is not present, by passing a maximum string length in the destination variable intended to receive the string length. Regardless of whether there is an **M** option associated with the string field, the value returned in the destination variable is the minimum of the actual length of the string supplied by **vfmt_$read5** and the maximum string length specified in the call.

SEE ALSO

vfmt_$read10, vfmt_$read2.

NAME

vfmt_$rs10 – formatted read from a stream

SYNOPSIS (C)

```
#include <apollo/base.h>
#include <apollo/vfmt.h>

void vfmt_$rs10(
        int &stream_id,
        char *control_string,
        int *field_count,
        status_$t *status,
        void *a1, *a2, *a3, *a4, *a5, *a6, *a7, *a8, *a9, *a10)
```

SYNOPSIS (Pascal)

```
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/vfmt.ins.pas';

procedure vfmt_$rs10(
        in stream_id: univ integer;
        in control_string: univ vfmt_$string_t;
        out field_count: integer;
        out status: status_$t;
        var a1, a2, a3, a4, a5,
                a6, a7, a8, a9, a10: univ vfmt_$generic_unsigned_arg);
```

SYNOPSIS (FORTRAN)

```
%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/vfmt.ins.ftn'

      integer*2 nchar
      parameter (nchar = 128)

      integer*4 status
      integer*2 stream_id, field_count
      character control_string*(nchar)
      integer*4 a1, a2, a3, a4, a5, a6, a7, a8, a9, a10

      call vfmt_$rs10(stream_id, control_string, field_count, status,
     &                a1, a2, a3, a4, a5, a6, a7, a8, a9, a10)
```

DESCRIPTION

Vfmt_$rs10 reads an ASCII record from the stream specified by *stream_id*, formatted in fields as defined by *control_string*, into 10 destination variables.

stream_id

The stream ID of the stream to read from.

control_string
A VFMT format string defining the input fields to be decoded.

field_count
The number of input fields decoded into the destination variables.

status The completion status.

a1, a2, a3, a4, a5, a6, a7, a8, a9, a10
Destination variables to receive the decoded input.

NOTES

A string field in *control_string* requires two destination variables: a character array to receive the value of the string, and a 2-byte integer variable to receive the number of bytes in the decoded string.

The length of a string field can be defined in *control_string* with the **M** option or, if the **M** option is not present, by passing a maximum string length in the destination variable intended to receive the string length. Regardless of whether there is an **M** option associated with the string field, the value returned in the destination variable is the minimum of the actual length of the string supplied by **vfmt_$rs10** and the maximum string length specified in the call.

SEE ALSO

vfmt_$rs2, vfmt_$rs5.

NAME

vfmt_$rs2 – formatted read from a stream

SYNOPSIS (C)

```
#include <apollo/base.h>
#include <apollo/vfmt.h>

void vfmt_$rs2(
        int &stream_id,
        char *control_string,
        int *field_count,
        status_$t *status,
        void *a1, *a2)
```

SYNOPSIS (Pascal)

```
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/vfmt.ins.pas';

procedure vfmt_$rs2(
        in stream_id: univ integer;
        in control_string: univ vfmt_$string_t;
        out field_count: integer;
        out status: status_$t;
        var a1, a2: univ vfmt_$generic_unsigned_arg);
```

SYNOPSIS (FORTRAN)

```
%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/vfmt.ins.ftn'

      integer*2 nchar
      parameter (nchar = 128)

      integer*4 status
      integer*2 stream_id, field_count
      character control_string*(nchar)
      integer*4 a1, a2

      call vfmt_$rs2(stream_id, control_string, field_count, status,
     &               a1, a2)
```

DESCRIPTION

Vfmt_$rs2 reads an ASCII record from the stream specified by *stream_id*, formatted in fields as defined by *control_string*, into two destination variables.

stream_id

The stream ID of the stream to read from.

control_string
A VFMT format string defining the input fields to be decoded.

field_count
The number of input fields decoded into the destination variables.

status The completion status.

a1, a2 Destination variables to receive the decoded input.

NOTES

A string field in *control_string* requires two destination variables: a character array to receive the value of the string, and a 2-byte integer variable to receive the number of bytes in the decoded string.

The length of a string field can be defined in *control_string* with the **M** option or, if the **M** option is not present, by passing a maximum string length in the destination variable intended to receive the string length. Regardless of whether there is an **M** option associated with the string field, the value returned in the destination variable is the minimum of the actual length of the string supplied by **vfmt_$rs2** and the maximum string length specified in the call.

SEE ALSO

vfmt_$rs10, vfmt_$rs5.

NAME

vfmt_$rs5 – formatted read from a stream

SYNOPSIS (C)

```
#include <apollo/base.h>
#include <apollo/vfmt.h>

void vfmt_$rs5(
        int &stream_id,
        char *control_string,
        int *field_count,
        status_$t *status,
        void *a1, *a2, *a3, *a4, *a5)
```

SYNOPSIS (Pascal)

```
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/vfmt.ins.pas';

procedure vfmt_$rs5(
        in stream_id: univ integer;
        in control_string: univ vfmt_$string_t;
        out field_count: integer;
        out status: status_$t;
        var a1, a2, a3, a4, a5: univ vfmt_$generic_unsigned_arg);
```

SYNOPSIS (FORTRAN)

```
%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/vfmt.ins.ftn'

      integer*2 nchar
      parameter (nchar = 128)

      integer*4 status
      integer*2 stream_id, field_count
      character control_string*(nchar)
      integer*4 a1, a2, a3, a4, a5

      call vfmt_$rs5(stream_id, control_string, field_count, status,
     &               a1, a2, a3, a4, a5)
```

DESCRIPTION

Vfmt_$rs5 reads an ASCII record from the stream specified by *stream_id*, formatted in fields as defined by *control_string*, into five destination variables.

stream_id
: The stream ID of the stream to read from.

control_string
A VFMT format string defining the input fields to be decoded.

field_count
The number of input fields decoded into the destination variables.

status The completion status.

a1, a2, a3, a4, a5
Destination variables to receive the decoded input.

NOTES

A string field in *control_string* requires two destination variables: a character array to receive the value of the string, and a 2-byte integer variable to receive the number of bytes in the decoded string.

The length of a string field can be defined in *control_string* with the **M** option or, if the **M** option is not present, by passing a maximum string length in the destination variable intended to receive the string length. Regardless of whether there is an **M** option associated with the string field, the value returned in the destination variable is the minimum of the actual length of the string supplied by **vfmt_$rs5** and the maximum string length specified in the call.

SEE ALSO

vfmt_$rs10, vfmt_$rs2.

NAME

vfmt_$write – formatted write to standard output

SYNOPSIS (C)

```
#include <apollo/base.h>
#include <apollo/vfmt.h>

void vfmt_$write(
        char *control_string,
        ...)
```

SYNOPSIS (Pascal)

```
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/vfmt.ins.pas';

procedure vfmt_$write(
        in control_string: univ vfmt_$string_t;
        in a1, a2, a3, a4, a5, a6, a7, a8, a9, a10,
               a11, a12, a13, a14, a15, a16, a17, a18, a19, a20:
                      univ vfmt_$generic_signed_arg);
```

SYNOPSIS (FORTRAN)

```
%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/vfmt.ins.ftn'

      integer*2 nchar
      parameter (nchar = 128)

      character control_string*(nchar)
      integer*4 a1, a2, a3, a4, a5, a6, a7, a8, a9, a10, a11
      integer*4 a12, a13, a14, a15, a16, a17, a18, a19, a20

      call vfmt_$write10(control_string,
     &                   a1, a2, a3, a4, a5, a6, a7, a8, a9, a10, a11,
     &                   a12, a13, a14, a15, a16, a17, a18, a19, a20)
```

DESCRIPTION

Vfmt_$write encodes up to twenty source variables into an ASCII string whose format is defined by *control_string*, and writes the result to standard output.

control_string
: A VFMT format string defining how to format the source variables.

a1, a2, a3, ... a19, a20
: Up to 20 source variables containing data to encode into ASCII.

NOTES

A string field in *control_string* requires two source variables: a character array containing the value of the string, and a 2-byte integer variable specifying the number of bytes in the string.

SEE ALSO

vfmt_$write10, vfmt_$write2, vfmt_$write5.

NAME

vfmt_$write10 – formatted write to standard output

SYNOPSIS (C)

```
#include <apollo/base.h>
#include <apollo/vfmt.h>

void vfmt_$write10(
        char *control_string,
        ...)
```

SYNOPSIS (Pascal)

```
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/vfmt.ins.pas';

procedure vfmt_$write10(
        in control_string: univ vfmt_$string_t;
        in a1, a2, a3, a4, a5, a6, a7, a8, a9, a10:
                univ vfmt_$generic_signed_arg);
```

SYNOPSIS (FORTRAN)

```
%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/vfmt.ins.ftn'

        integer*2 nchar
        parameter (nchar = 128)

        character control_string*(nchar)
        integer*4 a1, a2, a3, a4, a5, a6, a7, a8, a9, a10

        call vfmt_$write10(control_string,
       &                   a1, a2, a3, a4, a5, a6, a7, a8, a9, a10)
```

DESCRIPTION

Vfmt_$write10 encodes up to ten source variables into an ASCII string whose format is defined by *control_string*, and writes the result to standard output.

control_string
: A VFMT format string defining how to format the source variables.

a1, a2, a3, ... a9, a10
: Up to 10 source variables containing data to encode into ASCII.

NOTES

A string field in *control_string* requires two source variables: a character array containing the value of the string, and a 2-byte integer variable specifying the number of bytes in the string.

SEE ALSO

vfmt_$write, vfmt_$write2, vfmt_$write5.

NAME

vfmt_$write2 – formatted write to standard output

SYNOPSIS (C)

```
#include <apollo/base.h>
#include <apollo/vfmt.h>

void vfmt_$write2(
        char *control_string,
        ...)
```

SYNOPSIS (Pascal)

```
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/vfmt.ins.pas';

procedure vfmt_$write2(
        in control_string: univ vfmt_$string_t;
        in a1, a2: univ vfmt_$generic_signed_arg);
```

SYNOPSIS (FORTRAN)

```
%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/vfmt.ins.ftn'

      integer*2 nchar
      parameter (nchar = 128)

      character control_string*(nchar)
      integer*4 a1, a2

      call vfmt_$write2(control_string, a1, a2)
```

DESCRIPTION

Vfmt_$write2 encodes up to two source variables into an ASCII string whose format is defined by *control_string*, and writes the result to standard output.

control_string
A VFMT format string defining how to format the source variables.

a1, a2 Up to two source variables containing data to encode into ASCII.

NOTES

A string field in *control_string* requires two source variables: a character array containing the value of the string, and a 2-byte integer variable specifying the number of bytes in the string.

SEE ALSO

vfmt_$write, vfmt_$write10, vfmt_$write5.

NAME

vfmt_$write5 – formatted write to standard output

SYNOPSIS (C)

```
#include &lt;apollo/base.h&gt;
#include &lt;apollo/vfmt.h&gt;

void vfmt_$write5(
        char *control_string,
        ...)
```

SYNOPSIS (Pascal)

```
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/vfmt.ins.pas';

procedure vfmt_$write5(
        in control_string: univ vfmt_$string_t;
        in a1, a2, a3, a4, a5: univ vfmt_$generic_signed_arg);
```

SYNOPSIS (FORTRAN)

```
%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/vfmt.ins.ftn'

        integer*2 nchar
        parameter (nchar = 128)

        character control_string*(nchar)
        integer*4 a1, a2, a3, a4, a5

        call vfmt_$write5(control_string, a1, a2, a3, a4, a5)
```

DESCRIPTION

Vfmt_$write5 encodes up to five source variables into an ASCII string whose format is defined by *control_string*, and writes the result to standard output.

control_string
: A VFMT format string defining how to format the source variables.

a1, a2, a3, a4, a5
: Up to five source variables containing data to encode into ASCII.

NOTES

A string field in *control_string* requires two source variables: a character array containing the value of the string, and a 2-byte integer variable specifying the number of bytes in the string.

SEE ALSO

vfmt_$write, vfmt_$write10, vfmt_$write2.

NAME

vfmt_$ws – formatted write to a stream

SYNOPSIS (C)

```
#include <apollo/base.h>
#include <apollo/vfmt.h>

void vfmt_$ws(
        ios_$id_t &stream_id,
        char *control_string,
        ...)
```

SYNOPSIS (Pascal)

```
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/vfmt.ins.pas';

procedure vfmt_$ws(
        in stream_id: ios_$id_t;
        in control_string: univ vfmt_$string_t;
        in a1, a2, a3, a4, a5, a6, a7, a8, a9, a10,
              a11, a12, a13, a14, a15, a16, a17, a18, a19, a20:
                    univ vfmt_$generic_signed_arg);
```

SYNOPSIS (FORTRAN)

```
%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/vfmt.ins.ftn'

      integer*2 nchar
      parameter (nchar = 128)

      integer*2 stream_id
      character control_string*(nchar)
      integer*4 a1, a2, a3, a4, a5, a6, a7, a8, a9, a10, a11
      integer*4 a12, a13, a14, a15, a16, a17, a18, a19, a20

      call vfmt_$ws(stream_id, control_string,
     &              a1, a2, a3, a4, a5, a6, a7, a8, a9, a10, a11,
     &              a12, a13, a14, a15, a16, a17, a18, a19, a20)
```

DESCRIPTION

Vfmt_$ws encodes up to twenty source variables into an ASCII string whose format is defined by *control_string*, and writes the result to the stream specified by *stream_id*.

stream_id
: The stream ID of the stream to write to.

control_string
: A VFMT format string defining how to format the source variables.

a1, a2, a3, ... a19, a20
Up to 20 source variables containing data to encode into ASCII.

NOTES

A string field in *control_string* requires two source variables: a character array containing the value of the string, and a 2-byte integer variable specifying the number of bytes in the string.

SEE ALSO

vfmt_$ws10, vfmt_$ws2, vfmt_$ws5.

NAME

vfmt_$ws10 – formatted write to a stream

SYNOPSIS (C)

```
#include <apollo/base.h>
#include <apollo/vfmt.h>

void vfmt_$ws10(
        ios_$id_t &stream_id,
        char *control_string,
        ...)
```

SYNOPSIS (Pascal)

```
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/vfmt.ins.pas';

procedure vfmt_$ws10(
        in stream_id: ios_$id_t;
        in control_string: univ vfmt_$string_t;
        in a1, a2, a3, a4, a5, a6, a7, a8, a9, a10:
                univ vfmt_$generic_signed_arg);
```

SYNOPSIS (FORTRAN)

```
%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/vfmt.ins.ftn'

        integer*2 nchar
        parameter (nchar = 128)

        integer*2 stream_id
        character control_string*(nchar)
        integer*4 a1, a2, a3, a4, a5, a6, a7, a8, a9, a10

        call vfmt_$ws10(stream_id, control_string,
       &                a1, a2, a3, a4, a5, a6, a7, a8, a9, a10)
```

DESCRIPTION

Vfmt_$ws10 encodes up to ten source variables into an ASCII string whose format is defined by *control_string*, and writes the result to the stream specified by *stream_id*.

stream_id

The stream ID of the stream to write to.

control_string

A VFMT format string defining how to format the source variables.

a1, a2, a3, ... a9, a10

Up to 10 source variables containing data to encode into ASCII.

NOTES

A string field in *control_string* requires two source variables: a character array containing the value of the string, and a 2-byte integer variable specifying the number of bytes in the string.

SEE ALSO

vfmt_$ws, vfmt_$ws2, vfmt_$ws5.

NAME

vfmt_$ws2 – formatted write to a stream

SYNOPSIS (C)

```
#include <apollo/base.h>
#include <apollo/vfmt.h>

void vfmt_$ws2(
        ios_$id_t &stream_id,
        char *control_string,
        ...)
```

SYNOPSIS (Pascal)

```
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/vfmt.ins.pas';

procedure vfmt_$ws2(
        in stream_id: ios_$id_t;
        in control_string: univ vfmt_$string_t;
        in a1, a2: univ vfmt_$generic_signed_arg);
```

SYNOPSIS (FORTRAN)

```
%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/vfmt.ins.ftn'

        integer*2 nchar
        parameter (nchar = 128)

        integer*2 stream_id
        character control_string*(nchar)
        integer*4 a1, a2

        call vfmt_$ws2(stream_id, control_string, a1, a2)
```

DESCRIPTION

Vfmt_$ws2 encodes up to two source variables into an ASCII string whose format is defined by *control_string*, and writes the result to the stream specified by *stream_id*.

stream_id
: The stream ID of the stream to write to.

control_string
: A VFMT format string defining how to format the source variables.

a1, a2
: Up to two source variables containing data to encode into ASCII.

NOTES

A string field in *control_string* requires two source variables: a character array containing the value of the string, and a 2-byte integer variable specifying the number of bytes in the string.

SEE ALSO

vfmt_$ws, vfmt_$ws10, vfmt_$ws5.

NAME

vfmt_$ws5 – formatted write to a stream

SYNOPSIS (C)

```
#include <apollo/base.h>
#include <apollo/vfmt.h>

void vfmt_$ws5(
        ios_$id_t &stream_id,
        char *control_string,
        ...)
```

SYNOPSIS (Pascal)

```
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/vfmt.ins.pas';

procedure vfmt_$ws5(
        in stream_id: ios_$id_t;
        in control_string: univ vfmt_$string_t;
        in a1, a2, a3, a4, a5: univ vfmt_$generic_signed_arg);
```

SYNOPSIS (FORTRAN)

```
%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/vfmt.ins.ftn'

      integer*2 nchar
      parameter (nchar = 128)

      integer*2 stream_id
      character control_string*(nchar)
      integer*4 a1, a2, a3, a4, a5

      call vfmt_$ws5(stream_id, control_string, a1, a2, a3, a4, a5)
```

DESCRIPTION

Vfmt_$ws5 encodes up to five source variables into an ASCII string whose format is defined by *control_string*, and writes the result to the stream specified by *stream_id*.

stream_id
: The stream ID of the stream to write to.

control_string
: A VFMT format string defining how to format the source variables.

a1, a2, a3, a4, a5
: Up to five source variables containing data to encode into ASCII.

NOTES

A string field in *control_string* requires two source variables: a character array containing the value of the string, and a 2-byte integer variable specifying the number of bytes in the string.

SEE ALSO

vfmt_$ws, vfmt_$ws10, vfmt_$ws2.

Reader's Response

Please take a few minutes to send us the information we need to revise and improve our manuals from your point of view.

Document Title: *Domain/OS Call Reference, Volume 2*
Order No.: 012888-A00
Date of Publication: June, 1988

What type of user are you?

_____ System programmer; language ____________________
_____ Applications programmer; language ____________________
_____ System maintenance person
_____ System Administrator
_____ Student Programmer
_____ Other
_____ Manager/Professional
_____ Technical Professional
_____ Novice

How often do you use the Domain system?____________________

What parts of the manual are especially useful for the job you are doing?____________________

What additional information would you like the manual to include?____________________

Please list any errors, omissions, or problem areas in the manual. (Identify errors by page, section, figure, or table number wherever possible. Specify additional index entries.)____________________

Your Name ____________________ Date

Organization

Street Address

City ____________________ State ____________________ Zip

No postage necessary if mailed in the U.S.

cut or fold along dotted line

fold

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 78 CHELMSFORD, MA 01824

POSTAGE WILL BE PAID BY ADDRESSEE

APOLLO COMPUTER INC.
Technical Publications
P.O. Box 451
Chelmsford, MA 01824

fold

Reader's Response

Please take a few minutes to send us the information we need to revise and improve our manuals from your point of view.

Document Title: *Domain/OS Call Reference, Volume 2*
Order No.: 012888-A00
Date of Publication: June, 1988

What type of user are you?

_____ System programmer; language ____________________

_____ Applications programmer; language ____________________

_____ System maintenance person

_____ System Administrator

_____ Student Programmer

_____ Other

_____ Manager/Professional

_____ Technical Professional

_____ Novice

How often do you use the Domain system? ____________________

What parts of the manual are especially useful for the job you are doing? ____________________

What additional information would you like the manual to include? ____________________

Please list any errors, omissions, or problem areas in the manual. (Identify errors by page, section, figure, or table number wherever possible. Specify additional index entries.) ____________________

____
Your Name   Date

____
Organization

____
Street Address

____
City   State   Zip

No postage necessary if mailed in the U.S.

cut or fold along dotted line

fold

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 78 CHELMSFORD, MA 01824

POSTAGE WILL BE PAID BY ADDRESSEE

APOLLO COMPUTER INC.
Technical Publications
P.O. Box 451
Chelmsford, MA 01824

fold

Reader's Response

Please take a few minutes to send us the information we need to revise and improve our manuals from your point of view.

Document Title: *Domain/OS Call Reference, Volume 2*
Order No.: 012888-A00
Date of Publication: June, 1988

What type of user are you?

_____ System programmer; language _______________
_____ Applications programmer; language _______________
_____ System maintenance person
_____ System Administrator
_____ Student Programmer
_____ Other
_____ Manager/Professional
_____ Technical Professional
_____ Novice

How often do you use the Domain system?_______________

What parts of the manual are especially useful for the job you are doing?_______________

What additional information would you like the manual to include?_______________

Please list any errors, omissions, or problem areas in the manual. (Identify errors by page, section, figure, or table number wherever possible. Specify additional index entries.)_______________

Your Name _______________ Date

Organization

Street Address

City _______________ State _______________ Zip

No postage necessary if mailed in the U.S.

ut or fold along dotted line

fold

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 78 CHELMSFORD, MA 01824

POSTAGE WILL BE PAID BY ADDRESSEE

APOLLO COMPUTER INC.
Technical Publications
P.O. Box 451
Chelmsford, MA 01824

fold

Reader's Response

Please take a few minutes to send us the information we need to revise and improve our manuals from your point of view.

Document Title: *Domain/OS Call Reference, Volume 2*
Order No.: 012888-A00
Date of Publication: June, 1988

What type of user are you?

_____ System programmer; language ______________________

_____ Applications programmer; language __________________

_____ System maintenance person _____ Manager/Professional

_____ System Administrator _____ Technical Professional

_____ Student Programmer _____ Novice

_____ Other

How often do you use the Domain system?______________________________

What parts of the manual are especially useful for the job you are doing?________________

What additional information would you like the manual to include?________________

Please list any errors, omissions, or problem areas in the manual. (Identify errors by page, section, figure, or table number wherever possible. Specify additional index entries.)________________

Your Name Date

Organization

Street Address

City State Zip

No postage necessary if mailed in the U.S.

cut or fold along dotted line

fold

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 78 CHELMSFORD, MA 01824

POSTAGE WILL BE PAID BY ADDRESSEE

APOLLO COMPUTER INC.
Technical Publications
P.O. Box 451
Chelmsford, MA 01824

fold